OpenEdge® Data Management: SQL Development
Notices

© 2015 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.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Deliver More Than Expected, Easyf, Fathom, Icenium, Kendo UI, Making Software Work Together, OpenEdge, Powered by Progress, Progress, Progress Control Tower, Progress RPM, Progress Software Business Making Progress, Progress Software Developers Network, Rollbase, RulesCloud, RulesWorld, SequeLink, Sitefinity (and Design), SpeedScript, Stylus Studio, TeamPulse, Telerik, (Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, AppsAlive, AppServer, BravePoint, BusinessEdge, DataDirect Spy, DataDirect SupportLink, Future Proof, High Performance Integration, Modulus, NativeScript, OpenAccess, Pacific, ProDataSet, Progress Arcade, Progress Pacific, Progress Profiles, Progress Results, Progress RFID, Progress Progress Software, ProVision, PSE Pro, SectorAlliance, Sitefinity, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress 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.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

The Release Notes can be found in the OpenEdge installation directory and online at:

For the latest documentation updates see OpenEdge Product Documentation on Progress Communities: (https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx).

October 2015

Last updated with new content: Release 11.6.0
Table of Contents

Preface.........................................................................................................................13
  Purpose..................................................................................................................13
  Audience.............................................................................................................13
  Organization.......................................................................................................14
  Typographical conventions..................................................................................15
  Examples of syntax diagrams (SQL).................................................................16
    Long SQL syntax descriptions split across lines..............................................17

Chapter 1: Introduction.........................................................................................19
  An overview of OpenEdge SQL........................................................................19
  OpenEdge SQL client/server architecture......................................................20
    Multi-threaded architecture..........................................................................21

Chapter 2: JDBC Client.......................................................................................23
  Introduction to the JDBC client........................................................................23
    JDBC architecture.........................................................................................24
    JDBC components.........................................................................................24
    JDBC API support.........................................................................................25
    Internet Protocol Support..............................................................................25
  Setting environment variables..........................................................................26
    Setting environment variables in a character environment..........................26
    Setting environment variables in Windows..................................................26
  Connecting to an OpenEdge database with a JDBC driver.............................27
    Connecting using SQL Explorer....................................................................27
    Connecting from a Java application using a URL..........................................28
      Database connection examples....................................................................28
    Connecting from a Java application using a data source..............................31
    Enabling encryption.......................................................................................31
      EncryptionMethod.......................................................................................32
      ValidateServerCertificate..........................................................................32
      HostNameInCertificate................................................................................32
      TrustStore and TrustStorePassword..........................................................32
    JDBC connection parameters.........................................................................34
    Troubleshooting database connection problems...........................................34

Chapter 3: ODBC Client......................................................................................37
  Overview of ODBC............................................................................................37
Chapter 4: Data Control Language and Security.................................53

Working with database security.................................................................53
Comparing OpenEdge SQL and ABL security........................................54
OpenEdge SQL security........................................................................54
ABL security.......................................................................................54
Comparing authentication and authorization........................................55
Authentication......................................................................................55
Authorization.......................................................................................56
Creating users......................................................................................56
Creating database administrators.........................................................57
Creating users in the _User table..........................................................57
SQL Only Users..................................................................................58
Create an SQL only user.......................................................................59
Alter a user to an SQL only user..........................................................59
Granting privileges.............................................................................60
Privilege basics..................................................................................60
GRANT statement...............................................................................60
Database-wide privileges.....................................................................61
Table-specific privileges.....................................................................62
Granting public access........................................................................63
Verifying privileges............................................................................63
Revoking privileges............................................................................63

Chapter 5: OpenEdge SQL Data Definition Language.........................67

Using Data Definition Language statements........................................67
Working with tables............................................................................68
CREATE TABLE..................................................................................68
ALTER TABLE....................................................................................72
DROP TABLE......................................................................................................................76
Working with indexes......................................................................................................................76
  CREATE INDEX ......................................................................................................................76
  DROP INDEX........................................................................................................................78
Using CURRVAL and NEXTVAL in a statement........................................................................81
  ALTER SEQUENCE..............................................................................................................82
Enabling 64-bit sequences......................................................................................................81
  DROP SEQUENCE..............................................................................................................82
Handling cycles in referential integrity................................................................................97
  ALTER SEQUENCE..............................................................................................................82
Creating tables in cycles......................................................................................................98
  SHOW PARTITION..............................................................................................................91
Inserting rows in a cycle........................................................................................................98
  SHOW GROUP...................................................................................................................89
  DROP GROUP....................................................................................................................89
  ALTER GROUP..................................................................................................................90
Working with groups...........................................................................................................88
  DROP TENANT...................................................................................................................87
  CREATE TENANT................................................................................................................85
  CREATE SUPER-TENANT......................................................................................................86
Candidate keys.........................................................................................................................95
  ALTER TENANT..................................................................................................................86
Primary keys..........................................................................................................................95
  DROP TENANT...................................................................................................................87
Check constraints................................................................................................................93
  SHOW TENANT....................................................................................................................88
Column-level check constraints.........................................................................................93
  SHOW PARTITION..............................................................................................................91
Table-level check constraints............................................................................................94
  CREATE GROUP..................................................................................................................88
Candidate keys.........................................................................................................................95
  DROP GROUP....................................................................................................................89
Primary keys..........................................................................................................................95
  ALTER GROUP..................................................................................................................90
Referential constraints..........................................................................................................96
  SHOW GROUP...................................................................................................................89
Foreign key constraint.........................................................................................................96
  DROP TENANT...................................................................................................................87
Handling cycles in referential integrity.............................................................................97
  DROP TENANT...................................................................................................................87
Creating tables in cycles......................................................................................................98
  DROP TENANT...................................................................................................................87
Inserting rows in a cycle........................................................................................................98
  DROP TENANT...................................................................................................................87
Working with SQL utilities....................................................................................................99
  DROP TENANT...................................................................................................................87
Using the SQLDUMP utility .................................................................................................99
  DROP TENANT...................................................................................................................87
Chapter 8: Data Control Language and Transaction Behavior

Working with transaction control............................................................145
   COMMIT statement..............................................................................146
   ROLLBACK statement.........................................................................146
Transactions and isolation levels..........................................................147
   Dirty read............................................................................................147
   Nonrepeatable read............................................................................147
   Phantom read.....................................................................................148
Setting isolation levels...........................................................................148
Understanding transactions and locking..............................................149
   Lock modes.......................................................................................149
   How lock levels and lock modes interact .........................................150
Understanding lock acquisition...........................................................151
   Information schema locks..................................................................151
   Table and record locks......................................................................152
Enhancing performance with locking hints.........................................152
   The READPAST locking hint.............................................................152
Monitoring locking and database performance....................................153
Online schema changes........................................................................154
Authorized data truncation.................................................................155
   Logging.............................................................................................157
   Writing to database log file..............................................................157
   Logging instances.............................................................................157
   SQL Utility update to prevent data loss.............................................159
Autonomous Schema Update.................................................................159

Chapter 9: Performing Multi-database Queries

Multi-database query overview...........................................................161
   The process of multi-database queries..............................................161
   1. Defining the databases...............................................................162
   2. Ensuring permissions to access databases to be queried............162
   3. Connecting to the databases......................................................162
Chapter 10: Working with JTA Transactions

JTA’s role in J2EE
Understanding JTA architecture
Understanding application interfaces
XDataSource
XACConnection
XRResource
XRResource methods
JTA and the distributed transaction process
JTA transactions and two-phase commit protocol
JTA transactions and conventional transactions
JTA transactions and crash recovery
JTA transactions and OpenEdge Replication
Planning for JTA transaction support
JTA transactions and database resource planning
Enabling JTA support
Disabling JTA support
Monitoring JTA transactions
Resolving JTA transactions
Chapter 11: Stored Procedures and Triggers

11 OpenEdge Data Management: SQL Development
Chapter 12: Optimizing Query Performance.........................................................209

Understanding optimization..........................................................................................209
How the query optimizer works......................................................................................210
Representing the statement as a query tree.................................................................210
The statement parser....................................................................................................211
Quantified predicates and other subqueries .....................................................................211
Views ............................................................................................................................212
Optimizer phases..........................................................................................................212
Early evaluation of constant expressions........................................................................212
GROUP BY optimization...............................................................................................213
   Re-order GROUP BY columns......................................................................................213
   Considering Equi-Constant Predicates.......................................................................213
Pushing restrict operations close to the data origin.........................................................214
Using indexes for restrictions .........................................................................................214
Choosing the best index .................................................................................................214
Predicate expressions ....................................................................................................214
Generating candidate indexes .......................................................................................214
Selecting an index ............................................................................................................215
Providing index hints.......................................................................................................215
Join optimization.............................................................................................................215
Determining join order among adjacent join nodes.........................................................215
Choosing the join algorithm ...........................................................................................216
Augmented nested loop join ...........................................................................................216
Merge join .......................................................................................................................216
Nested loop join ..............................................................................................................216
Sort optimization.............................................................................................................217
Eliminating redundant sorts ............................................................................................217
Converting table scans to index bracket scans ...............................................................217
Indexes to evaluate MAX/MIN functions .......................................................................217
Index bracket scan optimization.....................................................................................217
Inspecting what the optimizer produces..........................................................................218
   The _Sql_Qplan virtual system table .........................................................................218
Affecting what the optimizer produces............................................................................219
   Working with the UPDATE STATISTICS command..................................................219
SQL use of index statistics..............................................................................................219
Updating index statistics.................................................................................................220

Index................................................................................................................................223
Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Typographical conventions
- Examples of syntax diagrams (SQL)

Purpose

OpenEdge Data Management: SQL Development provides information for developers who are using SQL within the OpenEdge® application development environment. The information in this manual is also useful for database administrators, and, to a lesser degree, to application end users.

Audience

The audience for this book is composed of three specific groups:

- Database Administrators who 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 who will use the book to:
  • Manage database connections and set up data sources.
  • Create queries to draw information from the database.
  • Tune database queries.
  • Develop application business logic.

• End users who will use the book to:
  • Connect to databases.
  • Call stored procedures.
  • Issue queries.
  • Understand error messages.

Organization

Introduction on page 19
Provides an overview of OpenEdge SQL and the OpenEdge SQL client/server architecture.

JDBC Client on page 23
Presents an overview of the JDBC client, details on setting environment variables, and procedures for connecting to a database with the JDBC driver.

ODBC Client on page 37
Offers an overview of the ODBC client and information on configuring data source.

Data Control Language and Security on page 53
Reviews information on working with database security, creating users, and granting, modifying, and revoking privileges.

OpenEdge SQL Data Definition Language on page 67
Furnishes information on OpenEdge SQL database structure and methods for creating, altering, and dropping database objects.

OpenEdge SQL Data Manipulation Language on page 113
Provides information on the Data Manipulation Language statements, indexes, and join operations.

OpenEdge SQL and Advanced Business Language Interoperability on page 131
Addresses the interoperability of ABL and OpenEdge SQL.

Data Control Language and Transaction Behavior on page 145
Summarizes information on transactions, isolation levels, and locking.

Performing Multi-database Queries on page 161
Provides information on the performance of multi-database queries.
Typographical conventions

This manual uses the following typographical and syntax conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td>SMALL, BOLD CAPITAL LETTERS</td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td>KEY1+KEY2</td>
<td>A plus sign between key names indicates a simultaneous key sequence: you press and hold down the first key while pressing the second key. For example, CTRL+X.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>A space between key names indicates a sequential key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
</tbody>
</table>

**Syntax:**

| Fixed width                 | A fixed-width font is used in syntax, code examples, system output, and file names. |
| Fixed-width italics         | Fixed-width italics indicate variables in syntax. |
| Fixed-width bold            | Fixed-width bold italic indicates variables in syntax with special emphasis. |
| UPPERCASE fixed width       | ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class. |
| Period (.) or colon (:)     | All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon. |
### Examples of syntax diagrams (SQL)

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:

```
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:

```
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:

```
CREATE [ PUBLIC ] SYNONYM synonym
FOR [ owner_name. ] { table_name | view_name | synonym };
```
In this example, you must specify `table_name` or `view_name`:

```
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:

```
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 ] ;
```
OpenEdge® SQL is part of an open, standards-based interface for the OpenEdge application development platform, providing you with the ability to quickly and efficiently develop, deploy, integrate, and manage world-class business applications.

This chapter offers a basic introduction to the features and capabilities of OpenEdge SQL, as described in the following sections.

For details, see the following topics:

• An overview of OpenEdge SQL
• OpenEdge SQL client/server architecture

An overview of OpenEdge SQL

OpenEdge offers essential flexibility in developing software solutions by providing a high performance relational database and an open, standards-based interface for SQL. The OpenEdge database and its SQL interface provide an environment that enables efficient integration with third-party tools, such as development software, reporting, and Online Analytical Processing (OLAP) tools.

Featuring a state-of-the-art, cost-based query optimizer, the OpenEdge SQL Engine incorporates APIs for ODBC and JDBC.

OpenEdge SQL consists of the following components:

• **OpenEdge SQL Engine** — The OpenEdge SQL Engine is installed as part of the OpenEdge Relational Database Management System. To support the OpenEdge Service Oriented Architecture, the SQL engine offers robust data type support, enables online schema changes,
OpenEdge SQL client/server architecture

The OpenEdge SQL Engine is designed to support software requirements for deployed applications. OpenEdge implements the SQL interface in a client/server configuration. OpenEdge SQL Access consists of:

- An SQL database engine
- A client application accessing the database through two available interfaces:
  - A JDBC API
  - An ODBC API

Because the SQL interface works through a client/server connection, it is integrated with the OpenEdge client/server architecture. Advanced Business Language (ABL), created and developed by Progress Software Corporation, is the programming language used to develop OpenEdge applications.

The following figure illustrates the architecture of the ABL and OpenEdge SQL clients when connected to the database in a client/server configuration.

**Figure 1: Client/server architecture**

In this configuration:

- Both SQL servers and ABL servers are able to connect with the OpenEdge database.
• The SQL server integrates with OpenEdge processes and the Admin Service through which it can be started.

• The SQL server processes always perform database operations through the shared memory attached to the database. The shared memory is common to all processes running against a specific OpenEdge database.

• The SQL database servers are only accessible from SQL clients. The connection is always established through the network layer from the SQL clients.

**Note:** The multiple volumes, located on the right side of the above figure, symbolize the fact that the database is multi-volume.

### Multi-threaded architecture

OpenEdge supports a multi-threaded architecture (for SQL and utilities only), multi-process architecture, which provides multiple paths to a database. A multi-threaded, multi-process architecture provides specific functionality for a database in the following ways:

• Each local client or self-serving client can access the database and service its own requests.

• Each database server listens for and runs requests for one or more remote clients. The database broker initializes shared memory and either starts a new server for each additional client or set of clients that access the database or dispatches additional clients to an existing server.

• All machines that run operating systems with shared memory can run a SQL server as multi-threaded, enabling multiple remote clients on a network to access the database simultaneously using shared memory.
JDBC Client

OpenEdge uses a JDBC client provided by DataDirect Technologies. The JDBC client enables programs written in Java to access the OpenEdge SQL Engine. This chapter provides an overview of the JDBC client and describes configuration procedures, as outlined in the following sections.

Note: Information in this chapter is presented with the assumption that you are familiar with Java and JDBC and have some knowledge of setting environmental variables.

For details, see the following topics:

• Introduction to the JDBC client
• Setting environment variables
• Connecting to an OpenEdge database with a JDBC driver

Introduction to the JDBC client

The JDBC client is installed with OpenEdge Client Access. For more information on installation procedures, see OpenEdge Getting Started: Installation and Configuration.

Java Database Connectivity (JDBC) is a Java application programming interface (API) that allows SQL statements to perform database operations. The JDBC API consists of classes and interfaces written in Java. The JDBC API allows developers to write portable database applications using pure Java. These applications are portable because database-specific JDBC drivers convert the JDBC API call to a database-specific call.
JDBC architecture

OpenEdge uses a Type 4 JDBC driver. The following figure shows the architecture of a Java application using a Type 4 JDBC driver.

Figure 2: OpenEdge JDBC Type 4 driver architecture

In the following figure, the Java application includes calls to the JDBC API. A JDBC API call must be performed using either the `DriverManager.getConnection` or the `DataSource.getConnection` method. The `getConnection` method obtains a connection to the appropriate JDBC Driver. The `DriverManager` or `DataSource` class is used to manage that connection.

JDBC components

The JDBC architecture consists of several files. The JDBC driver includes the `openedge.jar` AND `pool.jar` files. The following table lists the locations of the files.
Table 1: JDBC driver component file locations

<table>
<thead>
<tr>
<th>Platform</th>
<th>Files and locations</th>
</tr>
</thead>
</table>
| Windows  | %DLC%\java\openedge.jar  
|          | %DLC%\java\pool.jar |
| Sun Solaris SPARC (32 bit and 64 bit) | $DLC/java/openedge.jar  
| Compaq Tru64 UNIX/Linux X86IBM AIXHP-UX (32 bit and 64 bit) | $DLC/java/openedge.jar  
|          | $DLC/java/pool.jar |

DLC is the directory where OpenEdge is installed on the machine. In Windows, the DLC environment variable is set in the Registry during the installation of OpenEdge. On UNIX, you must set the DLC environment variable yourself. If an environment variable is modified while the database is up, then the database must be shut down and restarted so the database broker and the SQL engine can see the new value.

**JDBC API support**

The JDBC driver contains the core JDBC 3.0 APIs and some extended features. The JDBC 3.0 API includes the `java.sql` package, which contains the core APIs, and the `javax.sql` package, which contains the extended features of JDBC 3.0.

The OpenEdge SQL Engine supports the Java Transaction API (JTA) of the J2EE framework architecture. Support for JTA enables the OpenEdge database to participate in distributed SQL transactions. With distributed JTA transactions, the OpenEdge database takes the role of the resource manager in the J2EE framework, and relies on an external transaction manager to coordinate the commit or rollback of the distributed transactions.

For a complete list of the APIs supported by the JDBC driver, see *OpenEdge Data Management: SQL Reference*.

**Internet Protocol Support**


The driver accepts an IPv6 address in both the server name portion of the connection URL and as a value for the server name data source property. The driver also accepts a host name which resolves to an IPv6 address and connects to the resolved IPv6 address. When specifying an IPv6 address in the connection URL, the address must be enclosed in square brackets. This notation, specified in RFC 3986 "Format for Literal IPv6 Addresses in URLs", allows the driver to distinguish between the IPv6 address and the port number to connect to at that address. For example the following connection URL specifies that the driver is to connect port 1433 at the IPv6 address 2001:0db8:85a3:08d3:1319:8a2e:0370:7344:

```
```
Additionally, the OpenEdge JDBC driver supports mixed mode operation. A driver running on an IPv6 client can connect to a server with an IPv4 address and a driver running on an IPv4 client can connect to a server with an IPv6 address provided the addresses are suitable for mixed mode operations.

IPv6 addresses will be supported when connecting to Release 10.1C databases or later. To support IPv6 addresses, the OpenEdge 10.1C JDBC driver must be running in J2SE 1.4 or later for Solaris and Linux platforms and must be running in J2SE 5.0 or later for Windows platforms.

The SQLDump, SQLLoad, and SQLSchema utilities and ESQLC clients should enclose IPv6 addresses in brackets when they are specified on a connection URL. For example, when specifying the IPv6 address of localhost:

```
progress:T:[::]:2800:mydb
```

**Setting environment variables**

The `CLASSPATH` must be set on each client machine. The `CLASSPATH` points to the location of OpenEdge JDBC driver classes.

The following sections offer procedures for setting environment variables in either a character environment or Windows.

**Setting environment variables in a character environment**

Before you can use the JDBC driver, you must first set your environment variables.

To set an environment variable in a character environment, use the following command as an example. This command can be either typed at a system prompt or placed in a file that is run during login to the system, as shown below:

```
CLASSPATH=$DLC/java/openedge.jar: $CLASSPATH
```

**Setting environment variables in Windows**

Work from the Settings menu of your Windows environment to set your JDBC driver environment variables.

To set the environment variable in Windows:

1. Select **Start** > **Settings** > **Control Panel** > **System**. The **System Properties** window appears.
2. Select the **Advanced** tab and click **Environment Variables**. The **Environment Variable** window appears.
3. At the **Environment Variables** box, click **New** under the **System variables** group box.

   The **New System Variable** dialog box appears:
4. In the **Variable name** field, type **CLASSPATH**. In the **Variable value** field, type:
   `%CLASSPATH%;%DLC%\java\openedge.jar; %DCL%\java\util.jar; %DLC%\java\base.jar;%CLASSPATH%`.

5. Click **OK**.

6. In the **Environment Variable** window, click **OK**.

---

**Connecting to an OpenEdge database with a JDBC driver**

You can connect to an OpenEdge database through either the SQL Explorer or a JDBC application.

For information on connecting to multiple databases in order to perform multiple-database queries, see [Performing Multi-database Queries](#) on page 161

---

**Connecting using SQL Explorer**

SQL Explorer is a Java-based tool you can use to execute OpenEdge SQL statements, shell commands, and client control commands.

To start the SQL Explorer tool and connect to an OpenEdge database, execute a command using one of the following syntax examples:

```
sqlexp -db database-name -S port | service-name -H host -user userid -password password
```

```
sqlexp -url jdbc:datadirect:openedge://host:port;databaseName=dbname;servicename=service_name -user userid -password password
```

---

**Notes**

- The `-db` preceding the `database-name` is optional. `host` is optional and defaults to `localhost`, which is valid only for nonremote databases.
- The `user` is optional and defaults to the current user. The user’s password is also optional and defaults to null.
- The current user, when using the default, must be the user who created the OpenEdge database.
- In Windows and on UNIX, a user and password are optional until the first user is created in the OpenEdge database.
• You can use an encrypted password. Follow the procedure for generating and using encrypted passwords in the Password(-P) section of OpenEdge Deployment: Startup Command and Parameter Reference.

• Once the CREATE USER statement is executed, a user and password must be provided.

Connecting from a Java application using a URL

A Java application must perform two steps to connect to an OpenEdge database through a JDBC driver:

1. It must load the JDBC driver.
2. It must connect to the JDBC driver

To load the JDBC driver, call the Class.forName method in the JDBC application. The Class.forName method takes the fully qualified class name of the JDBC driver as its argument. The fully qualified class name for the JDBC driver is com.ddtek.jdbc.openedge.OpenEdgeDriver.

This class name is case-sensitive and must be typed exactly as shown. The Class.forName method also registers the specified JDBC driver with the Driver Manager class so that the driver is available for connections.

Here is an example of the call needed to load and register the driver:

```
CLASS.FORNAME ( "com.ddtek.jdbc.openedge.OpenEdgeDriver");
```

Now that the JDBC driver has been loaded and registered, the DriverManager.getConnection method must be called to establish a connection to the database. The getConnection method takes three arguments:

• A string containing a URL. The Driver Manager uses this URL to find a driver that can connect to the database represented by the given URL. Once the driver is found, the URL is used by the Driver class to establish the connection to the database.

OpenEdge syntax for the URL string is:

```
jdbc:datadirect:openedge://host:port;databaseName=db_name;
servicename=service_name;defaultSchema=schema_name;
statementCacheSize=Cachesize;
```

See the JDBC URL connection string in the following section for an explanation of each component.

• The ID of the user trying to connect to the database.

• The user's password.

Database connection examples

Several variations of the call needed to connect to the database follow:
• Variation 1:

```java
Connection con = DriverManager.getConnection ( url, myuserid, mypassword );
```

This is an example of how to connect to the OpenEdge database using Variation 1:

```java
String url;
String myuserid;
String mypassword;
url = new String ( "jdbc:datadirect:openedge://myhost:6718;databaseName=sports2000" );
myuserid = new String ( "jones" );
mypassword = new String ( "secret" );
Connection con = DriverManager.getConnection ( url, myuserid, mypassword );
```

• Variation 2:

```java
Connection con = DriverManager.getConnection ( url, info );
```

In this variation info is a Properties object that contains a string of tag/value pairs used for connecting to the database. Normally the info object would include at least the user ID and password. The password is optional and defaults to NULL. However, once the CREATE USER statement has been executed and a user is created in the database, a password is required.

The JDBC driver expects the user ID tag to be named user and the password tag to be named password. The user and password tags are case-sensitive and must be in all lowercase letters.

This is an example of how to connect to the database using Variation 2 (tag/value pairs):

```java
String url;
String myuserid;
String mypassword;
url = new String ( "jdbc:datadirect:openedge://myhost:6718;databaseName=sports2000" );
myuserid = new String ( " jones " );
mypassword = new String ( "secret " );
java.util.Properties info = new java.util.Properties ( );
info.put ( "user ", myuserid );
info.put ( "password ", mypassword );
Connection con = DriverManager.getConnection ( url, info );
```

• Variation 3:

```java
Connection con = DriverManager.getConnection ( url );
```

This variation takes only the URL as an argument. The URL in this case contains the user and password tags shown in Variation 2. The password is optional and defaults to NULL. However, once the CREATE USER statement has been executed and a user is created in the OpenEdge database, a password is required.
The JDBC driver expects the user ID tag to be named user and the password tag to be named password. The user and password tags are case sensitive and must be in all lowercase letters.

This is an example of how to connect to the OpenEdge database using Variation 3:

```java
String url;
url = new String ("jdbc:datadirect:openedge://myhost:6718;databaseName=sports2000;user=jones;password=secret");
Connection con = DriverManager.getConnection(url);
```

- **JDBC URL connection string**

  The OpenEdge JDBC URL string has the following syntax:

  ```java
  jdbc:datadirect:openedge://host:port;databaseName=db_name;
servicename=service_name;
defaultSchema=schema_name;
statementCacheSize=CacheSize;
  ```

  This is an example of a connection string:

  ```java
  jdbc:datadirect:openedge://myhost:6718;databaseName=sports2000;
  ```

  This is an example of a connection string, using the optional service_name parameter:

  ```java
  jdbc:datadirect:openedge://myhost:-1;databaseName=sports2000;
servicename=myservice;defaultSchema=schema_name;statementCacheSize=CacheSize;
  ```

  The components of the URL string are:

  - **jdbc:datadirect:openedge://**
    - jdbc is the protocol to be used. The protocol in a JDBC URL is always jdbc.
    - datadirect is the subprotocol and it designates the name of the JDBC driver.
    - openedge indicates that the driver is for OpenEdge.
  - **host**
    - The name of the host on which the OpenEdge database resides. If this is not specified it defaults to localhost (localhost is valid only if the database is not remote). The host is myhost in the example.
  - **port**
  - Port number or service name to be used for the connection. The port is 6718 in the first example.

  **Note:** If you specify the service_name, the port must be -1, as shown in the second example.
• **db_name**
  - The name of the database. The `db_name` is `sports2000` in the example.

• **service_name**
  - An optional parameter, indicating the name of the service. The `service_name` is `myservice` in the example. If you specify `service_name`, the `port` must be `-1`.

• **schema_name**
  - Indicates the schema to be used during statement processing. For more information about the `defaultSchema` connection parameter, see the.

• **CacheSize**
  - Indicates how many entries will be in the statement cache. For more information about the `statementCacheSize` connection parameter, see the.

### Connecting from a Java application using a data source

You can connect to a database from a Java application using a data source. There are required settings and optional settings when connecting using a data source. The following table lists the settings.

<table>
<thead>
<tr>
<th>Required settings</th>
<th>Optional settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>oeds.setServerName(&quot;host&quot;);</td>
<td>oeds.setServiceName(&quot;name&quot;);</td>
</tr>
<tr>
<td>oeds.setDatabaseName(&quot;name&quot;);</td>
<td>oeds.setDescription(&quot;text&quot;);</td>
</tr>
<tr>
<td>oeds.setPortNumber(number);</td>
<td>oeds.setDataSourceName(&quot;name&quot;);</td>
</tr>
<tr>
<td>oeds.setUser(&quot;user&quot;);</td>
<td>oeds.setStatementCacheSize(cache_size);</td>
</tr>
<tr>
<td>oeds.setPassword(&quot;password&quot;);</td>
<td>oeds.setDefaultSchema(&quot;name&quot;);</td>
</tr>
</tbody>
</table>

**Note:** The `oeds` in the settings signifies an OpenEdge data source.

To connect using a data source, specify the connection type (connection, pooled, or XA) and specify the settings.

### Enabling encryption

OpenEdge supports data privacy and client/server authentication over connections between OpenEdge clients, servers, and middleware using the Secure Sockets Layer (SSL). You can use the following connection options to enable encryption for the JDBC driver. For more information on encryption, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.
EncryptionMethod

The value of encryptionMethod determines if and how the driver encrypts and decrypts the data sent between the driver and the database server. The value of the encryptionMethod option can be one of the following values:

- **NoEncryption**—The data which flows between the driver and the server is not encrypted.
- **SSL**—Require SSL encryption be used. If the server the driver is connecting to does not support SSL encryption, the connection attempt fails.

The default value is NoEncryption. The OpenEdge driver will throw an invalid option value exception if any value other than NoEncryption or SSL is specified.

ValidateServerCertificate

The validateServerCertificate option determines whether the driver will validate the server certificate returned by the database server while establishing the SSL connection. The validateServerCertificate option can have the following values:

- **False**—The driver will not validate the certificate returned by the database server. This is useful in test and debug environments because it eliminates the need to specify a trust store on all of the client machines in the test environment.
- **True**—The driver validates the certificate returned by the database server. The certificate returned must have been issued from a certificate authority (CA) that is included in the trust store. The driver also validates the ServerName in the certificate as specified by the HostNameInCertificate option. The value of ValidateServerCertificate is ignored if the value of the encryptionMethod connection option does not specify SSL encryption. The default value is false.

HostNameInCertificate

This option ensures that the driver is connecting to the requested server. The value of hostNameInCertificate is ignored if validateServerCertificate is set to false or if the encryptionMethod option does not specify an SSL encryption method.

If a name is specified for hostNameInCertificate, the driver examines the values included in the certificate. If certificate’s name value does not match the name specified for hostNameInCertificate, the connection will fail. The driver will not check the host name in the server certificate if the hostNameInCertificate is not specified or is set to an empty string. The default value is the empty string.

TrustStore and TrustStorePassword

When using basic server authentication with SSL, the certificate returned by the server to the client must have been issued by a certifying authority that is trusted by the client for the SSL session to be established. Certifying authorities trusted by the client reside in an encrypted file called a trust store. Most if not all trust stores are password protected. The Java platform defines standard system properties for specifying the location of the trust store and the value of the trust store password. These properties are:

- `javax.net.ssl.trustStore`
• javax.net.ssl.trustStorePassword

These can be specified on the java command line as:

-Djavax.net.ssl.trustStore=C:\Certificates\MyTrustStore
-Djavax.net.ssl.trustStorePassword=myPassword

To enable the trust store and trust store password, the driver employs two connect options - trustStore and trustStorePassword. The value of the trustStore option is a pathname which specifies the location of the trust store file. The value of the trustStorePassword is the password required to access the contents of the trust store. The values specified for the trustStore and trustStorePassword connect options override any value specified by the corresponding Java system property. If a trustStore or trustStorePassword option is not specified, the driver recognizes any value specified for the corresponding system property.

The following table provides descriptions of the methods used to enable encryption in the driver.

Table 3: Encryption methods for the JDBC Driver

<table>
<thead>
<tr>
<th>Encryption Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setEncryptionMethod</td>
<td>Specifies the driver encryption method. The encryption method determines whether the driver encrypts and decrypts the data sent between the driver and the data server.</td>
</tr>
<tr>
<td>getEncryptionMethod()</td>
<td>Returns the driver encryption method.</td>
</tr>
<tr>
<td>setValidateServerCertificate</td>
<td>Specifies whether the driver will validate the server certificate returned by the database server.</td>
</tr>
<tr>
<td>getValidateServerCertificate()</td>
<td>Returns whether the driver will validate the server certificate returned by the database server.</td>
</tr>
<tr>
<td>setHostNameCertificate</td>
<td>Specifies the name the driver will use to compare with the certificate common name returned by the database server during SSL session establishment.</td>
</tr>
<tr>
<td>getHostNameCertificate</td>
<td>Returns the name the driver will use to compare with the certificate common name returned by the database server.</td>
</tr>
<tr>
<td>setTrustStore</td>
<td>Specifies the full path to the trust store file which contains the certificate authorities trusted by the driver.</td>
</tr>
<tr>
<td>getTrustStore()</td>
<td>Returns the path to the trust store.</td>
</tr>
<tr>
<td>setTrustStorePassword</td>
<td>Specifies the password used to gain access to the trust store.</td>
</tr>
<tr>
<td>getTrustStorePassword</td>
<td>Returns the trust store password.</td>
</tr>
</tbody>
</table>
JDBC connection parameters

When connecting to an OpenEdge database using a JDBC driver, you can specify two optional connection parameters:

- `defaultSchema` — The schema to be used during statement processing
- `statementCacheSize` — The number of entries in the statement cache

Here is an example of how to specify these connection parameters:

```
jdbc:datadirect:openedge//myhost:6718;databaseName=sports2000;defaultSchema=schema1;statementCacheSize=10;
```

Troubleshooting database connection problems

This section identifies common errors that occur while connecting to a database from a Java application using JDBC APIs, the possible causes of the errors, and the solutions. For example:

- **Error** — [JDBC OpenEdge Driver]:Error in Network Daemon (8933)
  - **Cause** — A `host-name` of `localhost` was specified, either by default or through the `-H` option, when starting the OpenEdge database. This prevents connections from clients on other machines. Connections from only those clients that reside on the same machine as the database are allowed.
  - **Solution** — Shut down the database and restart it using the actual name of the host on which the database resides.

- **Error** — error in tcp bind 10061
  - **Cause** — The port number or service name used in the URL to connect to the database is not the same as the one used to start the database.
  - **Solution** — Modify the port number or service name in your URL to match the one with which the database was started. This is designated by the `-S` option when starting the database.

- **Error** — No suitable driver
  - **Cause** — The `CLASSPATH` is not specified correctly or is not set at all. The class name of the JDBC driver string, passed to the `CLASS.FORNAME` method, might be incorrect. The URL string might be incorrect.
  - **Solution** — Use the following guidelines for setting the `CLASSPATH`:
    - In Windows, ensure the `CLASSPATH` includes `%DLC%\java\openedge.jar`
    - On UNIX, ensure the `CLASSPATH` includes `$DLC/java/openedge.jar`
    - Check the JDBC driver string to see if it matches `com.ddtek.jdbc.openedge.OpenEdgeDriver`
    - Check the URL string to see if it complies with the syntax of either:
Connecting to an OpenEdge database with a JDBC driver

- jdbc:datadirect:openedge://host_name:port;
databaseName=database_name, or

- jdbc:datadirect:openedge://host_name:port;
databaseName=database_name;servicename=service_name
ODBC Client

The ODBC (Open Database Connectivity) Driver for the OpenEdge SQL engine is supplied by DataDirect Technologies. The driver is installed with OpenEdge SQL Client Access and provides ODBC client access to the OpenEdge RDBMS.

This chapter provides a brief overview of the ODBC interface. It also provides configuration procedures needed to make data source connections. Procedures are offered for both Windows and UNIX settings, as described in the following sections.

For details, see the following topics:

- Overview of ODBC
- Configuring data sources

Overview of ODBC

The ODBC interface is a C-based interface that has emerged as an industry standard that enables applications to access data from different sources. The ODBC API provides you with a standard interface to interact with relational database systems.

To become accessible from ODBC client applications, databases must provide an ODBC driver. The driver translates ODBC function calls into calls the data source can process, and returns data to the application. The OpenEdge SQL Engine provides the DataDirect ODBC driver to enable client access to the OpenEdge RDBMS.

For the most up-to-date information on the ODBC driver supported platforms, see the OpenEdge Getting Started: Installation and Configuration.
ODBC architecture

The ODBC architecture consists of the following components:

- **Application** — An ODBC application is any program that calls ODBC functions and uses them to issue SQL statements.

- **ODBC driver manager** — The driver manager routes calls from an application to the ODBC driver. The ODBC driver manager loads the requested driver in response to an application's call to the ODBC `SQLConnect` or `SQLDriverConnect` functions.

- **ODBC driver** — An ODBC driver is a dynamic link library (DLL) or a shared library that processes ODBC function calls for a specific data source. The driver connects to the data source, translates the standard SQL statements into syntax the data source can process, and returns data to the application.

- **Data source** — A data source is a combination of a database system, the operating system it uses, and any network software required to access it.

The following figure shows the components of an ODBC environment.

**Figure 3: Components of an ODBC environment**
Configuring data sources

This section provides information for configuring Windows and UNIX clients to use the ODBC driver for the OpenEdge SQL Engine. It identifies information about your system that you will need prior to configuration. UNIX environment variable requirements are explained. Setting environment variables is not required for Windows clients. Procedures for configuration are presented.

For information on connecting to multiple databases in order to perform multiple-database queries, see Performing Multi-database Queries on page 161

Internet Protocol support


Configuring Windows clients

Windows clients are configured by using the ODBC Administrator, a Microsoft utility. Use this utility to configure the Windows client, establish a new data source for the client, and connect to that data source. OpenEdge installs the ODBC Driver Manager and associated ODBC Data Source Administrator if it was not already installed on your system.

Adding a new data source

The following procedure establishes a new data source for the Windows client.

To configure the ODBC client:

1. From the Windows Start menu, choose Settings > Control Panel > Administrative Tools > Data Sources (ODBC).

The ODBC Data Source Administrator dialog box appears:
2. Click **Add** to display a list of installed drivers.

The **Create New Data Source** dialog box appears:

3. Highlight the **Progress OpenEdge 10.2B** driver, then click **Finish**.

The **Progress OpenEdge Wire Protocol Driver Setup** dialog box appears.
4. Specify values for the following:

- **Data Source Name** — Identifies the data source configuration name. For example, Accounting.
- **Description** — An optional long description of the data source name. For example, My Accounting Database.
- **Data Source Name** — Identifies the data source configuration name. For example, Accounting.
- **Host Name** — The name of the system where the database or database broker is located.
- **Port Number** — The system port number setup for the database listener process.
- **Database Name** — The name of the database to which you want to connect by default.
- **User ID** — The default logon ID (user name) used to connect to your OpenEdge database. Your ODBC application can override this value or you can override it in the Logon dialog box or connection string.

5. From the **Progress OpenEdge Wire Protocol Driver Setup** dialog box, click the **Advanced** tab.

The **Advanced** tab dialog box appears:
6. Specify values for the following:

- **Default Isolation Level** — Specifies the default isolation level for concurrent transactions. Choose from the following values: READ COMMITTED, READ UNCOMMITTED, REPEATABLE READ, and SERIALIZABLE. **REPEATABLE READ** is the default.

- The **DefaultIsolationLevel** setting requires careful consideration. A transaction's isolation level will influence the type of record locks that are applied to records read from the database. The default value for this setting is **REPEATABLE READ**, unless you specify another setting during configuration. This will instruct the server to apply record **SHARE** locks, at a minimum, for the duration of a transaction to records read from the database. For most query tools, it is appropriate to set a default level of **READ COMMITTED**. The default isolation level setting can and will be overridden by applications.

- See Data Control Language and Transaction Behavior on page 145 for an explanation of the **SET TRANSACTION ISOLATION LEVEL command**.

- **Fetch Array Size** — The number of rows the driver retrieves when fetching from the server. This is not the number of rows given to the user. The default is 50.

- **Enable Timestamp with Time Zone** — Select this check box to expose timestamps with time zones to the application. By default, the check box is selected.

  When selected, the format is: **YYYY-MM-DD HH:MM:SS.SSS+HH:MM**.

  The equivalent connection string attribute is **EnableTimestampWithTimezone**.
• **Use Wide Character Types** — With this option enabled, character columns are defined as ODBC Unicode/wide types. For example, a VARCHAR column is defined as the SQL_WVARCHAR ODBC type. For compatibility reasons with OpenEdge Releases 10.X and earlier, character columns are defined as their basic types. For example, a VARCHAR column is defined as the SQL_VARCHAR ODBC type.

7. **Click Test Connect** to attempt to connect to the data source using the connection properties specified in the **Driver Setup** dialog box.

If the driver can connect, it releases the connection and displays a connection established message.

## Enabling encryption

OpenEdge supports data privacy and client/server authentication over connections between OpenEdge clients, servers, and middleware using the Secure Sockets Layer (SSL). You can use the following connection options to enable encryption for the ODBC driver. For more information on encryption, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.

To enable encryption:

1. From the ODBC Progress OpenEdge Wire Protocol Driver Setup window, click Security and the Security tab dialog appears:

![Security tab dialog box](image)

2. Specify values for the following:

   • **Encryption Method** — This options determines if the driver will encrypt and decrypt data sent to and from the database server. Select either No Encryption or SSL from the drop-down list.
• **Validate Server Certificate** — Select this option to enable the driver to validate the server certificate returned by the database server while establishing the SSL connection. If selected, the certificate returned must be issued by a certificate authority that is included in the trust store. Leaving this option unselected might be useful in test and debug environments because it eliminates the need to specify a trust store on all client machines in the test environment.

• **Truststore** — Certificate authorities trusted by the client reside in an encrypted file called a truststore. Enter the pathname that specifies the truststore file location.

• **Truststore Password** — Enter the trust store password to access the contents of the truststore.

• **Host Name In Certificate** — Ensures the driver is connecting to the requested server. Enter a name which the driver can examine against the value included in the certificate. If the name you enter and the value in the certificate do not match, then the connection fails. This feature is not enabled if **Validate Server Certificate** is not selected.

### Configuring UNIX clients

Before configuring UNIX clients for the ODBC driver, you must set your environment variables.

### Setting environment variables

The following table lists the environment variables you must set to configure the ODBC driver and the location where you must set them.
### Configuring data sources on a UNIX environment

UNIX platforms utilize a file called `odbc.ini` to configure ODBC settings for an installation. A sample version for OpenEdge is provided in the `$DLC/odbc` directory. For example:

```
[ODBC Data Sources]
sports=Progress_SQL92_Driver
[sports]
Driver=/usr/dlc/odbc/lib/pgoe1023.so
DatabaseName=default
PortNumber=2055
HostName=localhost
LogonID=testuser
Password=xxx
APILevel=1
ConnectFunctions=YYN
CPTimeout=60
DriverODBCVer=03.50
FileUsage=0
SQLLevel=0
UsageCount=1
ArraySize=50
DefaultLongDataBuffLen=2048
DefaultIsolationLevel=REPEATABLE READ
StaticCursorLongColBuffLen=4096
[ODBC]
InstallDir=/usr/dlc/odbc
Trace=0
TraceFile=odbctrace.out
TraceDll=/usr/dlc/odbc/lib/odbctrac.so
UseCursorLib=0
```

The first section, `[ODBC Data Sources]`, contains a list of data sources available for an ODBC client. In this case, the data source name (DSN) is `sports`.

---

**Table 4: ODBC driver environment variables**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Environment variable</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solaris, Linux</td>
<td>LD_LIBRARY_PATH</td>
<td>$DLC/odbc/lib</td>
</tr>
<tr>
<td>IBM AIX</td>
<td>LIBPATH</td>
<td>$DLC/odbc/lib</td>
</tr>
<tr>
<td>Hewlett Packard</td>
<td>SHLIB_PATH</td>
<td>$DLC/odbc/lib</td>
</tr>
</tbody>
</table>
The following section, [sports], contains data-source-specific information for the DSN sports. The odbc.ini file should contain a section like this for each DSN the client might wish to use. The name of the DatabaseName, PortNumber, and HostName should be replaced with the unique DataSource Name, the HostName serving the database, the PortNumber, and Database Name identified during the preparation steps. For example:

```
[sports]
Driver=/usr/dlc/odbc/lib/pgoe1023.so
DatabaseName=mysports
PortNumber=2055
HostName=myhost
LogonID=testuser
Password=xxx
APILevel=1
ConnectFunctions=YYN
CPTimeout=60
DriverODBCVer=03.50
FileUsage=0
SQLLevel=0
UsageCount=1
ArraySize=50
DefaultLongDataBuffLen=2048
DefaultIsolationLevel=REPEATABLE READ
StaticCursorLongColBuffLen=4096
```

Definitions of ODBC.INI tags

The following table provides definitions of the ODBC.ini tags.

Table 5: Definitions of ODBC.ini tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>The data-source-specific library used by the Driver Manager.</td>
</tr>
<tr>
<td>Database name</td>
<td>The name of the database used by the DSN.</td>
</tr>
<tr>
<td>(mandatory)</td>
<td></td>
</tr>
<tr>
<td>PortNumber</td>
<td>The Network Service number on which the database server is listening for</td>
</tr>
<tr>
<td>(mandatory)</td>
<td>connections.</td>
</tr>
<tr>
<td>HostName</td>
<td>The name of the host machine on which the database is running.</td>
</tr>
<tr>
<td>(mandatory)</td>
<td></td>
</tr>
<tr>
<td>LogonID</td>
<td>The default user ID for logon to the database server.</td>
</tr>
<tr>
<td>Password</td>
<td>The password for the default user ID.</td>
</tr>
<tr>
<td>APILevel</td>
<td>A number indicating the ODBC interface conformance level supported by the</td>
</tr>
<tr>
<td></td>
<td>driver:</td>
</tr>
<tr>
<td></td>
<td>• 0 — None</td>
</tr>
<tr>
<td></td>
<td>• 1 — Level 1 supported</td>
</tr>
<tr>
<td></td>
<td>• 2 — Level 2 supported</td>
</tr>
<tr>
<td>Tag</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ConnectFunctions</td>
<td>A three-character string indicating whether the driver supports SQLConnect, SQLDriverConnect, and SQLBrowseConnect.</td>
</tr>
<tr>
<td></td>
<td>• If the driver supports SQLConnect, the first character is &quot;Y&quot;; otherwise, it is &quot;N&quot;.</td>
</tr>
<tr>
<td></td>
<td>• If the driver supports SQLDriverConnect, the second character is &quot;Y&quot;; otherwise, it is &quot;N&quot;.</td>
</tr>
<tr>
<td></td>
<td>• If the driver supports SQLBrowseConnect, the third character is &quot;Y&quot;; otherwise, it is &quot;N&quot;.</td>
</tr>
<tr>
<td></td>
<td>• For example, if a driver supports SQLConnect and SQLDriverConnect but not SQLBrowseConnect, the three-character string is &quot;YYN&quot;.</td>
</tr>
<tr>
<td>CPTTimeout</td>
<td>The time interval setting for Connection Pooling Timeout. Not supported for UNIX.</td>
</tr>
<tr>
<td>DriverODBCVer</td>
<td>A character string with the version of ODBC that the driver supports. The version is of the form nn.nn, where the first two digits are the major version and the next two digits are the minor version.</td>
</tr>
<tr>
<td>FileUsage</td>
<td>A number indicating how a file-based driver directly treats files in a data source:</td>
</tr>
<tr>
<td></td>
<td>• 0 — The driver is not a file-based driver. For example, a driver may be a DBMS-based driver.</td>
</tr>
<tr>
<td></td>
<td>• 1 — A file-based driver treats files in a data source as tables. For example, an Xbase driver treats each Xbase file as a table.</td>
</tr>
<tr>
<td></td>
<td>• 2 — A file-based driver treats files in a data source as a catalog. For example, a Microsoft® Access driver treats each Microsoft Access file as a complete database.</td>
</tr>
<tr>
<td>SQLLevel</td>
<td>A number indicating the SQL-92 grammar supported by the driver:</td>
</tr>
<tr>
<td></td>
<td>• 0 — SQL-92 Entry</td>
</tr>
<tr>
<td></td>
<td>• 1 — FIPS127-2 Transitional</td>
</tr>
<tr>
<td></td>
<td>• 2 — SQL-92 Intermediate</td>
</tr>
<tr>
<td></td>
<td>• 3 — SQL-92 Full</td>
</tr>
<tr>
<td>UsageCount</td>
<td>Indicates driver libraries are in use. This value should not be modified.</td>
</tr>
<tr>
<td>Tag</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ArraySize</td>
<td>A hint that can be provided to the ODBC driver with regards to how many records can be retrieved from the server at a time.</td>
</tr>
<tr>
<td>DefaultLongDataBuffLen</td>
<td>Controls the size of an LOB that can be accessed in SQL.</td>
</tr>
<tr>
<td>DefaultIsolationLevel</td>
<td>The default isolation level under which data will be accessed.</td>
</tr>
<tr>
<td>StaticCursorLongColBuffLen</td>
<td>The default size for retrieving sections of LVARBINARY data.</td>
</tr>
</tbody>
</table>

The **DefaultLongDataBuffLen** parameter controls the size of an LOB that can be accessed in SQL. It defaults to 2048 (2mb) and must be set to match the largest possible LOB used by the application. The parameter is set via the registry on Windows and the odbc.ini file on UNIX and is specific to a DSN.

The **DefaultIsolationLevel** setting requires careful consideration. A transaction’s isolation level will influence the type of record locks that are applied to records read from the database. The default value for this setting is **REPEATABLE READ**, unless you specify another setting during configuration. This will instruct the server to apply record **SHARE** locks, at a minimum, for the duration of a transaction to records read from the database. For most query tools, it is appropriate to set a default level of **READ COMMITTED**. The default isolation level setting can and will be overridden by applications.

The **ArraySize** setting is a hint that can be provided to the ODBC driver with regards to how many records can be retrieved from the server at a time. The optimum setting will vary from application to application.

The location of the **odbc.ini** file must be determined once the correct modifications have been made and saved to the **odbc.ini** file. The driver will look, by default, to see if the **ODBCINI** environment variable exists. The driver expects that **ODBCINI** environment variable be exported into the environment to provide the full path and filename. For example:

```
ODBCINI=/usr/dlc/wrk/odbc.ini
```

Identifying the **odbc.ini** file in this manner provides a way to have one **INI** file in a shared location. This enables all ODBC clients to share a single **INI** file. The default location for the driver to look if the **ODBCINI** environment variable is not set is in the **$HOME** directory for a **.odbc.ini** file.

**Note:** The **.ini** file must be prefixed with a period.

---

**Testing your ODBC connection on UNIX**

This section contains a sample C program that can be used to test your ODBC connection. The sample program can be built at the UNIX OS prompt provided that:

- The OpenEdge SQL client is installed
- A C compiler is available on the system
- The source code is saved as **test_connect.c**
• The resulting executable is called testConnect

To build the executable, the following command will invoke the C compiler to compile and link the executable:

**Solaris and AIX**

```
cc -o testConnect -I$DLC/odbc/include -L$DLC/odbc/lib -lodbc test_connect.c
```

**HPUX**

```
cc -o testConnect -I$DLC/odbc/include -lc -L$DLC/odbc/lib -lodbc test_connect.c
```

**Linux**

```
cc -o testConnect -I$DLC/odbc/include -L$DLC/odbc/lib -lodbc -lodbcinst test_connect.c
```

Use the below sample program to test your ODBC connection.

testconnect.c

```c
#include <stdio.h>
#include "sql.h" /* ODBC declarations */
#include "sqlext.h" /* more ODBC declarations */
int main(int argc, char *argv[])
{
    SQLRETURN sqlReturn;
    SQLHANDLE environmentHandle;
    SQLHANDLE connectionHandle;
    /* make sure we got at least 3 arguments to the exe */
    if (argc < 4)
    {
        printf("Insufficient parameters provided.\n\n");
        printf("Usage - %s dsn userid password\n", argv[0]);
        return 1;
    }
    else
    {
        /* got at least 3 arguments to the exe, display */
        /* arguments with internal usage */
        printf("DSN NAME = %s\n", argv[1]);
        printf("USER ID = %s\n", argv[2]);
        printf("PASSWORD = *****\n"); /* don't show actual */
        /* allocate an ODBC environment handle */
        sqlReturn = SQLAllocHandle(SQL_HANDLE_ENV,
                                   SQL_NULL_HANDLE,
                                   &environmentHandle);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to allocate Environment Handle, exiting.\n\n");
            return 2;
        }
        /* allocate an ODBC connection handle */
        sqlReturn = SQLAllocHandle(SQL_HANDLE_DBC, 
                                    environmentHandle, 
                                    &connectionHandle);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to allocate Connection Handle, exiting.\n\n");
            return 3;
        }
        /* set the DSN */
        sqlReturn = SQLSetConnectOption(connectionHandle, SQL_OPTION_DSNCAT, 
                                         argv[1], 
                                         SQL_NTS_STRING);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to set DSN, exiting.\n\n");
            return 4;
        }
        /* set the user id */
        sqlReturn = SQLSetConnectOption(connectionHandle, SQL Option USER, 
                                         argv[2], 
                                         SQL_NTS_STRING);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to set User ID, exiting.\n\n");
            return 5;
        }
        /* set the password */
        sqlReturn = SQLSetConnectOption(connectionHandle, SQL Option PASSWORD, 
                                         argv[3], 
                                         SQL_NTS_STRING);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to set Password, exiting.\n\n");
            return 6;
        }
        /* execute a query */
        sqlReturn = SQLExecDirect(connectionHandle, "SELECT * FROM users", 
                                   SQL_NTS_CURSOR_ROWCOUNT);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to execute query, exiting.\n\n");
            return 7;
        }
        /* fetch the result */
        sqlReturn = SQLFetch(connectionHandle);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to fetch result, exiting.\n\n");
            return 8;
        }
        /* print the result */
        printf("Username\n\n");
        sqlReturn = SQLExectMultiple(connectionHandle, "SELECT * FROM users", 
                                      SQL_NTS_CURSOR_COUNT);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to execute query, exiting.\n\n");
            return 9;
        }
        /* close the connection */
        sqlReturn = SQLDisconnect(connectionHandle);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to disconnect, exiting.\n\n");
            return 10;
        }
        /* close the environment */
        sqlReturn = SQLFreeHandle(SQL_HANDLE_ENV, 
                                   environmentHandle);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to free environment handle, exiting.\n\n");
            return 11;
        }
        /* close the connection handle */
        sqlReturn = SQLFreeHandle(SQL_HANDLE_DBC, 
                                   connectionHandle);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to free connection handle, exiting.\n\n");
            return 12;
        }
        /* close the SQL environment */
        sqlReturn = SQLFreeHandle(SQL_HANDLE_ENV, 
                                   environmentHandle);
        if (sqlReturn != SQL_SUCCESS)
        {
            printf("Unable to free SQL environment, exiting.\n\n");
            return 13;
        }
    }
    return 0;
}
```

return sqlReturn;
}
/* set the ODBC application version to 3.x */
sqlReturn = SQLSetEnvAttr(environmentHandle,
SQL_ATTR_ODBC_VERSION,
(SQLPOINTER) SQL_OV_ODBC3,
SQL_IS_UINTEGER);
if (sqlReturn != SQL_SUCCESS)
{
    printf("Unable to set ODBC Versoin to 3.x, exiting.\n");
    SQLFreeHandle(SQL_HANDLE_ENV, environmentHandle);
    return sqlReturn;
}
/* allocate a database connection handle */
sqlReturn = SQLAllocHandle(SQL_HANDLE_DBC,
environmentHandle,
&connectionHandle);
if (sqlReturn != SQL_SUCCESS)
{
    printf("Unable to allocate Connection Handle, exiting.\n");
    SQLFreeHandle(SQL_HANDLE_ENV, environmentHandle);
    return sqlReturn;
}
/* attempt to connect o the database server */
sqlReturn = SQLConnect(connectionHandle,
(SQLCHAR *)argv[1], /* dsn name */
SQL_NTS, /* name null terminated */
(SQLCHAR *)argv[2], /* user id */
SQL_NTS, /* user id null terminated */
(SQLCHAR *)argv[3], /* user pwd */
SQL_NTS); /* pwd null terminated */
if (sqlReturn == SQL_SUCCESS)
{
    printf("Connection to %s successful!\n", argv[1]);
    /* now disconnect from server */
    sqlReturn = SQLDisconnect(connectionHandle);
    if (sqlReturn != SQL_SUCCESS)
    {
        printf("Unable to disconnect client from %s.\n", argv[1]);
    }
    else
    {
        printf("Unable to connect to %s.\n", argv[1]);
    }
    /* do the clean up before exiting */
    if (connectionHandle != NULL)
    {
        SQLFreeHandle(SQL_HANDLE_DBC, connectionHandle);
    }
    if (environmentHandle != NULL)
    {
        SQLFreeHandle(SQL_HANDLE_ENV, environmentHandle);
    }
    return sqlReturn;
}

The test executable, once built, can be used to test the ability to connect to a running database
server. The executable will take three parameters: DSN, user id, and user password, as shown:

testConnect sports myuser mypwd
A successful connection, using the test executable, will result in a displayed message similar to the following:

```
DSN NAME = sports
USER ID = foo
PASSWORD = ****
Connection to sports successful.
```
The OpenEdge SQL Data Control Language (DCL) provides security for your database. The DCL consists of the `GRANT`, `REVOKE`, `COMMIT`, and `ROLLBACK` statements. `GRANT` and `REVOKE` statements enable you to determine whether a user can view, modify, add, or delete database information.

This chapter covers the `GRANT` and `REVOKE` statements. `COMMIT` and `ROLLBACK` are covered in Data Control Language and Transaction Behavior on page 145.

**Note:** This chapter gives an overview of DCL statements. For complete syntax of each statement, see *OpenEdge Data Management: SQL Reference*.

For details, see the following topics:

- Working with database security
- Creating users
- Granting privileges
- Verifying privileges
- Revoking privileges

**Working with database security**

For a more comprehensive overview of database security in OpenEdge, including a comparison between SQL and ABL access, see *OpenEdge Getting Started: Identity Management*.
Comparing OpenEdge SQL and ABL security

Both OpenEdge SQL and ABL can be used to develop security measures for your database. If users can access the database with either language, you must then use both languages to enforce rules governing database security.

OpenEdge SQL security

OpenEdge SQL is a closed system and always requires a user identification and password to allow database access. When the SQL DBA assigns a user ID and password, the user has access to the database, but is unable to access or manipulate data until the DBA specifically grants privileges.

The DBA uses SQL to grant different types of privileges. The DBA may choose to grant all users unlimited capabilities. However, privileges are usually restricted, allowing users the ability to retrieve, update, or delete only that data needed to perform their assigned jobs.

Privileges assigned by the SQL DBA are enforced only when the user accesses the database through a SQL client. If a user has the ability to access the database through either SQL or ABL clients, the DBA should ensure the user has been granted identical SQL privileges and ABL permissions.

ABL security

An ABL database is an open system. ABL clients encounter no security restrictions when accessing a newly created OpenEdge database. However, a database administrator can begin to restrict access by using the Data Administration tool, OpenEdge Management, or OpenEdge Explorer to add certain users as security administrators. A security administrator can limit access to the database’s tables, fields, and _User table accounts. A database administrator can create these users in the database _User table accounts or in any other user account systems that OpenEdge supports.

Note that OpenEdge-supported user accounts can, with appropriate permissions, access an OpenEdge database through either the ABL or SQL security systems; however, ABL user accounts with a SQL only designation prevent ABL clients from using the account to access the database. The mechanisms that each security system uses to restrict access to the database by these user accounts differs.

ABL (unlike SQL) also has the concept of a default user that can connect to and access any OpenEdge database simply by not using the connection options to specify a user ID and password. This default user has a blank ("") user name in no particular domain, and has no real physical account defined anywhere. Without additional security measures, this default user has unfettered access to the database. By defining specific, non-blank users as Security Administrators, this prevents the default user, itself, from acting as a Security Administrator.

The Security Administrator can further enhance ABL security by assigning user permissions (including permissions on the default user) to define actions that may be taken on specific database objects. Again, ABL permissions can be assigned through the Data Administration tool, OpenEdge Management, or OpenEdge Explorer. For information on using the Data Administration tool, see OpenEdge Data Management: Database Administration. For information on using OpenEdge Explorer and OpenEdge Management, see OpenEdge Management and OpenEdge Explorer: Getting Started. From this point forward in this document, Database Administration refers to all three of these tools.
Any security permissions that a Security Administrator defines for users through Database Administration apply only to users accessing the database through an ABL client. (A SQL DBA cannot administer ABL security.) If users are able to access a database through both SQL and ABL clients, then DBAs should ensure that equivalent security models are developed for both clients.

Comparing authentication and authorization

In order to properly implement security measures, it is important to understand the difference between authentication and authorization.

Authentication

Authentication is the process of providing your user ID and password to the server in order to connect to the database from a client. The server then verifies your user identification. The client might be either an ABL client or an OpenEdge SQL (JDBC or ODBC) client.

For both OpenEdge SQL and ABL, support for users depends on the definition of OpenEdge security domains configured for supported authentication systems. A security domain defines the common mechanisms that a given security system (or systems) relies on to provide authentication, authorization, and session management services. A DBA defines these domains in each OpenEdge database using Database Administration and configures each domain for a particular authentication system. An authentication system determines the mechanism by which users are authenticated for access to the resources (e.g., databases) controlled by a given security system (such as OpenEdge SQL or ABL). By configuring an OpenEdge domain with a given authentication system, the DBA allows all the user accounts controlled by that authentication system to authenticate to the OpenEdge database in which the domain is defined (or in which the domain is defined to be a trusted authentication domain).

For both OpenEdge SQL and ABL, a user identifies the authentication system that OpenEdge uses to authenticate their user ID and password by specifying the name of an appropriate OpenEdge domain as part of their user ID. Such a user ID is known as a fully qualified user ID, because it specifies both their user name and domain name separated in a single string by the '@' delimiter character. Depending on the database configuration, non-qualified user IDs (the user name, alone, with no specified domain) can be used. In this case, OpenEdge uses the authentication system configured for the blank ('') domain to authenticate non-qualified user IDs.

**Note:** Only multi-tenant databases require and assume the use of a qualified user ID for authentication. For non-multi-tenant databases, while fully qualified user IDs can be defined and used, non-qualified user IDs can also be used for authentication.

OpenEdge supports a particular built-in set of authentication systems. Of these built-in authentication systems, two can be used to authenticate both SQL and ABL client connections to a database server:

- _oeusertable — Authenticates users defined in the database _User table accounts.
- _oslocal — Authenticates users defined in the server operating system accounts. Currently these include accounts created in the native Windows or Unix account systems.

For more information on OpenEdge support for security domains, authentication systems, and user IDs for authentication, see OpenEdge Getting Started: Identity Management.
From OpenEdge SQL, authentication can be enabled through _oeusertable domains when a DBA uses the `CREATE` statement to create a user and assign a password, thus creating an entry in the database's _User table accounts. ABL DBAs can create entries in the same _User table accounts by assigning user IDs and passwords through Database Administration. In addition, users defined in _User table accounts can be designated as SQL-only accounts, which can only authenticate users for access through OpenEdge SQL. For both OpenEdge SQL and ABL, administrators can enable authentication through _oslocal domains by creating user IDs and passwords for the user accounts of the Windows or Unix system that hosts the database server.

Although OpenEdge SQL and ABL security models are independent of each other, they do share common authentication systems. User IDs and passwords created for ABL in the supported authentication systems are also recognized by OpenEdge SQL, unless marked for SQL use only. Likewise, user IDs and passwords created using OpenEdge SQL are also recognized by ABL. In OpenEdge SQL, authentication occurs at the server; in ABL, authentication occurs at the ABL client.

**Note:** When initially populating the _User table from the SQL side, ensure that the first table entry is that of the DBA and ensure that particular username is granted DBA and RESOURCE privileges in the same transaction. Similarly, before populating the _User table from the ABL side, ensure that a database administrator defines at least one non-blank user ID as a Security Administrator. Otherwise, you can lock yourself out of the database.

**Authorization**

Authorization is the process by which DBAs (or Security Administrators for ABL) enable users to perform tasks on the database, such as retrieving, updating and deleting information. OpenEdge SQL and ABL each employ their own mechanism for performing authorization.

OpenEdge SQL performs authorization through the `GRANT` statement. ABL database administrators (initially) and Security Administrators (primarily) authorize users to perform database tasks using Database Administration. In addition, ABL programmers can use the same permissions mechanism to authorize access to application functions.

For more information on defining and using authorization permissions through Database Administration and ABL, see *OpenEdge Getting Started: Identity Management*, *OpenEdge Data Management: Database Administration*, and *OpenEdge Management and OpenEdge Explorer: Getting Started*.

**Note:** OpenEdge supports authorization based on a fully qualified user ID only for access to multi-tenant databases. For non-multi-tenant databases, even if you authenticate access to a database connection using a fully-qualified user ID, OpenEdge authorizes access based only on the non-qualified user ID (user name).

**Creating users**

The `CREATE` statement is not a part of the Data Control Language, but rather the Data Definition Language. This chapter addresses the `CREATE` statement as it relates to the creation of database administrators and users. For more information on the `CREATE` statement, see *OpenEdge SQL Data Definition Language* on page 67.
Creating database administrators

Database security is defined and controlled by database administrators (DBAs). Within the scope of database security, DBAs are responsible for:

- Adding users
- Deleting users
- Permitting access to specific database objects
- Limiting or prohibiting access to database objects
- Granting users privileges to view or modify database objects
- Modifying or revoking privileges that have been granted to the users

A user who initially creates a database becomes its default administrator. Therefore, this initial user has the authority to create other administrator accounts for that particular database.

OpenEdge Studio offers two methods for creating DBAs:

- In SQL, the DBA can use the `CREATE` statement to create a user in the `_User` table and then use the `GRANT` statement to provide the user with administrative privileges.
- In ABL, a DBA uses Database Administration to define other administrators and Security Administrators.

Creating users in the `_User` table

Using the `CREATE USER` statement, you can only create users in the `_User` table. Users defined in other user accounts must be created outside of SQL. Use the following syntax to employ the `CREATE USER` statement:

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

Examples: `CREATE USER` statement

In the following example, an account with DBA privileges creates the 'username' 'GPS' with 'password' 'star'.

```
CREATE USER 'GPS', 'star';
```

In the following example, `CREATE USER` creates the user Jasper to connect to mtdomain domain with the password spaniel:

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

The user Jasper should be associated with the tenant identified by the domain mtdomain.
A user’s password can be changed easily by using the ALTER USER statement, as shown:

```
ALTER USER { 'username' | 'username@domain_name' }, 'old_password', 'new_password';
```

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

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

**Examples: ALTER USER statement**

The following example demonstrates the use of the ALTER USER statement:

```
ALTER USER 'GPS', 'star', 'star1';
```

The following example demonstrates 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';
```

When users are created, the default DBA (the user who created the database) becomes disabled. It is important to grant DBA privileges to at least one user so you will have a valid DBA account.

For complete details on the CREATE USER statement, see *OpenEdge Data Management: SQL Reference*.

**SQL Only Users**

As the OpenEdge SQL databases can be multi-tenant enabled, SQL access to a multi-tenant database is made more secure by enabling the SQL DBA to define an improved means of authenticating the SQL users who log into the OpenEdge SQL server.

This section describes the extended support to users of SQL only in the existing CREATE USER and ALTER USER statements:

- Create an SQL only user
- Alter a user to an SQL only user
Create an SQL only user

Allows DBAs to create user definitions—in the database _User table—that are only visible to and used by the OpenEdge SQL, and are transparent to OpenEdge Advanced Business Language (ABL).

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

Examples: Creating an SQL ONLY user

The following example demonstrates how a user with DBA privileges creates an SQL only user Jasper with the password spaniel:

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

The following example demonstrates how a user with DBA privileges on a multi-tenant table, creates an SQL only user Jasper to connect to mtdomain domain with the password spaniel:

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

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

For complete details on the CREATE USER statement, see OpenEdge Data Management: SQL Reference.

Alter a user to an SQL only user

Allows DBAs to alter user definitions—in the database _User table—such that the user is only visible to and used by the OpenEdge SQL, and is transparent to the OpenEdge ABL.

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

Examples: Altering an SQL ONLY user

The following example demonstrates how the ALTER USER statement changes Jasper to an SQL only user:

```
ALTER USER 'Jasper'
FOR SQL ONLY;
```
The following example demonstrates how the `ALTER USER` statement changes the SQL only user `Jasper` to an SQL and an ABL visible user:

```
ALTER USER 'Jasper@mtdomain'
NOT FOR SQL ONLY;
```

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

For complete details on the `ALTER USER` statement, see *OpenEdge Data Management: SQL Reference*.

---

**Granting privileges**

**Privilege basics**

There are two types of privileges—those granted on databases and those granted on tables, views, and procedures.

Privileges for databases are:

- Granting or restricting system administration privileges (`DBA`)
- Granting audit privileges (audit administration, audit archive, or audit application insert)
- Granting or restricting general creation privileges on a database (`RESOURCE`)

Privileges granted on tables, views, and procedures grant or restrict operations on specific operations, such as:

- Altering an object definition
-Deleting, inserting, selecting and updating records
- Executing stored procedures
- Granting privileges
- Defining constraints to an existing table

**GRANT statement**

The `GRANT` statement can be used to provide the user with database-wide or table-specific privileges. It can also grant public access to restricted columns.
Database-wide privileges

Database-wide privileges grant the user **DBA, RESOURCE, AUDIT_ADMIN, AUDIT_ARCHIVE, or AUDIT_INSERT** privileges. Users with **DBA** privileges have the ability to access, modify, or delete a database object and to grant privileges to other users. **RESOURCE** privileges allow a user to create database objects. Users with **AUDIT_ADMIN** privileges can read the data of audit-enabled databases. Users with **AUDIT_ARCHIVE** privileges can read, archive, and delete audit data. Users with **AUDIT_INSERT** privileges can insert application audit events into audit tables. For general information about implementing auditing, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.

Syntax

The **GRANT** statement syntax for granting **RESOURCE** or **DBA** privileges is:

```
GRANT { RESOURCE, DBA, AUDIT_ADMIN, AUDIT_ARCHIVE, AUDIT_INSERT }
    TO user_identifier[, user_identifier], ...
    [ WITH GRANT OPTION ];
```

**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.

**username | username@domain_name**

**Note:** By employing the **WITH GRANT OPTION** clause, you enable a user to grant the same privilege he or she has been granted to others. This clause should be used carefully due to its ability to affect database security. Example

**Examples: GRANT RESOURCE statement**

The following example demonstrates the use of the **GRANT RESOURCE** statement.

```
GRANT RESOURCE TO 'GSP';
```

In this case, **GSP** is granted the privilege to issue **CREATE** statements, and can therefore add objects, such as tables, to the database.

The following example demonstrates how 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.
Table-specific privileges

Table-specific privileges can be granted to users so they can view, add, delete, or create indexes for data within a table. Privileges can also be granted to allow users to refer to a table from another table’s constraint definitions.

Syntax

The `GRANT` statement syntax for granting table-specific privileges is:

```
GRANT {privilege [, privilege], . . . | ALL } ON table_name
TO {username [, username], . . . | PUBLIC } [ WITH GRANT OPTION ];
```

This is the syntax for the `privilege` value:

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

In this instance, a DBA restricts the types of activities a user is allowed to perform on a table. In the following example, 'GSP' is given permission to update the item name, item number, and catalog descriptions found in the item table.

**Note**: By employing the `WITH GRANT OPTION` clause, you enable a user to grant the same privilege he or she has been granted to others. This clause should be used carefully due to its ability to affect database security.

Examples: GRANT UPDATE statement

The following example illustrates the granting of table-specific privileges.

The `GRANT UPDATE` statement has limited GSP’s ability to interact with the item table.

```
GRANT UPDATE
ON Item (ItemNum, ItemName, CatDescription)
TO 'GSP';
```

Now, if GSP attempts to update a column to which he has not been granted access, the database will return the error message in the following example.

```sql
== SQL Exception 1 ==
SQLState=HY000
ErrorCode=-20228
[JDBC Progress Driver]:Access Denied (Authorisation failed) (7512)
```
Granting public access

Example: Granting update privilege to public

The `GRANT` statement can be easily modified to make previously restricted columns accessible to the public, as in the following example.

```
GRANT UPDATE
ON Item (ItemNum, ItemName, CatDescription)
TO PUBLIC;
```

For detailed information on the `GRANT` statement, see *OpenEdge Data Management: SQL Reference*.

Verifying privileges

Privileges granted to users are specified in the following database system tables:

- **SYSPROGRESS.SYSCOLAUTH** — This system table identifies users and the columns for which they have privileges. It also identifies the privileges the user has been granted, and the person who granted those privileges.

- **SYSPROGRESS.SYSDBAUTH** — This system table identifies users who have authority to connect to a database, create tables, and perform administrative tasks.

- **SYSPROGRESS.SYSTABAUTH** — This system table identifies users and the tables for which they have privileges. It also identifies the privileges the user has been granted, and the person who granted those privileges.

 Revoking privileges

The `REVOKE` statement can be used for a wide variety of purposes. It can revoke a single user's access to a single column or it can revoke the public's privilege to access an entire database.

Privileges are revoked in the same manner in which they are granted—database-wide or table-specific.

Syntax

The syntax for using the `REVOKE` statement to revoke database-wide privileges is:

```
REVOKE 
FROM 
[ RESTRICT ] [ CASCADE ] [ GRANTED BY ANY_USER ];
```
Identifies a username. For a tenant user, you must mention the fully qualified user name, 
username@domain_name, to revoke access from a user.

```
username | username@domain_name
```

**Note:** Only the user who granted an audit-related privilege can revoke it. **CASCADE** is not supported for the revocation of audit-related privileges.

The syntax for using the `REVOKE` statement to revoke table-specific privileges is:

```
REVOKE [ GRANT OPTION FOR ] {privilege[, privilege], ...}
  | ALL [ PRIVILEGES ] } ON table_name
FROM {username[,username], ...}
| PUBLIC } [RESTRICT | CASCADE];
```

where privilege is:

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

**Examples: REVOKE statement**

The `REVOKE` statement can be used to remit the privileges previously granted to 'GPS', as shown in the following example.

```
REVOKE UPDATE
ON Item (ItemNum, ItemName, CatDescription)
FROM "GPS";
```

The following example demonstrates how 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;
```

In the above example, the user names `dbuser1` and `dbuser2` must be associated with the domain name `domuser1` in the multi-tenant table.

If the `REVOKE` statement specifies **RESTRICT**, SQL checks to see if the privilege being revoked was passed on to other users. This is possible only if the original privilege included the **WITH GRANT OPTION** clause. If so, 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 specifies neither **RESTRICT** nor **CASCADE**, the behavior is the same as for **CASCADE**.
For detailed information on the `REVOKE` statement, see *OpenEdge Data Management: SQL Reference*. 
OpenEdge SQL Data Definition Language

This chapter provides information on the Data Definition Language (DDL), which consists of the CREATE, ALTER, and DROP statements. The DDL is used to create and manage database objects, as described in the following sections.

Note: This chapter gives an overview of DDL statements. For complete syntax of each statement, see OpenEdge Data Management: SQL Reference.

For details, see the following topics:

• Using Data Definition Language statements
• Maintaining data integrity
• Working with SQL utilities

Using Data Definition Language statements

DDL statements are used to add, modify, or remove database objects. The DDL is integral to creating and maintaining a database. For detailed information on database creation and administration, see OpenEdge Data Management: Database Administration.
Note: When creating a table, view, index, or group in the SQL engine, choose table, view, index, group, and column names that are not SQL keywords. This is enforced by the SQL engine. However, if you are creating objects in the PUB schema using the SQL engine, you also must avoid using reserved keywords, such as DISPLAY. This is not enforced by the SQL engine. For a complete list of keywords, see OpenEdge Data Management: SQL Reference.

You can use DDL statements to manage encryption of database objects. For details, see OpenEdge Getting Started: Core Business Services - Security and Auditing.

Working with tables

The following sections provide details on creating, modifying, and deleting tables.

CREATE TABLE

The CREATE TABLE statement allows you to create a new table in an existing database by defining its column names and column data types. Optionally, you can include table and column constraints.

The CREATE TABLE syntax forms are explained below:

- The first syntax form explicitly specifies the definition of a column.
- The second syntax form, with the AS query_expression clause, implicitly defines the columns using the columns in a query expression.
- The third syntax form defines the new table as a multi-tenant table, and allocates storage area of a database to the tenants.
- The fourth syntax form specifies partition key details along with area definitions.

Note: The CREATE TABLE statement used for partitioned tables is an online operation. For detailed information on using the CREATE TABLE statement for partitioned tables, see OpenEdge Data Management: SQL Reference.

Syntax

The CREATE TABLE statement uses the following syntax:

```
CREATE TABLE [owner_name] table_name (  
{column_definition|table_constraint}, ... )  
[ AREA area_name ]  
[ ENCRYPT WITH cipher ]  
[ BUFFER_POOL { PRIMARY | ALTERNATE } ]  
[ progress_table_attribute_keyword value ] ;  
CREATE TABLE [owner_name] table_name [ {column_name [ NOT NULL ] , ... } ]  
[ AREA area_name ] [ ENCRYPT WITH cipher ]  
[ BUFFER_POOL { PRIMARY | ALTERNATE } ]  
AS query_expression ;  
CREATE TABLE
```
The following syntax is used to define an LOB column in `CREATE TABLE` statement:

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

For details on using the `CREATE TABLE` statement to designate objects for buffer pool assignments, including an alternate buffer pool, see *OpenEdge Data Management: Database Administration*.

For details on using the `CREATE TABLE` statement to enable transparent data encryption, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.

**Example: CREATE TABLE statement**

The following example illustrates a `CREATE TABLE` statement. The `cust_no` column has the column constraint `NOT NULL`, which indicates that no row in the customer table is to have a `NULL` value in the `cust_no` column.

```
CREATE TABLE SPORTS.Customer
(
  cust_no INTEGER NOT NULL,
  last_name CHAR (30),
  street CHAR (30),
  city CHAR (20),
  state CHAR (2)
) ;
```

The `CREATE TABLE` statement also allows you to specify the `DEFAULT` clause along with a column definition. The `DEFAULT` clause identifies the default value to be used for a column.
The default clause uses the following syntax:

```
column_name data_type
  [ DEFAULT { literal | NULL | SYSDATE | SYSTIME | SYSTIMESTAMP } ]
  [ column_constraint [ column_constraint , . . . ] ]
```

Example: CREATE TABLE statement with DEFAULT clause

The following `CREATE TABLE` statement shows how to use the `DEFAULT` clause. The following example sets a default value of 10 for the `deptno` column.

```
CREATE TABLE employee
  ( empno  INTEGER NOT NULL,
    deptno INTEGER DEFAULT 10
  ) ;
```

For more information on the `CREATE TABLE` statement and the `DEFAULT` clause, see *OpenEdge Data Management: SQL Reference*.

Example: Creating a multi-tenant table

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 ;
```

Example: Creating a multi-tenant table with default areas

The following example shows how to create a 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;
```

Example: Defining areas for a 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";
```

**Example: Creating a partitioned table**

The following example illustrates how to partition 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
  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";
```

**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)
    PARTITION USA_MA_BT VALUES IN ('USA','MA','Boston'),
    PARTITION USA_NY_NY VALUES IN ('USA','NY','Ney York'),
    PARTITION USA_MA_BD VALUES IN ('USA','MA','Bedford')
  USING INDEX AREA "secunderabad index area"
```

**Example: Creating a table with LOB partitions**
The following example illustrates how to create a table with LOB partitions:

```sql
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 Pub.tpcustomer_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.

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

For more information on the CREATE TABLE statement, see OpenEdge Data Management: SQL Reference.

**ALTER TABLE**

The ALTER TABLE statement lets you:

- Change the name of a table
- Change the name of a column within a table
- Add a column to a table
- 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

**Note:** For a complete description of the ALTER TABLE syntax, see OpenEdge Data Management: SQL Reference.
Syntax

```sql
ALTER TABLE [owner_name]table_name{ADD column-definition
  |SET progress_table_attribute value | SET { ENCRYPT WITH cipher
  | DECRIPT
  | ENCRYPT REKEY ]
 | BUFFER_POOL{ PRIMARY | ALTERNATE } }
| ALTER [ COLUMN ]column_name
  { SET DEFAULT value
    | DROP DEFAULT
    | SET [ NOT ] NULL
    | SET progress_column_attribute value}
 | SET ENCRYPT WITH cipher
 | SET DECRYPT
 | SET ENCRYPT REKEY
 | SET BUFFER_POOL{ PRIMARY | ALTERNATE } }
 | DROP COLUMN column_name
  { CASCADE | RESTRICT }
 | ADD [CONSTRAINT constraint_name]
  {primary_key_definition
    |foreign_key_definition
    |uniqueness_constraint
    |check_constraint}
  | AREA area_name]
 | DROP CONSTRAINT constraint_name[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 }
```
The addition or deletion of columns is a common modification for tables. When a column is added, the OpenEdge RDBMS places the column to the far right of the table. Unless you declare the column to be NOT NULL and assign a default value, the RDBMS will assume the column has a value of NULL for each row in the existing table.

For details on using the `ALTER TABLE ADD COLUMN` statement to designate objects for buffer pool assignments, including an alternate buffer pool, see *OpenEdge Data Management: Database Administration*.

For details on using the `ALTER TABLE` statement to enable transparent data encryption, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.

**Example: ALTER TABLE statement**

The following example shows how the `ALTER TABLE` statement is used to add a column to a table.

```
ALTER TABLE SPORTS.Customer
ADD Customer_phone CHAR (10);
```

The `ALTER TABLE` statement also can be used to change the name of an existing table. To do so, SQL uses the following syntax:

```
ALTER TABLE [owner_name.]table_name RENAME TO [owner_name.]table_name;
```

**Example: Using ALTER TABLE statement to rename table**

The following example shows how the `ALTER TABLE` statement is used to rename an existing table.

```
ALTER TABLE SPORTS.employee RENAME TO SPORTS.staff;
```

**Example: Altering a multi-tenant table**

The following example shows how to change the `pub.mtcustomer` table to a multi-tenant table with default space allocations:

```
ALTER TABLE pub.mtcustomer SET MULTI_TENANT;
```
Example: Altering a multi-tenant table with overriding space allocations

The following example shows how to alter 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.

Syntax for repairing 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`).

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

Example: Repairing CRC mismatch error

The following example shows how to repair a 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.

Example: Using ALTER TABLE statement for partitioned tables

The following example shows how to use `ALTER TABLE` statement for partitioned tables:

```
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 above statement creates a table with two partition aligned indexes: `IDX1` and `IDX2` and an index `IDX3` that is not partition aligned.
The below ALTER TABLE statement converts the above table to a partitioned table by adding one logical data partition 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.

```sql
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;
```

For more information, see *OpenEdge Data Management: SQL Reference*.

**DROP TABLE**

The DROP TABLE statement deletes all data and indexes for a table and erases its entry in the system catalog.

If a multi-tenant table is dropped, then, the DROP TABLE deletes the table partition for every tenant defined in the table.

You can also use the DROP TABLE statement to drop a partitioned table. When a partitioned table is dropped, all the table, index, and LOB partitions of the table are deleted.

**Syntax**

The DROP TABLE statement uses the following syntax:

```
DROP TABLE [owner_name.]table_name;
```

**Example: DROP TABLE statement**

The following example illustrates the use of a DROP TABLE statement.

```
DROP TABLE SPORTS.staff;
```

**Working with indexes**

An index is a database object used to speed the time in which data is retrieved from a table. Although OpenEdge creates an index for any column to which you assign a unique constraint, you can also explicitly create an index.

**CREATE INDEX**

The CREATE INDEX statement creates an index on one or more columns of a table. You can specify an index in ascending order (ASC) or descending order (DESC).

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.
**Note:** For a complete description of the **CREATE INDEX** statement, see *OpenEdge Data Management: SQL Reference*.

**Syntax**

The **CREATE INDEX** statement uses the following 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 ] ... 
[ FOR PARTITION partition_name USING INDEX AREA area_name ] ... ;
```

**Syntax to create a global index**

The following syntax is used to create a global index:

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

**Example: CREATE INDEX statement**

The index in this **CREATE INDEX** example is specified on the single column `empno`, and is of ascending order on the value of the column, as shown in the following example.

```
CREATE INDEX idx_emp ON SPORTS.employee (empno ASC) ;
```

**Example: Creating a multi-tenant index**

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

```
CREATE PRO WORD INDEX CommentsWordIdx ON mtcustomer (cust_num) 
  MULTI_TENANT 
  FOR TENANT DEFAULT USING INDEX AREA "Misc_Index_Area"
;
```

**Example: Creating local indexes for a partitioned table**
The following example shows how to create local indexes for the partitioned table `Pub.tpcustomer` with partitions:

```sql
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"
```

Example: Creating a global index for a partitioned table

The following example shows how to create local indexes for the partitioned table `Pub.tpcustomer` with partitions:

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

For details on using the `CREATE INDEX` statement to enable transparent data encryption, see OpenEdge Getting Started: Core Business Services - Security and Auditing.

For details on using the `CREATE INDEX` statement to designate objects for buffer pool assignments, including an alternate buffer pool, see OpenEdge Data Management: Database Administration.

**DROP INDEX**

Use the `DROP INDEX` statement to drop a table index. An index can only be dropped from tables with more than one index. The initial index of a table cannot be dropped.

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.

**Examples: DROP INDEX statement**

The following example demonstrates the use of a `DROP INDEX` statement.

```sql
DROP INDEX idx_cust ON customer;
```

The following example drops the local index `custnum` including all its partitions on the table `Pub.tpcustomer`.

```sql
DROP INDEX custnum ON Pub.tpcustomer;
```

The following example drops the global index `custname`, which has only one physical partition and that physical partition is dropped.

```sql
DROP INDEX custname ON Pub.tpcustomer;
```
Enabling large database index keys

As of OpenEdge Release 10.1B, large index keys are supported for databases with 4,096 and 8,192 byte block sizes. The large index keys may be as big as 2,000 bytes. In releases 10.1B and later, this feature is enabled by default. In Releases 10.1A and earlier, you can enable this feature by using the following online statement:

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

You must have SQL DBA privileges to enable large keys. If large keys are already enabled, an error message indicating such will be returned when you use the enabling statement. The database blocksize must be 4,000 or 8,000 bytes for large keys to be enabled.

For more information on the use of indexes, see OpenEdge SQL Data Manipulation Language on page 113

Working with views

A view is a virtual table created by a query. The query can retrieve data from specific columns in one or more tables. Views are created when users need to routinely query one or more tables for the same information.

**CREATE VIEW**

Use the `CREATE VIEW` statement to create a view on existing tables or views. You specify the name for the `VIEW`.

The following example demonstrates the use of a code sample. The statement is used to retrieve information from the customer base, but only information on those customers located within the six New England states.

```
CREATE VIEW ne_customer AS
    SELECT cust_no, last_name, street, city, state
    FROM SPORTS.customer
    WHERE state in ('NH', 'MA', 'ME', 'CT', 'RI', 'VT') ;
```

**DROP VIEW**

The statement needed to drop a view is a short and simple one, as shown in the following example.

```
DROP VIEW ne_customer;
```
Working with sequences

Sequences are database objects that automatically generate numbers in a sequential order. Usually, sequences are used to provide a table with a series of unique numbers such as primary key values. 32-bit sequences enable you to generate unique sets of INTEGER values and 64-bit sequences enable you to generate unique sets of BIGINT values.

A sequence defined as a terminating sequence provides a guarantee of order and uniqueness. A sequence defined to cycle at a limit provides applications with a control value that can be combined with another value to provide a unique value. Sequences provide you with an alternative to a control table for the generation of values. Sequences offer less contention in reading and generating values and therefore perform better than control tables.

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

Syntax

The CREATE SEQUENCE statement uses the following syntax:

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

Examples: CREATE SEQUENCE statement

In the following example, a sequence is used to generate unique customer numbers when a new row is inserted into the table `pub.customer`.

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

Example: Creating 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;
```

Sequences can be:
• Sequential values within any range of an OpenEdge **INTEGER** or **BIGINT** datatype
• Incremented or decremented
• Configured with an initial value
• Configured to terminate at a specific limit
• Configured to cycle at a specific limit

A sequence number is generated independently of the transaction which is committing or rolling back. It is possible that individual sequence numbers will appear to be skipped because they were generated and used in a transaction that ultimately rolled back. Sequence numbers are generated independently of tables so they can be used for more than one table.

For more on the **CREATE SEQUENCE** statement, see *OpenEdge Data Management: SQL Reference*.

### Enabling 64-bit sequences

You can enable 64-bit sequences in a database by using the following online statement:

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

OpenEdge databases support sequences with 64-bit maximums for positive and negative numbers. In releases 10.1B and later, 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. You must have SQL DBA privileges to use this command.

### Using CURRVAL and NEXTVAL in a statement

When you create a sequence, you can define its initial value and the increment between its values. The first reference to **NEXTVAL** returns the sequence's initial value. Subsequent references to **NEXTVAL** increment to the sequence value by the defined increment and return the new value. Any reference to **CURRVAL** always returns the sequence's current value, which is the value returned by the last reference to **NEXTVAL**. Therefore, a statement that contains multiple references to `sequence_name.NEXTVAL` will return the same value for each reference of `sequence_name.NEXTVAL`.

#### Example: INSERT statement using NEXTVAL

The following example uses **NEXTVAL** to assign a new customer number into a customer table.

```
INSERT INTO pub.customer (custnum, firstname, lastname, address)
VALUES (pub.customer_sequence.NEXTVAL, 'Jane', 'Smith', '1 Maple St.');
```

#### Example: SELECT statement using CURRVAL

To find the value entered in the customer table of the preceding example, use the following statement:

```
SELECT pub.customer_sequence.CURRVAL from pub.customer;
```
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. Use the following syntax to remove a sequence:

**Syntax**

```sql
DROP SEQUENCE [schema_name.]sequence_name;
```

**Example: DROP SEQUENCE statement**

In the following example, the DROP SEQUENCE statement removes the sequence named `customer` from the `pub` schema.

```sql
DROP SEQUENCE pub.customer;
```

For more on the DROP SEQUENCE statement, see *OpenEdge Data Management: SQL Reference*.

ALTER SEQUENCE

The ALTER SEQUENCE statement can be used to change the 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**

The syntax of the ALTER SEQUENCE statement is:

```sql
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 }];
```

Only one attribute of a sequence can be altered 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 64-bit sequence.

**Example: ALTER SEQUENCE statement**
The following example modifies a sequence that is used to generate unique customer numbers when a new customer is inserted into the table `pub.customer`.

```
ALTER SEQUENCE pub.customer
SET CURRVAL 57346147483647;
```

**Example: Altering a sequence of a multi-tenant table**

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;
```

**Example: Altering a sequence of a multi-tenant table using CURRVAL**

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;
```

For more on the `ALTER SEQUENCE` statement, see *OpenEdge Data Management: SQL Reference*.

**Working with existing 32-bit sequences**

In OpenEdge Releases 10.1B and later, you may convert previously created 32-bit sequences to 64-bit sequences using a command line utility. When the utility is executed, all sequences in a database convert from 32-bit to 64-bit. The utility, which can be executed online or offline, does not provide you with the ability to selectively convert sequences. All sequences are converted and once the utility is executed, it cannot be reversed. The conversion utility has the following syntax:

```
proutil database_name -C enableseq64
```

Once the conversion has been performed, the sequence is capable of returning values in the 64-bit range. However, the schema properties set during the sequence creation time remain unchanged. Therefore, to change the schema properties, such as minimum and maximum values within the 64-bit `BIGINT` value range, you must drop the sequence and then create a new sequence with the new 64-bit value range.

**Working with domains**

Domains are database objects that map every user to a user group, domain, in a multi-tenant database. A database where Operating system authentication is enabled, domains are required for user authentication. The extended user name authentication mechanism is of the form, `username@domain_name`.

This section provides information on the SQL statements supported for multi-tenant user authentication in OpenEdge SQL.
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. To create a domain, you must have Database Administrator (DBA) privileges.

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

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.

For more on the CREATE DOMAIN statement, see OpenEdge Data Management: SQL Reference.

DROP DOMAIN

Using the drop domain, you can delete a security domain. To drop a domain, you must have DBA privileges.

Syntax

```
DROP DOMAIN domain_name;
```

Note: To drop a domain, you must first drop all the users associated with the domain.

Example: DROP DOMAIN statement

```
DROP DOMAIN jasper;
```

For more on the DROP DOMAIN statement, see OpenEdge Data Management: SQL Reference.

Notes
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.
2. DROP DOMAIN for every domain defined for the tenant.
3. DROP TENANT.

### Working with tenants

A tenant refers to an individual user or a logically defined group of users in a database. The user can be from an organization, department, or a region within an organization. The grouping and association of users is specific for an application during deployment.

This section provides information on the SQL statements that support multi-tenancy for OpenEdge SQL.

### CREATE TENANT

The **CREATE TENANT** statement:

- Defines a tenant for a multi-tenant database
- Allocates database resources in the existing multi-tenant tables
- Adds a new tenant to an existing multi-tenant group that is defined for a multi-tenant table

#### Syntax

```sql
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 table_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 ] ]
;```

#### Example: 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;
```

Example: Creating a tenant and adding it to groups

The following example shows how to create a tenant and add it to multiple groups:

```
CREATE TENANT Consolidated_Freightways TABLE AREA "CF Data area"
   FOR TABLE pub.farm_location JOIN GROUP AgriDataGroup
   FOR TABLE pub.archive_10yr JOIN GROUP Joint_Archives;
```

For more on the `CREATE TENANT` statement, see *OpenEdge Data Management: SQL Reference*.

**CREATE SUPER-TENANT**

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.

**Syntax**

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

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

Example: Creating a super tenant

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

```
CREATE SUPER_TENANT tenantadmin;
```

For more on the `CREATE SUPER_TENANT` statement, see *OpenEdge Data Management: SQL Reference*.

**ALTER TENANT**

The `ALTER TENANT` statement:
- Modifies the definition of a tenant
- Allows you to change the NO SPACE parameter specified for one or more existing multi-tenant tables
- Adds an existing tenant to an existing multi-tenant group that is defined for a multi-tenant table

When a tenant joins a group, all the tenant’s data for that table can be found in that group.

```
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 ] } ] . . . ;
```

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;
```

For more on the `ALTER TENANT` statement, see *OpenEdge Data Management: SQL Reference*.

**DROP TENANT**

Deletes a tenant from a multi-tenant database.

If a tenant that is being dropped belongs to one or more groups, it is first disassociated from the group(s) before being dropped.

```
DROP TENANT [owner_name]tenant_name;
```

The following example shows how to drop the tenant, `mtcustomer`, from the multi-tenant table:

```
DROP TENANT pub.mtcustomer;
```

For more on the `DROP TENANT` statement, see *OpenEdge Data Management: SQL Reference*. 
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.

```
SHOW [ ALL TENANT | TENANT FOR tenant_name ] [ FOR TABLE table_name ]
   [ FULL | PRO_NAME | TABLE | TABLE_INDEX ];
```

The following example shows how to drop the tenant, mtcustomer, from the multi-tenant table:

```
SHOW ALL TENANT FOR TABLE pub.mtcustomer;
```

For more on the SHOW TENANT statement, see *OpenEdge Data Management: SQL Reference*.

Working with groups

Multi-tenant groups allow 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.

The following sections provide information on the SQL statements that support multi-tenant groups in OpenEdge SQL.

CREATE GROUP

The CREATE GROUP statement creates a group for the specified table in a multi-tenant database and uses the following syntax:

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 ];
```

Example: CREATE GROUP statement

The following example shows how to create a group for the pub.customer table and allocate partition space for 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"
```

For more on the CREATE GROUP statement, see *OpenEdge Data Management: SQL Reference*. 
DROP GROUP

The DROP GROUP statement deletes a group from a multi-tenant database. It deallocates the database resources in the group partition of the multi-tenant table for which the group is defined. All the tenants must be disassociated from the group before dropping it.

The DROP GROUP statement uses the following syntax:

Syntax

```
DROP GROUP group_name;
```

Example: DROP GROUP statement

The following example shows how to delete a group from a multi-tenant database:

```
DROP GROUP carz_group;
```

SHOW GROUP

The SHOW GROUP statement returns a result set with information about the database tables and the existing tenants that are associated with one or more groups.

The SHOW GROUP statement uses the following syntax:

Syntax

```
SHOW { ALL GROUP [ { FOR TABLE table_name | FOR TENANT tenant_name } ] | GROUP FOR group_name } [ FULL | PRO_NAME ];
```

Examples: SHOW GROUP statement

The following example shows how to get the basic information about a specific group:

```
SHOW GROUP FOR customer_group3;
```

The following example shows how to get full information about all the groups defined for the pub.mtcustomer 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;
```
For more on the SHOW GROUP statement, see OpenEdge Data Management: SQL Reference.

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.

The ALTER GROUP statement uses the following syntax:

**Syntax**

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

**Example: Alter a group to add tenants**

The following example illustrates the ALTER GROUP statement:

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

**Example: Alter a group to drop tenants**

The following example illustrates the ALTER GROUP statement:

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

For more on the ALTER TENANT statement, see OpenEdge Data Management: SQL Reference.

Working with partitions

Table partitioning enables a table to be divided into multiple self-contained sub-sections (partitions) that can be located in different physical areas. A row never spans a partition.

OpenEdge SQL supports the following kinds of partitions:

- **LIST** — A LIST partition is defined by a single, specific column value for a specific column. A column's value in a table row must be equal to the value defined for a partition, if the row is to be assigned to that partition. NULL cannot be a list partition key value. A list partition allows only one value per partition definition. Using <= for a list partition results in a syntax error.
• **RANGE** — A RANGE partition is defined by a logical range over a column's value for a specific column. A range has a lower bound and an upper bound value and must be explicitly defined by, at least, an upper bound value. The lower bound value of a range can span values from negative infinity to positive infinity. The upper bound must be a value that can be expressed as a regular SQL literal constant, such as 31-December-2025. The upper bound value must be greater than any expected value that applications use for a column. A range must not contain any gaps. For example, a date range of 2005-2015 must not exclude 2009. Values can be excluded from the beginning of all the ranges or from the end. Also, ranges cannot overlap. For example, if you are partitioning based on a date field and defining partitions for a year's worth of data, your partitions define ranges of January 1 through December 31, for each year.

• **Subpartitions** — A partition can be further divided into subpartitions. Subpartitions are defined by multiple partition key columns. Each partition key column, after the first, defines a subpartition. Subpartitions have the same attributes of a partition like optional name, range or list types, etc. There can be up to 15 levels of subpartitioning. 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. The following types of subpartitions are supported:
  - **LIST-RANGE** — A LIST-RANGE partition first defines a partition by a list, and then further partitions the data based on a range.
  - **LIST-LIST-RANGE** — Only the last subpartition can be a RANGE partition.

• **Composite** — A composite partition is a partition that contains the data for multiple partitions.

The following section provides information on the SQL statement specific to partitioned tables.

**SHOW PARTITION**

The SHOW PARTITION statement 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 statement uses the following syntax:

**Syntax**

```
SHOW { ALL PARTITION | PARTITION FOR partition_name } [ FOR TABLE table_name ] [ FULL | PRO_NAME | TABLE | PRO_AREA_NAME ] [ COLUMN SIZE size ]
```

**Examples: SHOW PARTITION statement**

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;
```

For more on the `SHOW PARTITION` statement, see *OpenEdge Data Management: SQL Reference*.

**Maintaining data integrity**

Integrity constraints are application rules that the database enforces. You define integrity constraints on base tables to ensure data integrity. An integrity constraint can specify unique values for a column, validate values of a column, or provide referential integrity.

Referential integrity ensures that a specific piece of data is consistent throughout the database. For instance, a customer's name will be spelled the same way when it appears in tables for purchase orders, merchandise shipments, and accounts receivable.

**Need for integrity constraints**

Integrity constraints are necessary because data in a database must be valid and consistent at all times. Data might be inconsistent because of entry errors, duplicate entries of rows, or other violations.

For example, to properly track a company’s employee information, each employee should be assigned a unique identification number. To ensure this, specify a `UNIQUE` constraint on the column that contains the employee number (`emp_no`), as shown in the following example.

```
CREATE TABLE employee_info (  
  emp_no INTEGER NOT NULL UNIQUE,  
  first_name VARCHAR(20) NOT NULL,  
  last_name VARCHAR(20) NOT NULL,  
  title VARCHAR(20)  
) ;
```

**Types of integrity constraints**

SQL provides four types of integrity constraints:

- Check constraints:
  - Column-level check constraint
  - Table-level check constraint

- Primary key specifications
- Candidate key specifications

SQL allows you to specify an integrity constraint, and to refer to that constraint in other SQL statements. The database assigns a constraint name if you do not specify one.
The following example shows the assignment of table constraint `prim_constr` on table `supplier_item`. You specify a constraint name with the `CONSTRAINT` keyword.

```sql
CREATE TABLE supplier_item (
    supp_no INTEGER NOT NULL,
    item_no INTEGER NOT NULL,
    qty INTEGER NOT NULL DEFAULT 0
    CONSTRAINT prim_constr
    PRIMARY KEY (supp_no, item_no)
) ;
```

### Check constraints

The values you enter for a row must be valid so that the data in the database is consistent. For example, the city names you enter into the supplier table must correspond to one of the cities where the suppliers are located. The database checks to ensure that each value corresponds to one of the valid city names. You achieve these validations by specifying check constraints during the definition of the table schema. Use check constraints when you want to restrict a column to a set of valid values.

The following example shows how to specify a check constraint on the supplier table. In this example, the city column is defined with a check constraint to verify that values for city are in the set of NEW YORK, BOSTON, DALLAS, or MANCHESTER. This CREATE statement does not use the `CONSTRAINT` keyword in the table definition. The system assigns a constraint name.

```sql
CREATE TABLE supplier (
    supp_no INTEGER NOT NULL,
    last_name CHAR (30),
    status SMALLINT,
    city CHAR (20) CHECK (supplier.city IN ('NEW YORK', 'BOSTON', 'DALLAS', 'MANCHESTER'))
) ;
```

A check constraint on a table specifies a condition on the column values of a row in that table. Whenever you issue an `INSERT` or `UPDATE` statement against a table containing check constraints, the database validates the column values. The `INSERT` or `UPDATE` operation is completed only after successful validation.

You can specify a check constraint at either the column level or the table level.

### Column-level check constraints

In an application, you might decide to check a particular column for valid data whenever you attempt to `INSERT` or `UPDATE` values for that column. For example, you design your database to disallow suppliers from a place called Toxic Island. Use a column-level check constraint for this type of validation.

**Example: Column-level check constraint**
In the following example, there is a column-level check constraint on the city column of the supplier table; this check constraint affects the city column only. When you issue an `INSERT` or `UPDATE` operation against the supplier table involving the city column, the SQL engine validates the column value, ensuring that the column does not contain the value 'Toxic Island'. If the `INSERT` or `UPDATE` statement violates the check condition, the database returns a constraint violation error.

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

Example: INSERT failure due to check constraint

In the following examples, the `INSERT` statement results in an error, and the corresponding row is not inserted into the table.

```
INSERT INTO supplier VALUES (1001, 'Worm', 20, 'Toxic Island') ;
```

Example: Constraint violation error message

```
=== SQL Exception 1 ===
SQLState= ErrorCode=-20116
[JDBC Progress Driver]:Constraint violation (7597)
```

Table-level check constraints

Your application might be required to enforce rules on multiple columns. To specify a constraint on more than one table column, define the constraint at the table level. For example, you might need to enforce a validation check on both the status and the city columns in the supplier table.

Example: Table-level check constraint

In the following example, the table-level check constraint verifies that when the city is CHICAGO, the status must be 20, otherwise the operation returns a table-level check constraint violation.

```
CREATE TABLE supplier (
  supp_no INTEGER NOT NULL,
  last_name CHAR (30),
  status SMALLINT CHECK (supplier.status BETWEEN 1 AND 100 ),
  city CHAR (20) CHECK (supplier.city IN ('NEWYORK', 'BOSTON', 'CHICAGO', 'MANCHESTER' ) ),
  CHECK (supplier.city <> 'CHICAGO' OR supplier.status = 20)
) ;
```
Since the check constraint specification involves more than one column, you must specify it at the table level. If an `INSERT` or `UPDATE` statement violates the check condition, the database returns an error.

**Example: Table-level check constraint violation**

The following example shows an `INSERT` statement for the supplier table created in the previous example. This `INSERT` operation results in a check constraint violation.

```
INSERT INTO supplier VALUES (1001, 'John', 40, 'CHICAGO') ;
```

**Primary keys**

A primary key consists of one or more columns in a table that uniquely identifies each row. For example, the `supp_no` column value in the supplier table must be unique. Every row of the table is uniquely identified by this column value. A table can contain only one primary key constraint. If you supply a duplicate value for a primary key column in an `INSERT` operation, the operation returns an error.

You can design your database table so that there is only one column that distinguishes a given row from other rows. In this case, a single column is the unique identifier of the table. For example, the `supp_no` column is a primary key for the supplier table. Primary key constraints are defined in the column definitions of a table.

In the following example, the `supp_no` column is a unique identifier in the supplier table, and the key consists of only one column. This example shows how to create a column-level primary key on the supplier table.

```
CREATE TABLE supplier
{
    supp_no INTEGER NOT NULL PRIMARY KEY,
    last_name CHAR (30),
    status SMALLINT,
    city CHAR (20)
}
```

**Candidate keys**

If you design a table to require that a column or combination of columns define a row as unique, you define the columns with a candidate key constraint.

In the following example, the employee number (`empno`) is the primary key in the employee table because it uniquely identifies each row. Each entry in the employee social security column must also be distinct. Because a primary key has been designated already for the table, you must place a candidate key constraint on the `ss_no` column.

```
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
}
```
You declare a column as a candidate key by using the keyword **UNIQUE**. Precede the **UNIQUE** keyword with the **NOTNULL** specification. Like a primary key, a candidate key also uniquely identifies a row in a table. Note that a table can have only one primary key, but can have any number of candidate keys.

If you supply a duplicate value for a candidate key in an **INSERT** or **UPDATE** operation, the operation returns an error.

### Referential constraints

Relational databases often contain tables that have identical columns. To maintain referential integrity, the data in the columns must remain consistent. Referential constraints ensure this integrity.

In the following example, the value in the **item_no** column of the **supplier_item** table depends on the value in the **item_no** column of the **item** table. The **item_no** column of the **supplier_item** table references the **item_no** column of the **item** table. The **item_no** column is a foreign key in the **supplier_item** table.

```sql
CREATE TABLE supplier_item
(
  suppl_no INTEGER NOT NULL PRIMARY KEY,
  item_no INTEGER REFERENCES item (item_no),
  quantity INTEGER
) ;
```

### Foreign key constraint

A foreign key is a column that references a primary key of another table. The foreign key value either is **NULL** or exists as the primary key value. The table that contains the foreign key is called the referencing table. The table that contains the primary key is called the referenced table.

During **INSERT** or **UPDATE** operations on a table containing a foreign key, the database checks to determine if the foreign key value matches a corresponding primary key value. If it does not match, the operation returns an error.

During **UPDATE** or **DELETE** operations on a table containing a primary or candidate key, if the values to be deleted or updated match the foreign key of the referencing table, the operation returns an error. A value corresponding to a primary or candidate key cannot be updated or deleted if there are references to it.

When you want to drop a table containing a primary or candidate key, the database checks to see if the table has any references to it. If there are tables containing foreign keys that reference the primary or candidate keys of the table you want to drop, the operation returns an error.

In the following example, **item_no** is the foreign key referencing the **item** table, and the foreign key is specified at the column level.

```sql
CREATE TABLE supplier_item
(
  supp_no INTEGER NOT NULL PRIMARY KEY,
  item_no INTEGER NOT NULL REFERENCES item,
  qty INTEGER
) ;
```
If a foreign key references a candidate key, you must name the referenced column in a column list. If a foreign key references a primary key, the column list is optional.

The following example illustrates both conditions. In the example, `invoice.item_no` references the primary key of the `item` table. The `invoice.partnum` column references `parts.part_no`. Since `parts.part_no` is a primary key, the parts (part_no) column list reference in `invoice.part_no` is optional.

```sql
CREATE TABLE invoice (
  inv_no INTEGER NOT NULL PRIMARY KEY,
  item_no INTEGER REFERENCES item,
  part_no CHAR(3) NOT NULL REFERENCES parts (part_no),
  qty INTEGER NOT NULL
) ;
```

## Handling cycles in referential integrity

A cycle exists when a series of primary-key-foreign-key relationships exists within a group of tables in a database.

**Example: Table reference cycle**

In the following example, the `parts.distrib_no` column references the primary key of the `distributor` table, and the `distributor.part_no` column references the primary key of the `parts` table. Each of the tables references the other, forming a cycle.

```sql
CREATE TABLE parts
(
  part_no INTEGER NOT NULL PRIMARY KEY,
  part_name CHAR (19),
  distrib_no INTEGER REFERENCES distributor
) ;
CREATE TABLE distributor
(
  distrib_no INTEGER NOT NULL PRIMARY KEY,
  distrib_name CHAR (19),
  address CHAR (30),
  phone_no CHAR (10),
  part_no INTEGER REFERENCES parts
) ;
```

**Example: Single-table reference cycle**

A special case of the cycle in referential integrity occurs when a foreign key of a table references the primary key of the same table. The following example shows this single-table cycle.

```sql
CREATE TABLE employee
(
  empno INTEGER NOT NULL PRIMARY KEY,
  ename CHAR (30) NOT NULL,
  deptno INTEGER NOT NULL,
  mgr_code INTEGER REFERENCES employee(empno)
) ;
```
Creating tables in cycles

To create a table cycle:

1. Create the first table with a reference to a table that is not yet created. Although the `CREATE TABLE` succeeds, it is marked incomplete. The `INSERT`, `UPDATE`, `SELECT`, and `DELETE` operations are not allowed on an incomplete table.

2. Create the referenced table with a primary or candidate key. The definition of the referencing table becomes complete. If the second table also contains a foreign key that references a table that is not yet created, this second table is also marked incomplete. This process continues until you create the last table.

Inserting rows in a cycle

insert rows into tables that form a cycle:

1. Insert rows into one of the tables that forms the cycle, with `NULL` values in the foreign key columns. If the foreign key is `NULL`, the database does not check for a match between the foreign key and the corresponding primary key. The insert succeeds. This is the referencing table.

2. Update or insert the values in the primary keys of the second, referenced table.

3. Update the foreign key values of the previous table, the referencing table.

The following example shows how to insert or update values into the `employee` table. This table forms a single-table cycle. First insert `NULL` into the `mgr_code` column. After you insert rows, update the values of the `mgr_code` column.

```sql
CREATE TABLE employee (
    empno INTEGER NOT NULL PRIMARY KEY,
    ename CHAR (30) NOT NULL,
    deptno INTEGER NOT NULL,
    mgr_code INTEGER NOT NULL REFERENCES employee (empno)
) ;
INSERT INTO employee VALUES (100, 'JOHN', 10, NULL) ;
INSERT INTO employee VALUES (500, 'MARY', 30, NULL) ;
INSERT INTO employee VALUES (101, 'ANITA', 10, NULL) ;
INSERT INTO employee VALUES (501, 'ROBERT', 30, NULL) ;
UPDATE employee set mgr_code = 101 where empno = 100 ;
UPDATE employee set mgr_code = 501 where empno = 500 ;
/*
** Anita is John's manager.
** John's employee row references Anita's employee row.
** Robert is Mary's manager.
** Mary's employee row references Robert's employee row.
** The mgr_code is still NULL in Anita's row and in Robert's row.
** To set the mgr_code in Anita's row and Robert's row:
**  1. Insert rows for Anita's manager and Robert's manager
**  2. Update Anita's row and Robert's row
*/
```
Working with SQL utilities

SQL utilities are programs that facilitate the administration of your OpenEdge database. The SQLDUMP, SQLLOAD, and SQLSCHEMA utilities are used to perform a variety of tasks including:

- Creating a new version of a database
- Economizing disk space
- Migrating a database to a different operating system
- Updating database schema

Using the SQLDUMP utility

The SQLDUMP utility is a command-line utility that dumps application data from SQL tables into one or more files. You can load the data from the files into another database with the SQLLOAD utility. The SQLDUMP utility does not dump data from ABL tables. The utility is available for only those tables that were created with SQL.

To dump data from multi-tenant tables to one or more files, the utility enables you to use the new command line option (-n) to specify the list of tenants for which the table data must be dumped.

You can also use the SQLDUMP utility to dump data of multi-tenant groups from the existing multi-tenant database.

You can also use the SQLDUMP utility to dump data from partitioned tables at the partition level. For more information, see the .

Syntax

The SQLDUMP utility has the following syntax:

```
sqldump -u user_name[@domain_name] [-a password]
          [-C code-page-name] -t [owner_name]table_name1
          [,owner_name]table_name2, ...]
          [-n tenant_name1, ...]
          [-g group_name1, ...]
          [-p partition_name1, ...] database_name
```

The SQLDUMP utility writes user data in row order into ASCII records with variable-length format. The column order in the files is identical to the column order in the tables. The utility writes both format and content header records to the dump file. You can dump multiple tables in a single execution by specifying multiple table names, separated by commas. Make sure there are no spaces before or after commas in the table list.
Data for one table always goes to a single dump file. Each dump file corresponds to one database table. For example, if you specify 200 tables in the $SQLDUMP$ command, you will create 200 dump files. The $SQLDUMP$ utility assigns the filenames that correspond to the $owner_name$ and $table_name$ in the database, with the file extension .dsql. If a dump file for a specified table already exists, it will be overwritten and replaced. Dump files are created in the current working directory.

The format of each record in a dump file is similar to the ABL .d file format, in that it:

- Converts all values to character representation
- Delimits CHARACTER values with double quotes
- Can contain any embedded characters except NULL values, allowing commas, new lines, and other control characters
- Uses two sets of double quotes to escape embedded double quotes
- Delimits NUMERIC and other noncharacter data types using a space
- Processes TIMESTAMP data as if it were CHARACTER data
- Has a size limit of 2K for a single column value
- Has a maximum record length of 32K for dump file records

Any error is a fatal error, and $SQLDUMP$ halts the dumping process so that data integrity will not be compromised. $SQLDUMP$ reports errors to standard output.

After successful processing, $SQLDUMP$ writes a summary report to standard output. For each table $SQLDUMP$ processes, the report shows:

- Table name
- Dump filename
- Number of records dumped
- Number of bytes dumped
- Number of seconds required for processing

**Example: SQLDUMP from selected tables**

The following example directs the $SQLDUMP$ utility to write the data from two tables to two dump files. The $user_name$ and $password$ for connecting to the database are $tucker$ and $sulky$. The $tucker$ account must have the authority to access the customers and products tables in database $salesdb$ with $owner_name$ $martin$.

```
sql_dump -u tucker -a sulky -t martin.customers,martin.products
progress:T:thunder:4077:salesdb
```

**Example: SQLDUMP based on table names**
The following example directs the SQLDUMP utility to write the data from all tables in the salesdb database that begin with any of these strings: cust, invent, and sales, and having any owner name that the user tucker has authority to access. The user_name and password for connecting to the database are tucker and sulky.

```
sqldump -u tucker -a sulky -t%.cust%,%.invent%,%.sales%
progress:T:thunder:4077:salesdb
```

The following example directs the SQLDUMP utility to write the data from all tables for all owner names in the salesdb database.

Example: SQLDUMP of entire database

```
sqldump -u tucker -a sulky s-t %.% progress:T:thunder:4077:salesdb
```

Note: The database_name must be the last parameter given. Each dump file records character set information in the identifier section of each file.

Example: Dump file

The following example depicts a dump file.

```
A^B^CProgress sqlschema v1.0 Quote fmt
A^B^CTimestamp 1999-10-19 19:06:49:0000
A^B^CDatabase dumpdb.db
A^B^CProgress Character Set: iso8859-1
A^B^CJava Character Set: Unicode UTF-8
A^B^CDate Format: MM/DD/YYYY
```

The character set recorded in the dump file is the client character set. The default character set for all non-JDBC clients is taken from the local operating system through the operating system APIs. JDBC clients use the Unicode UTF-8 character set.

To use a character set different than that used by the operating system, set the SQL_CLIENT_CHARSET environment variable to the name of the preferred character set. You can define any ABL-supported character set name. The name is not case-sensitive.

SQLDUMP does not support the following characters in schema names:

- Double quote ("")
- Forward slash (/)
- Backslash (\)

SQLDUMP, however, does support schema names that contain special characters such as a blank space, a hyphen (-), or a pound sign (#). These names must be used as delimited identifiers. Therefore, when specifying names with special characters on a UNIX command line, follow these rules:

- Use double quotes to delimit identifiers.
- So that the command line does not strip the quotes, use a backslash (\) to escape the double quotes used for delimited identifiers.
Use double quotes to enclose any names with embedded spaces, commas, or characters special to a command shell (such as the Bourne shell). This use of quotes is in addition to quoting delimited identifiers.

For example, to dump the table Yearly Profits, use the following UNIX command-line syntax:

```
sqldump -t "\"Yearly Profits\"" -u xxx -a yyydatabase_name
```

In Windows, the command interpreter rules for the use of double quotation marks varies from UNIX.

By default, SQLDUMP displays promsgs messages using the code page corresponding to code-page-name. That is, if you are dumping a Russian database, and code-page-name specifies the name of a Russian code page, the client displays promsgs messages using the Russian code page (unless you specify a different code page by setting the client's SQL_CLIENT_CHARSET_PROMSGS environment variable).

### Dumping Multi-tenant tables

The SQLDUMP utility is extended to support multi-tenancy. This utility allows you to dump data from multi-tenant tables—depending on the user type—to an external file with the new (- n) command line option. The following examples provide clarity on dumping multi-tenant tables:

The following example directs the SQLDUMP utility to dump data from the tenants ten1 and ten2 to two SQL dump files respectively. The user_name and password to connect to the database are supertenUser@superdom and superten. The supertenUser account in the superdom domain must have the authority to access the mttab1 table and the _tenant system table in mtdb database.

To separate the tenant specific data the SQLDUMP utility creates separate directories for each tenant. ten1/<OWNER>.MTTAB1.DSQL and ten2/<OWNER>.MTTAB1.DSQL are the two directories that are created to dump tenant data.

**Example: SQLDUMP from multi-tenant tables**

```
sqldump -u supertenUser@superdom -a superten -t mttab1 -n ten1,ten2
```

If the regTenantUser is mapped to a regular tenant, then the following example directs the SQLDUMP utility to dump the data for the regTenantUser tenant's partition.

**Example: SQLDUMP by a regular tenant**

```
sqldump -u regTenantUser@OpenEdgeA -a regTenant -t mttab1
```

If the regTenantUser is mapped to a regular tenant, then the following example directs the SQLDUMP utility to dump the data for the regTenantUser tenant's partition.

```
sqldump -u regTenantUser@OpenEdgeA -a regTenant -t mttab1 -n ten1
```
If regTenantUser is not mapped to ten1 tenant, the above statement throws an error.

**Example: SQLDUMP by a DBA**

If the dbaUser is a DBA, then, the following example directs the SQLDUMP utility to dump the tenant-specific data for all the tenants in their respective directory.

```
sqldump -u dbaUser -a dba -t mttab1 progress:T:localhost:9999:mtdb
```

If the superTenUser is a super-tenant, then, the following example directs the SQLDUMP utility to dump all the tenants which start with the word ‘ten’ from the table mttab.

**Example: SQLDUMP using a % operator**

```
sqldump -u superTenUser@superdom -a superten -t mttab1 -n ten% progress:T:localhost:9999:mtdb
```

If the dbaUser is a DBA, then the following example directs the SQLDUMP utility to dump all the tenants which start with the word ‘ten’ from all the tables that start with mttab.

```
sqldump -u dbaUser -a dba -t mttab% -n ten% progress:T:localhost:9999:mtdb
```

**Dumping partitioned tables**

The SQLDUMP utility enables you to dump data from partitioned tables to an external file with the command line parameter `-p`. The SQLDUMP utility dumps each partition into a separate file. Since these files belong to a single table, the SQLDUMP utility places them in the same directory.

The name of the directory is a combination of the name of the owner and the table name in the following format:

```
owner_name.table_name
```

The file name is a combination of the name of the owner, the table name, and the partition name in the following format:

```
owner_name.table_name.partition_name.dsql
```

The command line parameter `-p` is an optional parameter and specifies a comma-separated list or expressions of one or more partitions to be dumped. It is mutually exclusive with the `-n` and `-g` parameters.

**Notes**

- Existence of the `-p` parameter makes the use of `-t` optional. If the SQLDUMP utility specifies the `-t` parameter and not `-p`, then the utility dumps all the partitions of that table to a single file.

- If `-p` is specified, then the SQLDUMP utility dumps only those partitions that are specified in the `-p` list. If both `-t` and `-p` parameters are used, then each partition that belongs to or qualifies to belong to partition_name must have its table name specified in the `-t` list; otherwise, the SQLDUMP utility returns an error. It ignores the `-p` parameter for normal tables listed in the `-t` list.
The SQLDUMP utility also returns an error if a standalone % operator is specified with partition names or expressions that are not being dumped.

Examples: SQLDUMP for partitioned tables
The following examples illustrate dumping data from the customers table with three partitions, part_1, part_2, and part_3. The user_name and password for connecting to the database are tucker and sulky. The tucker account must have the authority to access the customers table in the salesdb database.

The following example directs the SQLDUMP utility to create a directory named tucker.customers and dump part_1 and part_3 data of the customers table into the files tucker.customers.part_1.dsql and tucker.customers.part_3.dsql. If the files already exist, the SQLDUMP utility overwrites them.

```
sqldump -u tucker -a sulky -t customers -p part_1,part_3
progress:T:thunder:4077:salesdb
```

The following example directs the SQLDUMP utility to search the specified directory for a table whose partition matches part_% and dump partitions of the specified table that match part_% into their respective files. If the files already exist, the SQLDUMP utility overwrites them; if no such files exist, the SQLDUMP utility creates them.

```
sqldump -u tucker -a sulky -t part_%
progress:T:thunder:4077:salesdb
```

Using the SQLLOAD utility
The SQLLOAD utility loads user data from a formatted file into an SQL database. Typically, the source file for the load is created by executing the SQLDUMP utility. The SQLLOAD utility can process a source file created by another application or utility, if the format of the file conforms to SQLLOAD requirements. The file extension made available to SQLLOAD for processing must be .dsql. See the entry on SQLDUMP for a description of the required file format.

To load data onto the multi-tenant tables from a formatted file, the new command-line option (-n) on the utility allows you to specify the list of tenants for which table data must be loaded. You can also use the SQLLOAD utility to load data of multi-tenant groups to the existing multi-tenant database.

Before you can execute SQLLOAD against a database server, the server must be configured to accept SQL connections and must be running.

You can also use the SQLLOAD utility to load data onto partitioned tables. For more information, see the .

Use the following syntax for the SQLLOAD utility:
Syntax

sqlload -u user_name[<@domain_name>]  
[-a password] [-C code-page-name] 
-t [owner_name]table_name1 
[<,owner_name>table_name2, ...] 
[-l log_file_name] 
[-b badfile_name] 
[-e max_errors] 
[-s skipcount] 
[-m max_rows] 
[-F comma | quote] 
[-n tenant_name1, ...] 
[<,group_name1, ...>] 
[<,partition_name1, ...>] database_name

The SQLLOAD utility reads application data from variable-length text-formatted files and writes the data into the specified database. The column order is identical to the table column order. SQLLOAD reads format and content header records from the dump file. You can load multiple tables in a single execution by specifying multiple table names, separated by commas. Data for one table is from a single dump file. Every source file corresponds to one database table. For example, if you specify 200 tables in the SQLLOAD command, you will load 200 database tables.

The format for the records in the input files is similar to the ABL .d file dump format. The maximum record length SQLLOAD can process is 32K.

Each database record read is share-locked, for consistency. You must ensure that the SQL Server has a lock table large enough to contain one lock for every record in the table. The default lock table size is 10,000 locks.

SQLLOAD writes any errors to standard output and halts the loading process for any error so that data integrity is not compromised.

Example: SQLLOAD of two dump files

The following example directs the SQLLOAD utility to load the data from two dump files into the salesdb database. The input files to SQLLOAD must be tucker.customers.dsql and tucker.products.dsql.

sqlload -u tucker -a sulky -t tucker.customers,tucker.products 
progress:T:thunder:4077:salesdb

Example: SQLLOAD from appropriately named files

The following example directs SQLLOAD to load the data from all appropriately named dump files into the specified tables in the salesdb database.

sqlload -u tucker -a sulky -t %.cust%,%.invent%,%.sales% 
progress:T:thunder:4077:salesdb
The database_name must be the last parameter given.

The character set used by SQLLOAD must match the character set information recorded in each dump file. If the character sets do not match, the load is rejected. You can use the SQL_CLIENT_CHARSET environment variable to specify a character set.

Each dump file you create with SQLDUMP contains character set information about that file. The character set recorded in the dump file is the client character set. The default character set for all non-JDBC clients is taken from the local operating system through the operating system APIs. JDBC clients use the Unicode UTF-8 character set. To use a character set different than that used by the operating system, set the SQL_CLIENT_CHARSET environment variable to the name of the preferred character set. You can define any ABL-supported character set name. The name is not case-sensitive.

At run time, SQLLOAD reports an error if it detects a mismatch between the code page of the dump file being loaded and the code page of the client running SQLLOAD.

By default, SQLLOAD displays promsgs messages using the code page corresponding to code-page-name. That is, if you are restoring a Russian database and code-page-name specifies the name of a Russian code page, the client displays promsgs messages using the Russian code page (unless you specify a different code page by setting the client's SQL_CLIENT_CHARSET_PROMSGS environment variable).

SQLLOAD does not support the following characters in schema names:

- Double quote ("")
- Forward slash (/)
- Backslash (\)

SQLLOAD, however, does support schema names that contain special characters, such as a blank space, a hyphen (-), or a pound sign (#). These names must be used as delimited identifiers. Therefore, when specifying names with special characters on a UNIX command line, follow these rules:

- Use double quotes to delimit identifiers.
- So that the command line does not strip the quotes, use a backslash (\) to escape the double quotes used for delimited identifiers.
- Use double quotes to enclose any names with embedded spaces, commas, or characters special to a command shell (such as the Bourne shell). This use of quotes is in addition to quoting delimited identifiers.

**Example: SQLLOAD of files with delimited identifiers**

To load the table Yearly Profits, use the UNIX command-line syntax, as shown in the following example.

```
sqlload -u xxx -a yyy -t ""Yearly Profits"" database_name
```

In Windows NT, the command interpreter rules for the use of double quotation marks varies from UNIX.
Loading Multi-tenant tables

The **SQLLOAD** utility is extended to support multi-tenancy. It allows you to load data onto the multi-tenant tables—depending on the user type from the formatted file(s) with the new 
\((-\ n)\)
command-line option. The following examples provide clarity on dumping multi-tenant tables:

The following example directs the **SQLLOAD** utility to load the data from the dump files in the tenant directories `ten1` and `ten2` to the `mttab1` table. The user_name and password to connect to the database are `supertenUser@superdom` and `superten`. The `supertenUser` account in the `superdom` domain must have the authority to access the `mttab1` table and the `_tenant` system table in the `mtdb` database.

The tenant directories are created for every defined tenant. The following example illustrates the creation of tenant directories when the tenants are dumped to a formatted file using **SQLLOAD** utility.

**Example: SQLLOAD from multi-tenant tables**

```
sqlload -u supertenUser@superdom -a superten -t mttab1 -n ten1,ten2 progress:T:localhost:9999:mtdb
```

**Example: SQLLOAD by a regular tenant**

If the `regTenantUser` is mapped to a regular tenant, then the following example directs the **SQLLOAD** utility to load the data of the partition of the tenant to `mttab1` table.

```
sqlload -u regTenantUser@OpenEdgeA -a regTenant -t mttab1 progress:T:localhost:9999:mtdb
```

**Example: SQLLOAD by a DBA**

If the `dbaUser` is a DBA, then the following example directs the **SQLLOAD** utility to load the tenant-specific data for all the tenants from their respective directories for the table `mttab1`.

```
sqlload -u dbaUser -a dba -t mttab1 progress:T:localhost:9999:mtdb
```

If the `superTenUser` is a super-tenant, then the following example directs the **SQLLOAD** utility to load all the data associated with the tenants which start with the word ‘ten’ of the table `mttab1`.

**Examples: SQLLOAD using a % operator**

```
sqlload -u superTenUser@superdom -a superten -t mttab1 -n ten% progress:T:localhost:9999:mtdb
```

If the `dbaUser` is a DBA, then the following example directs the **SQLLOAD** utility to load all data for the tables which start with the word ‘mttab’ for all the tenants whose name starts with ‘ten’.

```
sqlload -u dbaUser -a dba -t mttab% -n ten% progress:T:localhost:9999:mtdb
```
Example: SQLLOAD to load data for all tenants for a table

If the dbaUser is a DBA, then the following example directs the SQLLOAD utility to load all the data for tenants for the table mttab.

```
sqlload -u dbaUser -a dba -t mttab progress:T:localhost:9999:mtdb
```

Loading partitioned tables

The SQLLOAD utility allows you to load data onto partitioned tables with the command line parameter `-p`.

The command line parameter `-p`, is an optional parameter and specifies a comma-separated list or expressions of one or more partitions to be loaded. It is mutually exclusive with the `-n` and `-g` parameters.

Notes

- Existence of the `-p` parameter makes the use of `-t` optional. If the `-p` parameter is not specified, then the SQLLOAD utility treats the table as a normal table. If the `-p` parameter is specified, then the SQLLOAD utility loads only those partitions that are specified in the `-p` list.

- If both `-t` and `-p` parameters are specified, then each partition that belongs to or qualifies to belong to `partition_name` must have its table name specified in the `-t` list; otherwise, the SQLLOAD utility returns an error. It ignores the `-p` parameter for normal tables listed in the `-t` list.

- The SQLLOAD utility also returns an error if a standalone % operator is specified with partition names or expressions that are not being loaded.

Examples: SQLLOAD for partitioned tables

The following examples illustrate loading data onto the customers table with three partitions, `part_1`, `part_2`, and `part_3`. The user name and password for connecting to the database are tucker and sulky. The tucker account must have the authority to access the customers table in the salesdb database.

The following example directs the SQLLOAD utility to search for a directory named `tucker.customers` and load `part_1` and `part_3` data of the customers table from the files `tucker.customers.part_1.dsql` and `tucker.customers.part_3.dsql`. If the files do not exist, the SQLLOAD utility returns an error.

```
sqlload -u tucker -a sulky -t customers -p part_1,part_3
progress:T:thunder:4077:salesdb
```

The following example directs the SQLLOAD utility to search the specified directory for a table whose partition matches `part_%` and load partitions of the specified table that match `part_%` from their respective files. If the files do not exist, the SQLLOAD utility returns an error.

```
sqlload -u tucker -a sulky -p part_%
progress:T:thunder:4077:salesdb
```
Using the SQLSCHEMA utility

The SQLSCHEMA utility is a command-line utility that writes SQL database schema components to an output file selectively. You can capture table definitions including table constraints, views, stored procedures including related privileges, and triggers. At the command line you specify which components to dump.

The extended support for multi-tenancy includes domain name, name of the tenant, and sequence name utility option in the SQLSCHEMA utility to write SQL database schema definitions for domains, tenants, and sequences respectively. The existing table utility option is enhanced to support writing definitions for a multi-tenant table.

The SQLSCHEMA utility also supports writing definitions for multi-tenant groups.

Use the following syntax for the SQLSCHEMA utility:

Syntax

```
sqlschema -u user_name
  [-a password]
  [-t [owner_name]table_name1
   [owner_name]table_name2, ...]
  [-t [owner_name]view_name1
   [owner_name]view_name2, ...]
  [-p [owner_name]procedure_name, ...]
  [-T [owner_name]trigger_name, ...]
  [-G [owner_name]procedure_name, ...]
  [-g [owner_name]table_name, ...]
  [-s [owner_name]table_name, ...]
  [-o output_file_name]
  [-d domain_name]
  [-n tenant_name]
  [-q sequence_name]
  [-r group_name]
  [-z ]database_name
```

Table definitions include the database area name for the table, derived from a scan of the area and objects. When SQLSCHEMA writes a table definition, it does not automatically write associated triggers, synonyms, or privileges. These must be explicitly specified on the command line. Capturing database schema requires privileges to access the requested components.

Example: SQLSCHEMA utility for writing object definitions

The following example directs the SQLSCHEMA utility to write table definitions and trigger information. The output goes to the screen since no output_file_name is specified. Since the user_name and password are not specified, SQLSCHEMA will prompt the user for these values.

```
sqlschema -t tucker.customers,tucker.products -T
tucker.customers,tucker.products progress:T:thunder:4077:salesdb
```
Example: SQLSCHEMA for writing object definitions to output file

The following example directs the SQLSCHEMA utility to write table definitions to an output file named salesdbschema.dfsql.

```
sqlschema -u tucker -a sulky -t %.cust%,%.invent%,%.sales% -o salesdbschema.dfsql progress:T:thunder:4077:salesdb
```

Note: Each output file created by the SQLSCHEMA utility records character set information about the contents of the file. When you use SQLSCHEMA to dump schema information from a database, the schema is written in Unicode UTF-8.

For more information on SQL utilities and database administration, see OpenEdge Data Management: Database Administration.

Schema definition for multi-tenant tables

SQLSCHEMA utility is extended to support the following schema definitions in multi-tenant database:

- Writing schema definitions for domains
- Writing schema definitions for tenants
- Writing schema definitions for shared and multi-tenant tables
- Writing schema definition for shared and multi-tenant sequences

The following example illustrates the extended support by SQLSCHEMA for multi-tenancy:

Example: SQLSCHEMA to write schema definitions for a domain

The following example directs the SQLSCHEMA utility to write schema definitions for the domain OpenEdgeA.

```
sqlschema -u mtuser@mtdomain -a mtuser -d OpenEdgeA progress:T:localhost:2222:sportsdb
```

Example: SQLSCHEMA to write schema definitions for two domains

The following example directs the SQLSCHEMA utility to write schema definitions for two domains OpenEdgeA and OpenEdgeB respectively.

```
sqlschema -u mtuser@mtdomain -a mtuser -d OpenEdgeA,OpenEdgeB progress:T:localhost:2222:sportsdb
```

Example: SQLSCHEMA using a % operator

The following example directs the SQLSCHEMA utility to write schema definitions for domains with Access as its last 6 characters.

```
sqlschema -u mtuser@mtdomain -a mtuser -d %Access progress:T:localhost:2222:sportsdb
```
Example: SQLSCHEMA using the operator _ (underscore)
The following example directs the SQLSCHEMA utility to write schema definitions for domains with OpenEdge as its first 8 characters followed by a single character.

```
sqlschema -u mtuser@mtdomain -a mtuser -d OpenEdge_ progress:T:localhost:2222:sportsdb
```

Example: SQLSCHEMA to write schema definitions for a tenant
The following example directs the SQLSCHEMA utility to write schema definitions for a tenant.

```
sqlschema -u mtuser@mtdomain -a mtuser -n T1 progress:T:localhost:2222:sportsdb
```

Example: SQLSCHEMA to write schema definitions for two tenants
The following example directs the SQLSCHEMA utility to write schema definitions for tenants ten1 and ten2 respectively.

```
sqlschema -u mtuser@mtdomain -a mtuser -n ten1,ten2 progress:T:localhost:2222:sportsdb
```

Example: SQLSCHEMA to write schema definition for tenants whose names start with Test
The following example directs the SQLSCHEMA utility to write schema definitions for the tenants which start with ‘Test’.

```
sqlschema -u mtuser@mtdomain -a mtuser -n Test% progress:T:localhost:2222:sportsdb
```

Example: SQLSCHEMA to write a schema definition for tenant whose name starts with T followed by a single character
The following example directs the SQLSCHEMA utility to write schema definitions for the tenants which start with a ‘T’ followed by a single character.

```
sqlschema -u mtuser@mtdomain -a mtuser -n T_ progress:T:localhost:2222:sportsdb
```

The following example directs the SQLSCHEMA utility to write the schema definitions for a multi-tenant table.

```
sqlschema -u mtuser@mtdomain -a mtuser -t pub.acct_payable progress:T:localhost:2222:sportsdb
```
The following example directs the SQLSCHEMA utility to write schema definitions for a sequence, seq1.

```
sqlschema -u mtuser@mtdomain -a mtuser -Q seq1
progress:T:localhost:2222:sportsdb
```

The following example directs the SQLSCHEMA utility to write schema definitions for the two sequences seq1 and seq2.

```
sqlschema -u mtuser@mtdomain -a mtuser -Q seq1,seq2
progress:T:localhost:2222:sportsdb
```

The following example directs the SQLSCHEMA utility to write schema definitions for sequences that start with 'Test'.

```
sqlschema -u mtuser@mtdomain -a mtuser -Q Test%
progress:T:localhost:2222:sportsdb
```

The following example directs the SQLSCHEMA utility to write schema definitions for sequences that start with seq followed by a single character.

```
sqlschema -u mtuser@mtdomain -a mtuser -Q seq_
progress:T:localhost:2222:sportsdb
```

The following example illustrates the use of -z command line option in the SQLSCHEMA utility invocation.

Assuming the database sportsdb is empty, -z option indicates that the dumped definitions for CREATE TENANT statements do not have over-ridden area information for their corresponding tables.

```
sqlschema -u mtuser@mtdba -a mtuser -Z -t pub.customer,pub.acct_payable -n TN1 progress:T:localhost:2222:sportsdb
```

The -z command line option is ignored if the database in not empty.

The following example directs the SQLSCHEMA utility to write the schema definitions for a multi-tenant group group1.

```
sqlschema -u mtuser@mtdba -a mtuser -r group1
progress:T:localhost:2222:sportsdb
```
The Data Manipulation Language (DML) is used to select, insert, delete, or update information of a database, and is described in the following sections.

**Note:** This chapter gives an overview of DML statements. For complete syntax of each statement, see OpenEdge Data Management: SQL Reference.

For details, see the following topics:

- Using Data Manipulation Language statements
- Using indexes
- Working with join operations
- Using scalar subqueries

### Using Data Manipulation Language statements

The SQL Data Manipulation Language statements are critical to an application's business logic. DML statements retrieve, add, change, and remove database records.

**SELECT**

Use the `SELECT` statement to retrieve information from a database.
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.

The `SELECT` statement uses the following syntax:

**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 ] [ WITH locking_hints ] [ FOR UPDATE
update_condition ];
```

The `SELECT` statement provides you with countless ways to retrieve and analyze the data in your database. Queries vary from simple to sophisticated. They can retrieve information from a single column of a single table, or they can retrieve data meeting specific conditions from many columns across many tables.

The following statements offer just a few examples of how the `SELECT` statement can be used to create queries.

In the following example, the simple `SELECT` statement retrieves all columns from the `Customer` table.

```
SELECT * FROM Customer;
```

In the following example, the statement is easily modified to identify columns from which the data will be retrieved.

```
Select CustNum, Name, City FROM Customer;
```
The simple `SELECT - FROM` combination can even be used to retrieve a single set of results from multiple tables. The following example retrieves the customer and order information from both the `Customer` and `Order` tables.

```sql
SELECT Customer.CustNum, Customer.Name, Order.OrderNum, Order.OrderDate
FROM Customer, Order;
```

The `WHERE` clause can be used to further refine your query. In the following example, the `FROM` clause uses the `WHERE`, `GROUP BY`, and `HAVING` conditions to create a highly specific query statement. The query returns the customer number and number of orders for all customers who had more than 10 orders before March 31, 2008.

```sql
SELECT CustNum, COUNT(*)
FROM Order
WHERE OrderDate < '3/31/08'
GROUP BY CustNum
HAVING COUNT(*) > 10;
```

The `WHERE` clause limits a query to retrieving specified rows based upon a search condition. The `GROUP BY` clause produces a summary query in which similar rows are grouped together. The `HAVING` clause further restricts the `GROUP BY` clause by allowing only those groups that have been specified by a search condition.

Assuming the user is mapped to a DBA or a super-tenant, the following example 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.

```sql
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;
```

Assuming the user is mapped to a DBA or a super-tenant, the following example directs the `SELECT` statement to join a multi-tenant table to another multi-tenant table associated with a multi-tenant group.
When you join a multi-tenant table to a table with a multi-tenant group, as shown in this example, the data rows in the group are displayed exactly once for the entire set of tenants in the group.

```
SELECT * FROM Pub.mtcustomer AS c
INNER JOIN pub.mtorder AS o
ON tenantId_tbl (c) = tenantId_tbl (o)
INNER JOIN pub.mtorderline AS ol
ON c.custnum = o.custnum
AND tenantId_tbl (o) = tenantId_tbl (ol)
;
```

Alternately, you can view multi-tenant data on a tenant-by-tenant basis, including data from a multi-tenant group. Viewing data from a group on a tenant-by-tenant basis means that the data rows in the group are displayed once for each tenant in the group.

This can be accomplished by joining to the OpenEdge schema table _Tenant. In the following example, assume that the table pub.mtOrder is associated with a multi-tenant group for some its tenants’ data.

```
SELECT * FROM pub."_Tenant" AS t
INNER JOIN pub.mtorder AS o
ON t."_Tenant" = tenantId_tbl (o)
INNER JOIN pub.mtorderline AS ol
ON c.custnum = o.custnum
AND tenantId_tbl (o) = tenantId_tbl (ol)
;
```

OpenEdge SQL can access all databases objects created with ABL. ABL can access SQL-created objects, but they must contain data types recognized by ABL and must reside in the PUB schema of an OpenEdge RDBMS.

For more information on the SELECT statement, see OpenEdge Data Management: SQL Reference.

**INSERT**

Use the INSERT statement to add new rows to a table.

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.

The **INSERT** statement uses the following syntax:

**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 |
```

The following example depicts a single row being added to a table.

```
INSERT INTO Customer (CustNum, Name, Address, City, State)
VALUES
(1001, 'Global Fitness', '10 Columbia Street', 'New York', 'NY');
```

**INSERT** statements can also be executed based upon the results of a query expression. In the following example, rows are created in a table of New York customers based on a query of the Customer **table**.

```
INSERT INTO NYCustomer (CustNum, Name)
SELECT CustNum, Name FROM Customer WHERE state = 'NY';
```

The following example 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');
```
The following example lists two \texttt{INSERT} statements that move rows from the DEFAULT partition of \texttt{mtcustomer}, and then distribute the rows with an even customer number to the tenant \texttt{SNCSoftware} and rows with an odd customer number to the tenant \texttt{OEDProducts}.

\begin{verbatim}
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';
\end{verbatim}

\textbf{Note:} If the optional column list is used, then only the values for those columns in the statement are required. Otherwise, values must be specified or returned by a query expression. Using \texttt{VALUES} to specify columns will insert one row into the table. Use the query expression to insert multiple rows.

The following examples direct the \texttt{INSERT} statement to insert rows into the table \texttt{Pub.tpcustomer} that is partitioned by list. Assume that the table is partitioned based on \texttt{SalesRep} as given below:

- \texttt{PART1\_LIST} SalesRep IN \texttt{('SLS')}
- \texttt{PART2\_LIST} SalesRep IN \texttt{('JLP')}
- \texttt{PART3\_LIST} SalesRep IN \texttt{('KIK')}
- \texttt{PART4\_LIST} SalesRep IN \texttt{('BBB')}

\begin{verbatim}
INSERT INTO Pub.tpcustomer (cust_num, SalesRep) VALUES (100, 'SLS');
\end{verbatim}

The above statement inserts rows into the partition \texttt{PART1\_LIST}.

\begin{verbatim}
INSERT INTO Pub.tpcustomer (cust_num, SalesRep) VALUES (101, 'BBB');
\end{verbatim}

The above statement inserts rows into the partition \texttt{PART4\_LIST}.

For more information on the \texttt{INSERT} statement, see \textit{OpenEdge Data Management: SQL Reference}.

\section*{UPDATE}

The \texttt{UPDATE} statement updates the rows and columns of the specified table with the given values for rows that satisfy the \texttt{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.

**Note:** You must have the write permissions on the partitioned table to update its records.

The UPDATE statement uses the following syntax:

**Syntax**

```
UPDATE table_name SET assignment[, assignment] , ... [ WHERE search_condition ]
;
```

**assignment:**

```
column = { expr | NULL } | (column[, column] , ...) = ( expr [, expr ] | (column[, column] , ... ) = (query_expression)
```

In the following example, a simple UPDATE statement is used to revise the credit limit of all customers in the Customer table.

```
UPDATE Customer
    SET CreditLimit = 50000;
```
Use the `WHERE` clause to identify a specific column and row to be updated, as shown in the following example.

```sql
UPDATE Customer
SET CreditLimit = 50000
WHERE Name = 'World Cup Soccer';
```

The following example updates the postal code to '99999' for a customer '1428' for the tenant SNCSoftware.

```sql
UPDATE pub.mtcustomer
SET postalcode = '99999'
WHERE custnum = 1428 AND tenantName_tbl (pub.mtcustomer) = 'SNCSoftware';
```

The following example updates the postal code to '99999' for the customer '1428' for all tenants.

```sql
UPDATE pub.mtcustomer
SET postalcode = '99999'
WHERE custnum = 1428;
```

The following examples illustrate updating rows of a table that is partitioned by RANGE.

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')

```sql
UPDATE Pub.tporder
SET OrderDate = '01/05/1997'
WHERE OrderNum = 1;
```

The above `UPDATE` statement does not change the partition of the row.

```sql
UPDATE Pub.tporder
SET OrderDate = '01/05/1997'
WHERE OrderNum = 2;
```

The above `UPDATE` statement results in changing the partition of a row and moves the row from partition PART3_RANGE to PART1_RANGE.
DELETE

The DELETE statement deletes table rows specified in the statement's WHERE clause. If the optional WHERE clause is not specified, then the DELETE statement deletes all rows of the 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.

The DELETE statement uses the following syntax:

Syntax

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

The following example depicts a simple delete on a single table.

```
DELETE from Customer
   WHERE Name = 'Hoops' ;
```

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';
```

**Note:** A DELETE statement will fail in the event it attempts to remove rows from primary or candidate keys that reference other tables.

For more information on the DELETE statement, see *OpenEdge Data Management: SQL Reference*.

Using indexes

An index is a database object that enables quick information retrieval from a table. OpenEdge SQL employs a B-tree index, which organizes data in ascending or descending order. For every entry in the index table there is a corresponding entry in the database table to which it is associated. This speeds a query because it is more efficient to locate a row by searching a sorted index than by searching an unsorted table. This type of index is ideal for searching for a single value or a range of values.

Create an index when:

- The column is commonly used in a WHERE clause or in a join condition.
- The column contains a large number or a wide range of values.
- Two or more columns are frequently used together in a WHERE clause or a join condition.
- The table is large and most queries are expected to retrieve less than a small percentage of rows.

Do not create an index if:
- The table is frequently updated.
- Most queries retrieve a large percentage of the rows in a table.
- The columns are not frequently used as a condition of a query.

To create or drop index information, use the CREATE INDEX and DROP INDEX statements. For information on working with indexes to optimize query performance, see Optimizing Query Performance on page 209.

Index system catalog tables

The following SQL system catalog tables contain information concerning indexes:

- SYSINDEXES is a core system table. Each row in the table represents one component of each INDEX in the database.
- SYSIDXSTAT provides information on each index in the database.

For more information on the SQL system tables, see OpenEdge Data Management: SQL Reference.

Working with join operations

In order to support the business logic of almost any application, you must be able to retrieve information from several tables in a database and present the data in a single result set. A join is a way of selecting data from two or more related tables using the columns that define their relationship as the join condition. It retrieves all rows from both tables, where there are matches on the join columns.

This section includes the following topics:

- Using inner joins

Using inner joins

An inner join produces a results table consisting of only those rows that correspond to the tables specified in the query. A query expression can specify inner joins in either its FROM clause or its WHERE clause.

The basic syntax for a join is:
Working with join operations

Syntax

```
{ 
    table_ref 
    [ INNER | LEFT [ OUTER ]] JOIN 
    table_ref ON search_condition }
```

The following example demonstrates a join that retrieves information from two tables relating customers and their order information.

```
SELECT order.ordernum, order.custnum, customer.name, order.orderdate, order.shipdate
FROM order INNER JOIN customer
ON order.custnum = customer.custnum;
```

The statement produces the following results:

<table>
<thead>
<tr>
<th>Ordernum</th>
<th>CustNum</th>
<th>Name</th>
<th>OrderDate</th>
<th>ShipDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>53</td>
<td>Offside Hockey</td>
<td>2007-01-26</td>
<td>2007-01-31</td>
</tr>
<tr>
<td>2</td>
<td>81</td>
<td>Off The Wall</td>
<td>2006-10-05</td>
<td>2006-10-10</td>
</tr>
<tr>
<td>3</td>
<td>66</td>
<td>Hero Football</td>
<td>2006-09-23</td>
<td>2006-09-28</td>
</tr>
<tr>
<td>4</td>
<td>83</td>
<td>Swift Running</td>
<td>2007-01-17</td>
<td>2007-01-22</td>
</tr>
<tr>
<td>5</td>
<td>72</td>
<td>Pro Skates</td>
<td>2007-02-12</td>
<td>2007-02-17</td>
</tr>
</tbody>
</table>

Employing a table alias

Notice that in the previous query, each column name uses the standard naming convention of table-name.column-name. This can be cumbersome and prone to errors, especially with complex joins. A more convenient way is to define table aliases in the WHERE clause and invoke them throughout the query. The following example modifies the inner join in the previous example by employing table aliases.

```
SELECT o.ordernum, o.custnum, c.name, o.orderdate, o.shipdate
FROM order o INNER JOIN customer c
ON o.custnum = c.custnum;
```

Using outer joins

An outer join between two tables returns more information than a corresponding inner join. An outer join returns a result table that contains all the rows from one of the tables even if there is no row in the other table that satisfies the join condition.

OpenEdge SQL supports outer join operations from either the FROM clause or the WHERE clause. Note the following:

- In the FROM clause, specify the LEFT OUTER JOIN clause between two table names, followed by a search condition. The search condition can contain only the join condition between the
specified tables. This is the preferred method, as it is in keeping with the SQL standard. The syntax for a left outer join using the FROM clause is:

```
FROM table_ref LEFT OUTER JOIN table_ref ON search_condition
```

- In the WHERE clause, specify the outer join operator (+) after the column name of the table for which rows will not be preserved in the result table. Both sides of an outer join search condition in a WHERE clause must be simple column references. This syntax allows both left and right outer joins:

```
WHERE [table_name.]column (+) = [table_name.]column | WHERE [table_name.]column = [table_name.]column (+)
```

- Full (two-sided) outer joins are not supported.

**Left outer joins**

In a left outer join, the information from the table on the left is preserved: the result table contains all rows from the left table even if some rows do not have matching rows in the right table. Where there are no matching rows in the right table, SQL generates NULL values.

The following example depicts a join statement using the outer join operator in the WHERE clause:

```
SELECT Customer.Custnum, Customer.Name, Order.Ordernum, Order.Orderdate
FROM Customer, Order
WHERE Customer.CustNum = Order.CustNum (+) ;
```

The query requests information about all the customers and their orders. Even if there is not a corresponding row in the Order table for each row in the Customer table, NULL values are displayed for the Order.Ordernum and Order.Orderdate columns. This query produces the following results:

```
CustNum     Name      Ordernum   OrderDate     Lift
Tours       2006-02-11
1           Lift Tours 1   2006-03-17
1           Lift Tours 36  2006-05-01
1           Lift Tours 79  2006-06-22
. . .
5           Ace Tennis    NULL    NULL
7           Xtreme Surf   NULL    NULL
```

The following example uses the LEFT OUTER JOIN phrase in the FROM clause.

```
SELECT e.firstname, e.lastname, e.deptcode, d.deptname
FROM employee e LEFT OUTER JOIN department d
ON e.deptcode = d.deptcode
WHERE SUBSTR(e.firstname, 1, 1) = 'J'
ORDER BY d.deptcode, e.lastname;
```
This query produces the following results:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>DeptCode</th>
<th>DeptName</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Burton</td>
<td>200</td>
<td>Administration</td>
</tr>
<tr>
<td>Jenny</td>
<td>Morris</td>
<td>200</td>
<td>Administration</td>
</tr>
<tr>
<td>Jay</td>
<td>Ahern</td>
<td>300</td>
<td>Marketing</td>
</tr>
<tr>
<td>Justine</td>
<td>Smith</td>
<td>400</td>
<td>Sales</td>
</tr>
<tr>
<td>Jean</td>
<td>Brady</td>
<td>600</td>
<td>Development</td>
</tr>
</tbody>
</table>

Right outer joins

In a right outer join, the information from the table on the right is preserved. The result table contains all rows from the right table even if some rows do not have matching rows in the left table. Where there are no matching rows in the left table, SQL generates NULL values.

A right outer join retrieves all the rows from the right table even if there are not matches with the left table.

The following example offers an example of a right outer join.

```
SELECT e.firstname, e.lastname, d.deptcode, d.deptname
FROM employee e, department d
WHERE e.deptcode(+) = d.deptcode AND d.deptcode >= 500
ORDER BY d.deptcode, e.lastname;
```

This query produces the following results:

<table>
<thead>
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>DeptCode</th>
<th>DeptName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christine</td>
<td>Brown</td>
<td>500</td>
<td>Training</td>
</tr>
<tr>
<td>Brittney</td>
<td>Burton</td>
<td>500</td>
<td>Training</td>
</tr>
<tr>
<td>Larry</td>
<td>Dawsen</td>
<td>500</td>
<td>Training</td>
</tr>
<tr>
<td>Sabrina</td>
<td>Raymond</td>
<td>500</td>
<td>Training</td>
</tr>
<tr>
<td>Luke</td>
<td>Sanders</td>
<td>500</td>
<td>Training</td>
</tr>
<tr>
<td>Harold</td>
<td>Tedford</td>
<td>500</td>
<td>Training</td>
</tr>
<tr>
<td>Neil</td>
<td>Watson</td>
<td>500</td>
<td>Training</td>
</tr>
</tbody>
</table>

When working with right outer joins, remember the following:

- OpenEdge SQL has not yet implemented the RIGHT OUTER JOIN expression. Instead, it allows you to achieve the same effect of a right outer join by using the outer join operator (+).
- The outer join operator (+) appears on the left side of a right outer join.
- As you become fluent in SQL, you will notice that you can achieve the result of a right outer join by writing a left outer join and reversing the tables in the FROM and ON clauses. So, why do we have right outer joins? The reason is that right outer joins are a SQL standard with which OpenEdge SQL simply complies.
Using scalar subqueries

A scalar subquery is a parenthesized query expression that returns a single value (one row of a single column).

Scalar subqueries can be of the following types:

- Correlated scalar subquery: Returns a single value for each row of the correlated outer table.
  A scalar subquery can be a correlated to table(s) in the containing query block where it returns a single value for each outer row.

- Uncorrelated scalar subquery: Returns a single value to the containing query.

A correlated scalar subquery is used in the same way that a column is used, while an uncorrelated scalar subquery is used in the way that a constant value is used.

Scalar subqueries are not valid in the following cases:

- When used in ORDER BY and GROUP BY clauses
- When subqueries include UNION
- When subqueries include LOB data types

A scalar subquery uses the following syntax:

**Syntax**

```
scalar_subquery ::= subquery

subquery ::= (query_expression)
```

`scalar_subquery` selects exactly one column or expression in its select list.

The data type of a `scalar_subquery` is same as that of the column of the `query_expression` that it contains.

If the `scalar_subquery` returns more than one row, an error occurs stating that the subquery returns multiple rows. If the `scalar_subquery` returns 0 rows, then the `scalar_subquery` value is NULL.

A scalar subquery expression is valid in most syntax that calls for an expression (expr). Scalar subqueries can be used in the following components of SQL statements:

- `SELECT` statement
  - Select list
  - `WHERE` clause:
  - `ON` clause predicates
  - `HAVING` clause
  - `INSERT` statement
• **VALUES clause**

• **UPDATE statement**
  • **WHERE clause**

• **DELETE statement**
  • **WHERE clause**

The following examples illustrate the use of scalar subqueries in components of the **SELECT** statement:

```sql
SELECT d.DeptNo, d.DeptName, (SELECT COUNT (*) FROM pub.Employee e WHERE e.DeptNo = d.DeptNo) AS Num_emp_in_dept FROM Department d;
```

```sql
SELECT * FROM pub.Employee e
WHERE (SELECT MIN (EmpNum) + 2 FROM pub.Employee) = (SELECT MIN (EmpNum) FROM pub.Employee e, pub.Department d WHERE d.DeptCode = e.DeptCode AND d.DeptName='SALES');
```

```sql
SELECT * FROM Employee
WHERE Salary = (SELECT AVG (Salary) FROM Employee) + 1500;
```

```sql
SELECT e.EmpNum, d.DeptCode, d.DeptName FROM pub.Employee e
INNER JOIN pub.Department d
ON (SELECT DeptCode FROM pub.Employee where EmpNum = 1) = (SELECT DeptCode FROM pub.Department WHERE DeptName = 'Sales');
```

The following examples illustrate the use of scalar subqueries in components of the **INSERT** statement:

```sql
INSERT INTO pub.Employee
VALUES (1001, 'Scott', (SELECT AVG (Salary) FROM pub.Employee), 102);
```
The following examples illustrate the use of scalar subqueries in components of the DELETE statement:

```
DELETE FROM pub.Employee
WHERE DeptNum = (SELECT d.DeptNum FROM pub.Department d
WHERE DeptName = 'Sales')
```

The following examples illustrate the use of scalar subqueries in triggers and views:

```
CREATE VIEW V1 AS SELECT * FROM pub.Employee
WHERE (SELECT MIN (Salary) FROM pub.Employee) = (SELECT MIN (Salary)
FROM pub.Employee e, pub.Department d
WHERE d.DeptNum = e.DeptNum AND DeptName = 'Sales')
```

```
CREATE TRIGGER TRG_TST02902_TEST01 BEFORE INSERT ON tst_trg_01
BEGIN
  //Inserting Into tst1
  SQLIStatement insert_tst3 = new SQLIStatement("INSERT INTO tst_trg_03 VALUES
  ((SELECT AVG(c2) FROM pub.t1))");
  insert_tst3.execute();
END
```
The following examples illustrate the use of scalar subqueries as expressions:

```
SELECT (Salary + (SELECT AVG (Salary) FROM pub.Employee)) AS base_sal FROM pub.Employee;
```

```
SELECT (SELECT prod_name FROM pub.Products) || OrderId FROM pub.Orders
```

```
SELECT SUM (CustNum), COUNT (CustNum) FROM Pub.MtCustomer
WHERE CASE (SELECT MIN (CustNum) FROM pub.MtCustomer) WHEN (SELECT MIN (CustNum) FROM pub.MtOrder) THEN (SELECT MAX (CustNum) FROM pub.MtCustomer) ELSE (SELECT MIN (CustNum) FROM pub.MtCustomer) END = 2106;
```

```
SELECT ROUND ((SELECT MIN (CustNum) + 0.5 FROM pub.MtCustomer
WHERE Name = 'Lift Tours-OEDProducts'),0) FROM pub.MtCustomer;
```
OpenEdge SQL and Advanced Business Language Interoperability

This chapter addresses the interoperability of the Advanced Business Language (ABL) and OpenEdge SQL languages. ABL and OpenEdge SQL clients use their associated languages to interact with the OpenEdge Relational Database Management System. Understanding the relationship between ABL and OpenEdge SQL is critical to OpenEdge application development, and is described in the following sections.

For details, see the following topics:

• Managing Open Edge SQL and ABL database connections
• Establishing user accounts and assigning privileges
• ABL and OpenEdge SQL interaction in an OpenEdge application
Managing Open Edge SQL and ABL database connections

Determining database server requirements

To properly manage database connections for both OpenEdge SQL and ABL servers, you must first assess your startup parameter requirements. The OpenEdge SQL server and ABL server should be managed independently from each other. This section describes the process for determining server startup parameters.

To analyze database server requirements:

1. Determine if your database supports both SQL and ABL users.
2. Determine for each server type, SQL and ABL:
   - The maximum number of servers
   - The maximum number of servers per broker
   - The maximum number of remote clients per server
3. Calculate the maximum number of database users by using the following formula:
   \[-n = \text{Maximum SQL users} + \text{Maximum users} + \text{One per each additional broker} + \text{all other processes (APWs, BIWs, AIWs, PROMONs, online backups)}\]
4. Calculate the maximum number of database servers and brokers using the following formula:
   \[-M_n = (\text{Sum of all } M_{pb} \text{ values}) + \text{one per each additional broker}\]

Starting SQL and ABL brokers

Identify your brokers as either primary or secondary. The primary broker should always be the one with the highest value for \(-M_a\).

For the broker, invoke PROSERVE with the following:

- Set \(-M_a\) to the maximum number of users
- Set \(-M_{pb}\) to the maximum number of servers for this broker
- Set \(-\text{ServerType}\)

For example, use the following command to start an ABL (4GL) primary broker:

```
proserve Sports2002 -H localhost -S 6001 -ServerType 4GL -n 200 -Mn 7 -Mi 3
-Ma 3 -Mpb 4
```
Note: The database name specified by a client must be the same database name that was used to start the database server. The name of the OpenEdge programming language has changed from 4GL to Advanced Business Language (ABL). In some instances, such as the designation of server types, the programming language still uses the term "4GL."

proserve Sports2002

Starts the Sports2002 database.

-H localhost

Specifies the machine on which the server runs.

-S 6001

Specifies 6001 as the port number to be used when connecting to a broker process.

-ServerType

Specifies the server type.

-n 200

Limits the number of users to 200.

-Mn 7

Specifies 7 as the maximum number of servers.

-Mi 3

Limits the server to three clients.

-Mpb 4

Restricts the number of servers that can be spawned by the broker to 4.

For the secondary broker, invoke PROSERVE with the following:

• Use -m3 to start up this broker
• Use the -n and -Mn results from the formula above
• Set -Ma to the maximum number of SQL users
• Set -Mpb to the maximum number of servers for this SQL broker
• Set -ServerType to SQL

For example, use the following command to start an OpenEdge SQL secondary broker:

proserve Sports2001 -H localhost -S 6000 -ServerType SQL -n 200 -Mi 6 -Ma 3 -Mpb 2 -m3

proserve Sports2001

Starts the Sports2001 database
-H localhost
  Specifies the machine on which the server runs
-S 6000
  Specifies 6000 as the port number to be used when connecting to a broker process
-ServerType SQL
  Specifies the server type as SQL
-n 200
  Limits the number of users to 200
-Mi 6
  Limits the server to six clients
-Ma 3
  Specifies 3 as the maximum number of servers
-Mpb 2
  Restricts the number of servers that can be spawned by the broker to 2
-m3
  Identifies the SQL broker as the secondary broker

Establishing an encrypted connection

OpenEdge provides you with the ability to provide a Secure Socket Layer (SSL) connection for OpenEdge SQL and ABL. SSL provides an authenticated and encrypted peer-to-peer TCP/IP connection. You can establish the SSL connection by using the -ssl startup parameter. For more information on creating SSL brokers, see OpenEdge Getting Started: Core Business Services - Security and Auditing and OpenEdge Deployment: Startup Command and Parameter Reference.

For more information on managing brokers and startup parameters, see OpenEdge Deployment: Startup Command and Parameter Reference and OpenEdge Data Management: Database Administration.

Establishing user accounts and assigning privileges

Database security is maintained, in part, by requiring user authentication and assigning appropriate database privileges.
Using authentication

Logon validation is a mechanism that checks user identification and password at connection time using reference data stored in the OpenEdge RDBMS.

In the OpenEdge RDBMS, the reference table storing user identifications and passwords serves both OpenEdge SQL and ABL interfaces. Use the `CREATE USER` statement to enable authentication from the SQL interface. Use the OpenEdge Data Administration tool to enable authentication from the ABL interface. Authentication can be disabled from either interface.

In addition, ABL and SQL both support authentication to the operating system (OS) user accounts (in Windows or UNIX, wherever the OpenEdge process runs). If no users are defined in the `OER$USER` table accounts, ABL can either authenticate to the OS user accounts or connect (without authentication) using the OpenEdge default user ID; however, SQL must authenticate a user to one or the other set of user accounts in order to access a database.

The accounts used depend on the domain in which the user authenticates. A domain configured with the `OEUSERTABLE` authentication system supports authentication to the `OER$USER` table accounts; a domain configured with the `OSLOCAL` authentication system supports authentication to the OS user accounts. For more information, see *OpenEdge Getting Started: Identity Management*.

Creating, altering, or dropping a user via SQL is equivalent to creating, maintaining, or deleting a user with the OpenEdge Data Administration tool. The `OER$USER` table accounts updated for the OpenEdge SQL interface are updated for the ABL interface and vice versa.

In an environment where an OpenEdge RDBMS is accessed by applications using both ABL and SQL, the following conditions exist:

- **If no users have been created in the database** — All SQL users will be required to enter a username and password before they will be permitted access to the database.

- **If users have been created in the database** — Users accessing the database through the ABL interface are required to provide an identification and password, unless the default blank user ID access is allowed. Furthermore, users working from the interface by default have unlimited privileges. The DBA who controls access from the interface must place limitations on user privileges or restrict access to certain database objects. SQL users, by default, have no privileges and must be explicitly assigned.

Assigning privileges

Database administrators also control access to a database by assigning user privileges. SQL DBAs use the `GRANT` and `REVOKE` statements to authorize privileges for users who are working with the OpenEdge SQL interface. Privileges granted to users of the ABL interface are maintained separately using the OpenEdge Data Administration tool.

A DBA can grant specific privileges—such as selecting, updating, or deleting records—to individual users or to all users. When working through the SQL interface, all user actions against a database are prohibited unless explicitly authorized by the DBA. When users attempt to perform an action for which they do not have privileges, the OpenEdge SQL server generates an error message.

Conversely, all users working through the ABL interface have unlimited database privileges, unless those privileges are explicitly restricted by the DBA. Database privileges are defined for the interface by using the OpenEdge Data Administration tool.
DBAs must exercise caution when deciding which privileges should be assigned and to whom. For example, if a table or view is selectable by all users, the DBA can grant the ability to select data to the public. Otherwise, the privilege to select data should be granted to those individuals who have a need to do so. The same principle applies to other privileges, such as updating records or executing stored procedures.

For more information on using SQL `GRANT` and `REVOKE` statements and controlling user privileges, see Data Control Language and Transaction Behavior on page 145. For more information on database security, see OpenEdge Data Management: Database Administration.

### ABL and OpenEdge SQL interaction in an OpenEdge application

In an OpenEdge application, data management can be controlled by the programming language, the OpenEdge SQL language, or both. This section provides details on the relationship between the languages.

### Comparing ABL and OpenEdge SQL

ABL is a high-level procedural programming language, developed to enable you to build almost all aspects of an enterprise business application, from the user interface to database access and business logic.

**ABL**

ABL can create and maintain the databases of OpenEdge applications. You can use ABL to access database objects created with SQL, but only if they are located in the SQL schema called PUB and contain data types recognized by ABL.

**OpenEdge SQL**

OpenEdge SQL is a standards-based language used to create and maintain relational databases. OpenEdge SQL has the ability to retrieve, update, or delete data in an ABL database.

### Understanding OpenEdge SQL database structure

OpenEdge SQL and ABL relational schema differ in terminology. However, they both use the exact same underlying storage manager. This section describes SQL database objects and compares them to similar ABL objects.

The following figure offers an example of an SQL database table.
Comparing OpenEdge SQL and ABL database objects

Rows and columns

A SQL database table is made up of rows and columns. A row is a single occurrence of data in a table. A column characterizes one attribute of a row of data. ABL uses the term "field" instead of "column" and "record" instead of "row."

Tables

A SQL table is a group of related data composed of rows and columns. The term "table" is used similarly in ABL to describe a collection of fields and records.

Schemas

The term schema can be used three different ways:

- When discussing a database created with OpenEdge SQL, a schema is defined as a collection of related database objects, such as tables or views. A SQL database can contain several schemas.
- When referring to a database created with the ABL, schema is defined as the area in which all system and user information are stored. An ABL database contains only one schema area, referred to as the PUB (short for PUBLIC) schema.
- While working from the OpenEdge SQL client, you can ensure you are working in the correct schema by using the SET SCHEMA command. For more information on using SET SCHEMA, see OpenEdge Data Management: SQL Reference.

Naming objects for OpenEdge SQL and ABL databases

Naming database objects correctly will ensure the SQL client can easily access an object created with ABL. There are subtle differences to the naming conventions for both languages.
Naming conventions for ABL objects

The following rules apply to ABL data field naming conventions:

- Names can be up to 32 characters long and can consist of alphabetic characters (A-Z or a-z) and/or digits (0-9).
- They can be written in any combination of uppercase or lowercase letters and digits (0-9).
- They must begin with an uppercase letter or lowercase letter.
- They can contain the pound sign (#), dollar sign ($), percent sign (%), ampersand (&), hyphen (-), and underscore (_).
- They are not case sensitive.
- They cannot use ABL keywords.

Note: Although the ABL itself has many hyphenated keywords, Progress Software Corporation recommends that you do not use hyphens in database table and field names, because the SQL standard does not allow hyphens in table and column names. Hyphenated identifiers can be used if delimited in SQL.

Naming conventions for OpenEdge SQL identifiers

SQL database object names are referred to as identifiers. The two types of SQL identifiers are:

- Conventional identifiers
- Delimited identifiers that are enclosed in double quotation marks

Conventional identifiers

Conventional SQL identifiers must:

- Be limited to 32 characters
- Begin with an uppercase letter or lowercase letter
- Contain only letters (A-Z), digits (0-9), or the underscore character (_)
- Not contain a hyphen
- Not be reserved words, such as CREATE or DROP
- Only use ASCII characters only

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. To include a quotation mark character in a delimited identifier, precede it with another quotation mark.
Note: ABL database objects must be named in a manner consistent with SQL delimited identifiers in order to be accessed by an OpenEdge SQL client. For example, an ABL table with a name that includes a special character should also use quotation marks (for example, "Orders&Deliveries") in order to be accessible from the OpenEdge SQL interface.

Working with data type compatibility

While all ABL data types are supported by SQL equivalents, the following table illustrates the compatible data types of the two languages.

Table 6: ABL and OpenEdge SQL data types

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>OpenEdge SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>ARRAY, VARARRAY</td>
</tr>
<tr>
<td>BLOB</td>
<td>LVARBINARY, BLOB</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>VARCHAR, CHAR</td>
</tr>
<tr>
<td>CLOB</td>
<td>CLOB, LVARCHAR</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>DATETIME</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>DATETIME-TZ</td>
<td>TIMESTAMP WITH TIME ZONE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>DECIMAL or NUMERIC</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>BIT</td>
</tr>
<tr>
<td>RAW</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>RECID</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

For information on ABL data types, see OpenEdge Getting Started: ABL Essentials. For more information on OpenEdge SQL data types, see OpenEdge Data Management: SQL Reference.

Working with SQL column widths

An ABL database can contain fields, also referred to as columns, of variable length. However, SQL CREATE TABLE statements specify the maximum width of each column in a table. ABL programs have the ability to insert data whose length exceeds the Data Dictionary value of SQLWidth. SQL applications will not be able to read a row if a column contains data greater than the SQLWidth value defined in the Data Dictionary.

Two tools enable you to compensate for column width discrepancies:
The -checkwidth startup parameter

Using the -checkwidth startup parameter

OpenEdge ABL programmers can override Data Dictionary column definitions in ABL programs. Therefore, a SQL application cannot read a record if a column contains data greater than the SQLWidth value defined in the Data Dictionary. An attempt to retrieve a column that exceeds the SQLWidth definition generates an error message, and the attempt to access the record fails.

Use the -checkwidth startup parameter to specify whether ABL compares CHARACTER, DECIMAL, and RAW field data against the metaschema _width field value before updating a database record. The _width field value specifies the maximum width of the data allowed in a field.

The syntax for the -checkwidth startup parameter is:

**Syntax**

```
-checkwidth n
```

The -checkwidth startup parameter can be employed in the following modes:

- **0** — Ignore the _width field value and store the data. This is the default mode.
- **1** — Store the data and generate a warning message if the data exceeds the size specified in the _width field.
- **2** — Do not store data that exceeds the size specified in the _width field and generate an error. Specify this mode if you want the ABL to behave like SQL.

For more information on the -checkwidth startup parameter, see OpenEdge Deployment: Startup Command and Parameter Reference.

Using the DBTool utility

The DBTool utility allows users to identify when the size of column data in the database exceeds the Data Dictionary definition and therefore the SQLWidth value. The DBTool utility addresses this situation because it allows for the fast updating of Data Dictionary SQLWidth definitions.

The following error message is reported to a SQL application when the SQLWidth for a column exceeds the Data Dictionary SQLWidth definition:

```
Column column in table table has value exceeding its max length or precision (7864)
```

The syntax for DBTool is:

**Syntax**

```
dbtool dbname
```
To access DBTool from the command line:

1. Type `dbtool` and the database name and press Enter.

The DBTool option menu appears:

2. Select an option from the menu and press Enter.

The following table describes the options available in the DBTool option menu.

### Table 7: DBTool option menu

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Finds the maximum field sizes and reports them.</td>
</tr>
<tr>
<td>2</td>
<td>Finds the maximum field sizes and updates their widths.</td>
</tr>
<tr>
<td>3</td>
<td>Validates the schema versioning of the records after the records are updated by DBTool.</td>
</tr>
<tr>
<td>4</td>
<td>Validates the schema versioning before and after the records are updated in DBTool.</td>
</tr>
<tr>
<td>5</td>
<td>Validates db keys while scanning database blocks.</td>
</tr>
<tr>
<td>6</td>
<td>Scans records for indications of possible corruption.</td>
</tr>
<tr>
<td>7</td>
<td>Checks for inconsistencies in the database schema. This option identifies errors in schema records for word indexes. If an error is detected, this option reports the index number and recommends rebuilding that index.</td>
</tr>
<tr>
<td>9</td>
<td>Enables or disables file logging.</td>
</tr>
<tr>
<td>Q</td>
<td>Quits the DBTool utility.</td>
</tr>
</tbody>
</table>

For more information on the DBTool utility, see *OpenEdge Data Management: Database Administration*. 
Working with triggers

The following limitations apply in the use of SQL and ABL triggers:

- SQL clients and servers do not execute -based triggers.
- ABL clients and servers do not execute SQL Java triggers.
- If both SQL and ABL access are needed, you must have two triggers.
- ABL does not observe SQL constraints.

ABL possesses a rich set of trigger mechanisms. ABL triggers are procedures that are written by the user, and they fire when record-oriented events such as creating, deleting, writing, or finding a record occur. ABL triggers are associated with their corresponding database records using the Data Dictionary and execute in the context of an AppServer or an ABL client.

SQL also possesses a rich set of trigger mechanisms. SQL triggers are Java procedures written by the SQL database administrator, and they fire when record-oriented or statement-oriented events, such as inserting, deleting, or updating data, occur. SQL triggers are defined for their corresponding database tables using the `CREATE TRIGGER` command. SQL triggers execute in the SQL server.

The SQL and ABL trigger mechanisms are parallel but separate. The ABL does not execute SQL triggers, and SQL does not execute ABL triggers. This is because the two types of triggers require different execution engines: SQL requires the SQL engine, and the ABL requires the ABL engine.

Usually, an ABL trigger will contain significant database integrity enforcement logic, or perhaps enforcement of business logic rules. If a database can also be updated by SQL users using SQL statements, the database administrator will want to apply the same enforcement rules when SQL updates occur. To do so, the data-specific logic should be replicated from the ABL trigger to a SQL Java trigger. That new trigger can then be added to the database, using the `CREATE TRIGGER` statement. SQL will then use that trigger to enforce the defined logical rules and actions.

For more information on using ABL triggers, see *OpenEdge Getting Started: ABL Essentials. For more information on SQL triggers, Stored Procedures and Triggers* on page 181.

Working with locking behavior and isolation levels

Consistency and concurrency are maintained with record locks. Because SQL relies on table and record locks to carry out the intent of transaction isolation levels, table locks were implemented in the database engine. Both ABL and SQL clients encounter table locks while executing transactions.

A locking conflict occurs when two transactions request the same resource at the same time. The SQL client will wait for a resource for a specified time before giving up, at which point an error would be generated and the operation would need to be retried.

The default wait time is five seconds, but can be modified to a duration that meets your application's needs. The SQL lock wait time-out value can be set by using the startup parameter `-SQL_LOCKWAIT_TIMEOUT` along with the `proserve` command. The specified time-out value must be a minimum of five seconds.

For ABL clients, there is a lock wait time-out parameter (`-lkwtmo`) that specifies how long a client should wait for a resource. The current default value is 30 minutes. If a SQL client has a lock on a table for which the ABL client also requested a lock, the SQL client times out and gives up waiting long before the ABL client.
A `SELECT` statement can fail if some records of the selected tables are locked by other transactions. The `SELECT` transaction is not able to continue until the record locks are released by other transactions. The `READPAST` lock hint causes a transaction to skip rows locked by other transactions. The skipped rows do not appear in the result set, and a warning is returned to the client.

The following conditions apply to the `READPAST` locking hint:

- Applies only to the `SELECT` statement
- Applies only to transactions operating at `READ COMMITTED` isolation
- Reads only past row-level locks
- Only specified in the main `SELECT` statement but not in the subquery `SELECT` statement in the `search_condition` of the `WHERE` clause

As an option to the `READPAST` locking hint, you may use the `WITH NOLOCK` option as part of the table reference of the `SELECT` statement. While the `READPAST` hint will skip locked records, the `WITH NOLOCK` option allows a dirty read to be performed. For more information on record locks, the `READPAST` locking hint, and the `WITH NOLOCK` option, see Data Control Language and Transaction Behavior on page 145.
To properly maintain a database, it is important to understand the concepts of transactions, locks, and their proper management using OpenEdge SQL, as described in the following sections.

For details, see the following topics:

- Working with transaction control
- Transactions and isolation levels
- Understanding transactions and locking
- Enhancing performance with locking hints
- Monitoring locking and database performance
- Online schema changes
- Authorized data truncation
- Autonomous Schema Update

**Working with transaction control**

Applications execute a SQL statement or group of logically related SQL statements to perform a database transaction. The SQL statement or statements add, delete, or modify data in the database.
Transactions are atomic and durable. To be considered atomic, a transaction must successfully complete all of its statements; otherwise, none of the statements execute. To be considered durable, a transaction's changes to a database must be permanent.

Complete a transaction by using either the COMMIT or ROLLBACK statements. COMMIT statements make permanent the changes to the database created by a transaction. ROLLBACK restores the database to the state it was in before the transaction was performed.

**COMMIT statement**

You can complete a transaction in an application by using the COMMIT statement. When a COMMIT statement is executed, all the changes made to the database by the transaction are made permanent.

Depending on the isolation level of the transaction, changes made by one transaction might not be visible to other transactions before the transaction is committed. OpenEdge SQL’s default behavior is to make database changes visible only after the transaction has been committed.

The COMMIT statement uses the following syntax:

```sql
COMMIT [ WORK ];
```

The following example shows a COMMIT statement:

```sql
COMMIT WORK ;
```

**Note:** The COMMIT statement does not affect the contents of the host variables or the flow of control in the program.

For more on the COMMIT statement, see OpenEdge Data Management: SQL Reference.

**ROLLBACK statement**

The ROLLBACK statement undoes all the changes made to the database within a transaction. The ROLLBACK statement uses the following syntax:

```sql
ROLLBACK [ WORK ];
```

The following example shows how to use the ROLLBACK statement:

```sql
ROLLBACK WORK ;
```
Transactions and isolation levels

SQL defines how a database is expected to behave in different situations. These situations are defined as transaction isolation levels. The isolation levels are defined by possible phenomena that might exist in the database at the four possible levels. Before you can understand isolation levels, you must first understand what these phenomena are.

The following phenomena are used to define isolation levels.

Dirty read

A dirty read occurs when one user is updating or inserting a record while a different user is reading it, but the work is not yet committed to the database, as shown in the following example.

User A executes:
INSERT INTO State (state, state_name, region)
VALUES ('ME', 'Maine', 'Northeast');
User B executes: SELECT * FROM State;
User B sees: state 'ME'
User A executes: ROLLBACK WORK; User B has seen data that really did not exist.

Nonrepeatable read

A nonrepeatable read occurs when one user is repeating a read operation on the same records but has updated values, as shown in the following example.

User A executes: SELECT * FROM State;
User B executes:
UPDATE State
SET state_name = 'Arkansas'
WHERE state = 'AK';
COMMIT WORK;
SELECT * FROM pub.State
User A re-executes:
SELECT * FROM State;
User A has now updated records in the result set.
Phantom read

A phantom read occurs when one user is repeating a read operation on the same records, but has new records in the results set, as shown in the following example.

<table>
<thead>
<tr>
<th>User A executes:</th>
<th>User B executes:</th>
<th>User A re-executes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM State;</td>
<td>INSERT INTO pub.State (state, state_name, region) VALUES ('CT', 'Connecticut', 'Northeast'); COMMIT WORK;</td>
<td>SELECT * FROM pub.State;</td>
</tr>
</tbody>
</table>

User 1 has new records in the results set.

Setting isolation levels

The degree to which one transaction can interfere with other transactions by accessing the same rows concurrently is determined by setting the transaction isolation level in the program.

This is the syntax for the `SET TRANSACTION ISOLATION LEVEL` statement:

**Syntax**

```
SET TRANSACTION ISOLATION LEVEL isolation_level_name;
```

<table>
<thead>
<tr>
<th>isolation_level_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
</tr>
<tr>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
</tr>
</tbody>
</table>

SQL defines isolation levels in terms of the inconsistencies they allow:

**READ UNCOMMITTED**

Also called a "dirty read." When this isolation level is used, a transaction can read uncommitted data that later can be rolled back. A transaction that uses this isolation level can only fetch data but cannot update, delete, or insert data.

**READ COMMITTED**

With this isolation level dirty reads are not possible, but if the same row is read repeatedly 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 way that they now satisfy the search criteria.

**SERIALIZABLE**

This isolation level guarantees that none of the above happens. In addition, it guarantees that transactions that use this level will be completely isolated from other transactions.

The following table identifies which phenomena are either permitted or prevented by each isolation level.

**Table 8: Transaction isolation levels**

<table>
<thead>
<tr>
<th>Isolation</th>
<th>Dirty read</th>
<th>Nonrepeatable read</th>
<th>Phantom read</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Permitted</td>
<td>Permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>Prevented</td>
<td>Permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>Prevented</td>
<td>Prevented</td>
<td>Permitted</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>Prevented</td>
<td>Prevented</td>
<td>Prevented</td>
</tr>
</tbody>
</table>

**Note:** The isolation levels are ordered according to the phenomena they either permit or prevent. The first one, **READ UNCOMMITTED**, is the isolation level providing the highest level of concurrency but with the lowest level of consistency. Each subsequent level provides at least as much data consistency as the one before but will result in less concurrency. As a general rule, the more data consistency that is provided by the isolation level used from an application, the less concurrency is allowed between this application and other applications connected to the same database.

**Understanding transactions and locking**

Locks are made to ensure consistency in a database. Consistency provides you with successive, reliable, and uniform results of your database transactions. This section describes the locks that are used at each of the four transaction isolation levels.

**Lock modes**

Lock modes are prioritized tokens in a queue that indicate what action is being taken in the transaction process. For instance, intent to update a record requires a different mode of lock than to actually update it. Lock modes facilitate concurrency and provide consistency. They indicate intent and are used to stage lock requests. Lock requests are generated as a result of the execution of a transaction.
OpenEdge provides the following lock modes:

- **NO-LOCK (NL)** — You have no intention of performing an update, and accuracy of the resulting set of data is not important.

- **INTENT SHARE (IS)** — You intend to share-lock objects at the next lower level of granularity for this object (table). That is, you intend to get share locks on the rows of this table.

- **INTENT EXCLUSIVE (IX)** — You intend to exclusive-lock objects at the next lower level of granularity for this object (table). That is, you intend to get exclusive locks on the rows of this table.

- **SHARED (S)** — You want a share-lock on the object. Getting a share-lock on an object means that you implicitly get a share-lock on all of the objects that this object contains, that is, all of the rows for this table.

- **SHARED WITH INTENT EXCLUSIVE (SIX)** — You want a share-lock on the table so no one else can modify, delete, or add rows except you.

- **EXCLUSIVE (X)** — You want an exclusive-lock on the object. Getting an exclusive lock on an object means that you implicitly get an exclusive lock on all of the objects that this object contains, that is, all of the rows for this table.

### How lock levels and lock modes interact

The following tables describe how the SQL OpenEdge Engine uses locking to produce a desired transaction behavior. The tables identify the requested lock strength based on the transaction isolation level in effect for a given transaction.

**Table 9: Insert, update, or delete record operations**

<table>
<thead>
<tr>
<th>Isolation</th>
<th>Info schema lock</th>
<th>Table lock</th>
<th>Record lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>S</td>
<td>NL</td>
<td>NL</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>S</td>
<td>IX</td>
<td>X</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>S</td>
<td>IX</td>
<td>X</td>
</tr>
<tr>
<td>Serializable</td>
<td>S</td>
<td>SIX</td>
<td>X</td>
</tr>
</tbody>
</table>
Table 10: Fetch or select record operations

<table>
<thead>
<tr>
<th>Isolation</th>
<th>Info schema lock</th>
<th>Table lock</th>
<th>Record lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>S</td>
<td>NL</td>
<td>NL</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>S</td>
<td>IS</td>
<td>S</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>S</td>
<td>IS</td>
<td>S</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

There are no table or record locks acquired when the transaction isolation level is `READ UNCOMMITTED`.

In the `READ UNCOMMITTED` transaction isolation level you maximize concurrency, but you might also read dirty data.

The primary difference between the `READ COMMITTED` and `REPEATABLE READ` transaction isolation levels is that while in `REPEATABLE READ`, individual record locks are held for the duration of the transaction. For example, if your fetch criteria include all companies in the state of Idaho, each record in the result set will remain locked until all of the records meeting the criteria have been read. In the `READ COMMITTED` transaction isolation level, the record locks are released once the record has been read.

In the `SERIALIZABLE` transaction isolation level, a share lock on a table is held for the duration of the transaction, preventing any other transaction from updating the table. Any SQL operation that modifies the information schema is upgraded to `SERIALIZABLE`, regardless of the user's current transaction setting.

Understanding lock acquisition

Knowing which objects get locked and when goes a long way towards helping you develop applications that are more robust and predictable. SQL uses the transaction isolation level exclusively to determine what lock mode is applied to which objects. Understanding how this translates into object locks and lock modes is key to communicating your application's intentions to the SQL engine.

The strongest locks are held when the transaction isolation level is `SERIALIZABLE`, and the weakest locks are held when the transaction isolation level is `READ UNCOMMITTED`. This also translates into application concurrency—the higher the transaction isolation level, the less concurrent your application will be.

Information schema locks

Every operation performed by the OpenEdge SQL Engine operates inside a transaction.
For each transaction, an information schema share-lock is acquired at the beginning of the transaction and released at the end of the transaction. This is true whether a transaction is committed successfully or terminated abnormally. Acquiring the information schema share-lock protects the information schema from being altered while the transaction is active.

During the life of an active connection, many transactions can be performed. The first transaction begins upon connection to the SQL engine and is used to read the information schema. Once the information schema has been read, the transaction ends. Each successive operation will then begin and end a transaction requiring, at a minimum, a share-lock on the information schema.

While the connection is quiet, there is no active transaction and therefore no lock held on the information schema. If an operation is being performed that will modify the information schema, an exclusive lock on the information schema will be requested. For the exclusive lock on the information schema to be granted, there can be no other active transactions in the database. Once granted, the information schema lock is upgraded from a share to an exclusive lock. While this transaction is active, the exclusive lock prohibits other transactions.

The lock on information schema supersedes all locks on tables and records via transaction isolation level settings for data manipulation operations.

Table and record locks

To get a record lock of sufficient strength for an operation, you must first have a table lock of sufficient strength. Regardless of the current transaction isolation level, if the application's intent is to perform an operation other than a fetch, the lock mode is strengthened for the operation.

With the exception of the \texttt{READ UNCOMMITTED} isolation level, you are not prohibited from creating or updating records based on the transaction isolation level. It is the responsibility of the RDBMS to provide sufficient lock escalation when an operation is being performed that requires lock upgrades. Note that you are prohibited from creating or updating records when the isolation level is \texttt{READ UNCOMMITTED}.

Enhancing performance with locking hints

A range of table-level locking hints can be specified using the \texttt{SELECT}, \texttt{INSERT}, \texttt{UPDATE}, and \texttt{DELETE} statements to direct the OpenEdge SQL Engine to the type of locks to be used. Table-level locking hints can be used 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 \texttt{READPAST} locking hint

The \texttt{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 \texttt{READPAST} lock hint applies only to transactions operating at \texttt{READ COMMITTED} isolation level and will read only past row-level locks. This only applies to the \texttt{SELECT} statement.

The locking hint clause, such as \texttt{READPAST}, can only be specified in the main \texttt{SELECT} statement, but not in the subquery \texttt{SELECT} statement in the search condition of the \texttt{WHERE} clause.
The `SELECT` statement uses the following syntax:

```sql
SELECT column_list FROM table_list [WITH(NOLOCK)] [ WHERE search_condition ] [ GROUP BY grouping_condition ] [ HAVING search_condition ] [ ORDER BY ordering_condition ] [ WITH locking_hints ] [ FOR UPDATE update_condition ];
```

The **WITH NOLOCK** option, when used as part of the table reference, allows the `SELECT` statement to perform a dirty read.

The **WITH** phrase uses the following syntax:

```sql
WITH ( READPAST [ NOWAIT ] [ WAIT timeout ] )
```

**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 gets the required lock or until it times out waiting for the lock.

**WAIT timeout**

Overrides the default lock-wait time-out. The time-out value is in seconds and can be given a 0 or any positive number.

The `SELECT` statements in the following examples illustrate the use of the **READPAST** locking hint.

```sql
SELECT * FROM Customer WHERE "CustNum" < 100 ORDER BY "CustNum" FOR UPDATE
WITH (READPAST NOWAIT);
```

```sql
SELECT * FROM Customer WHERE "CustNum" < 100 ORDER BY "CustNum" FOR UPDATE
WITH (READPAST WAIT 1);
```

**Monitoring locking and database performance**

OpenEdge offers two tools for analyzing your database’s performance as it relates to locking:

- The **Progress Monitor (PROMON)** utility helps you monitor database activity and performance. **PROMON** also provides advanced options (called R&D options) for in-depth monitoring of database activity and performance.

- Virtual system tables give ABL and OpenEdge SQL applications access to the same type of database information that you collect with the **PROMON** utility. **Virtual system tables (VSTs)** enable an application to examine the status of a database and monitor its performance.
the database broker running, ABL and OpenEdge SQL applications can call a VST and retrieve the specified information as run-time data. The following virtual system tables relate to locking:

- **Lock Table Activity (ActLock)** — Displays lock-table activity, including the number of share, exclusive, upgrade, Rec Get, and redundant requests; the number of exclusive, Rec Get, share, and upgrade grants; the number of exclusive, Rec Get, share, and upgrade waits; the number of downgrades, transactions committed, cancelled requests, and database up time.

- **Lock Table Status File (Lock)** — Displays the status of the lock table, including the user number, the user name, lock type, record ID, number, flags, and chain.

- **Lock Request File (LockReq)** — Displays information about lock requests, including user name and number, record locks and waits, information schema locks and waits, and transaction locks and waits.

- **Record Locking Table File (UserLock)** — Reports the first 512 lock entries for a particular user with one request to the lock subsystem. It is a "snapshot" picture of each user per "_user-lock" record and not a changing picture as is each "_lock."

For information on the PROMON utility and virtual system tables, see *OpenEdge Data Management: Database Administration*.

## Online schema changes

The following commands can be executed against an online database:

- **ALTER TABLE ADD COLUMN**

**Note:** The `ALTER TABLE` command for adding a LOB column is also an online operation. The `ALTER TABLE` command for dropping a LOB column is an offline operation. For further information, see *OpenEdge Data Management: SQL Reference*.

- **ALTER TABLE ALTER COLUMN**

- **ALTER TABLE ADD PARTITION**

- **ALTER TABLE ALTER INDEX**

- **ALTER TABLE DROP PARTITION**

- **ALTER TABLE PREPARE FOR SPLIT**

**Note:** The `ALTER TABLE` command for splitting a composite partition (SPLIT PRO_INITIAL) and splitting a RANGE partition is also an online operation. For further information, see *OpenEdge Data Management: SQL Reference*.

- **CREATE INDEX**

**Note:** Inactive indexes can be created online. Local and global (inactive) indexes are also created online. Active indexes can only be created online if they are created on a newly created table within the same transaction as the `CREATE TABLE`. For further information, see *OpenEdge Data Management: SQL Reference*. 
Authorized data truncation

ABL applications at times produce data that is larger than the size defined by SQL, since the ABL character data type (although has database record size limitations) does not provide a maximum defined size. OpenEdge SQL now allows database administrators (DBAs) to authorize truncation of data, so that the selected data fits the defined column size and the part of the value larger than the defined size of the column is truncated.
OpenEdge SQL uses a common, uniform model of data truncation. It operates on the truncated data by identifying the database as if it contains only the data truncated to its defined size and not the large data that may exist in a table. Internally, it may temporarily identify the large data in its full size, but the effective (logical) use of the data is in its truncated form. This simple data model of truncated data provides consistency and understandability across the many ways in which SQL operates on data: using indexes, performing direct table scans, evaluating expressions, evaluating predicates, and other operations.

**Note:** The data truncation operation affects the output result set, as it contains the truncated data. Only columns of type VARCHAR are affected by an authorized data truncation.

A DBA can authorize truncation of data in the following ways:

- **The SQLTruncateTooLarge can be set to ON to truncate data exceeding the column size.** If the parameter is set to OFF, no data is truncated and SQL displays an error. SQL also displays an error if the SQLTruncateTooLarge parameter is not specified at all in case of data exceeding the column size. The default value for the SQLTruncateTooLarge parameter is OFF.

  ```
  proserve -db <dbname> -S <port-number> -SQLTruncateTooLarge <on/off>
  ```

- **Embedding a data truncation option in the connection URL** - SQL has an optional parameter truncateTooLarge that can be embedded in the connection URL to authorize truncation of data. With its value set to ON, the parameter allows truncating data exceeding the column size. If the value is set to OFF, no data is truncated and an error is displayed. An error is also displayed if the truncateTooLarge parameter is not specified at all. The default value for the truncateTooLarge parameter is OFF. Not specifying the parameter is equivalent to setting its value to OFF.

  The value of truncateTooLarge parameter is connection specific and is remembered by SQL only during the connection session. The log setting of a connection-specific truncation overwrites the authorized data truncation setting at the database level.

  ```
  jdbc:datadirect:openedge://<ip>:<port>;databaseName=<dbname>;truncateTooLarge=<on/off>;
  ```

**Note:** The ODBC drivers have been updated to support the new truncateTooLarge parameter. See the ODBC driver changes section for more details.
An authorized data truncation operation in a CREATE INDEX statement fails in the following cases:

- One of the index components is of type VARCHAR
- At least one row in the table on which the index is being created has a column value exceeding the defined maximum size of the column

Logging

When a DBA-authorized data truncation occurs and if logging is enabled for the data truncation, the table name, column name, rowid, partition ID, and tenant ID of any row containing a truncated data value is logged.

Note: Enabling logging for data truncation may cause the performance of SQL server process to degrade. If there is a large number of data truncation, large amounts of information is written to the log file, which may cause SQL server process to slow down.

Writing to database log file

SQL always maintains a connection-specific counter to count the number of data truncation events occurred during a connection. This number of data truncation events is always logged to the database log file when a connection is disconnected. The counter is initialized to zero when a connection is established with the SQL server.

At the time of disconnect, if value of this counter is zero, i.e., if there was no data truncation, no information related to the data truncation is written to the database log file. At the time of disconnect, if value of this counter is zero, i.e., if there was no data truncation, no information related to the data truncation is written to the database log file.

Message to the database log file is written in the following format:

\<date-time> \P-<process-id> \T-<thread-id> I SQLSRV2 <server-id> : <counter> authorized data truncation action(s) performed.

Logging instances

Data truncation can be enabled for the following logging instances:

- **Server-wide logging** - SQL logs the DBA-authorized data truncation event for all clients to the log files: SQL\_data\_truncate\_server\_<process-id>\_<timestamp>\_A.log and SQL\_data\_truncate\_server\_<process-id>\_<timestamp>\_B.log. SQL begins to write to the log file SQL\_data\_truncate\_server\_<process-id>\_<timestamp>\_A.log and if the size of this file exceeds 500 MB, the log information is continued to be written in the file SQL\_data\_truncate\_server\_<process-id>\_<timestamp>\_B.log. If this file also exhausts, SQL rotates to the previous file and logs information in the file SQL\_data\_truncate\_server\_<process-id>\_<timestamp>\_A.log, overwriting the previously written information. The logging process continues in this rotation format. A log file is maintained across processes, which means, only one log file is maintained for a SQL server process. Information about all clients is logged in this file. For partitioned tables, the log information is written in the following format. The log file has a header and a
body section. The body section logs details of the truncated column, table row, and partition ID.

Data truncation logging started at <date-time>

P-<process-id> T-<thread-id> <server-id> <table-name> <column-name> pro_partn_row_id: < pro_partn_row_id> : authorized data truncation action(s) performed.

• For regular tables, the log information is written in the following format:

Data truncation logging started at <date-time>

P-<process-id> T-<thread-id> <server-id> <table-name> <column-name> row_id: < row-id> : authorized data truncation action(s) performed.

Switching on server-wide logging
A DBA can turn server-wide logging on or off for data truncation by executing the following command in SQL:

SET PRO_DATA_TRUNCATE_SERVER LOG ON/OFF

If logging is enabled for the server, the log data truncation event for all the clients is written to the process-wide log file. If logging is not enabled for the server, no information about the data truncation event is logged. By default, the server-wide logging is turned OFF (disabled).

• Connection Specific Logging - To maintain connection-specific log information, SQL logs the DBA-authorized data truncation event for the current client to the client-specific log file:

SQL_data_truncate_connection_<connection-uid>_<timestamp> A.log and SQL_data_truncate_connection_<connection-uid>_<timestamp> B.log. SQL begins to write to the log file SQL_data_truncate_connection_<connection-uid>_<timestamp> A.log and if the size of this file exceeds 500 MB, the log information is continued to be written in the file SQL_data_truncate_connection_<connection-uid>_<timestamp> B.log. If this file also exhausts, SQL rotates to the previous file and logs information in the file SQL_data_truncate_connection_<connection-uid>_<timestamp> A.log, overwriting the previously written information. The logging process continues in this rotation format.

These log files are maintained across the connection with a different log file for each client. A log file contains information only for its corresponding client.

For partitioned tables, the log information is written in the following format. The log file has a header and a body section. The body section logs details of the truncated column, table row, and partition ID.

Data truncation logging started at <date-time>

P-<process-id> T-<thread-id> <server-id> <table-name> <column-name> pro_partn_row_id: < pro_partn_row_id> : authorized data truncation action(s) performed.

For regular tables, the log information is written in the following format:

Data truncation logging started at <date-time>

P-<process-id> T-<thread-id> <server-id> <table-name> <column-name> row_id: < row-id> : authorized data truncation action(s) performed.

Switching on connection-specific logging - A DBA can turn connection-specific logging on or off for data truncation by executing the following command in SQL:

SET PRO_DATA_TRUNCATE_CONNECTION LOG ON/OFF
If logging is enabled for the client, the log data truncation event is written for the current client to the client specific log file. If logging is not enabled for the connection, no information about the data truncation event is logged. By default, the connection-specific logging is turned OFF (disabled).

If the above command is executed from a client, logging is enabled only for that client and information of only that client is logged to the client-specific log file. To enable logging for individual clients, the command must be executed for each client.

There is no dependency between a server-wide log file and client-specific log file. If both are enabled, SQL writes the log information for all the clients to the server-wide log file and information of each client to the client-specific log file when any data is truncated.

If both are disabled, no log information is written to any of the files even if a data truncation event occurs. If only server-wide logging is enabled, SQL writes the log information of all the clients to the server-wide log file. No log information is written to the connection-specific log file. If only connection-specific logging is enabled, SQL writes the log information of the client to the client-specific log file. No log information is written to the server-wide log file.

For example, if there are five clients and both server-wide and connection-specific log files are enabled, five connection-specific files and a server-wide log file (a total of six log files) are generated.

**SQL Utility update to prevent data loss**

The data truncation feature is explicitly turned off in the SQL server for an SQL utility client, irrespective of the values of the of the parameters `SQLTruncateTooLarge` and `truncateTooLarge`, and the SQL Server always returns an error to SQL Dump for a too-large column rather than truncated data. This is to prevent data loss during data dump.

**Autonomous Schema Update**

Autonomous Schema Update (ASU) helps resolve the SQL width problem by updating the SQL column width in schema automatically when Authorized Data Truncation (ADT) occurs for data in a column. So, the first time a SQL query is executed, with ASU and ADT enabled, truncated data is returned and the column width is adjusted to accommodate the complete data. Users should execute the query again to get the complete data. The ASU feature is turned off by default and the user has to authorize this feature. To enable the feature, a server startup parameter `-SQLWidthUpdate` is added. This parameter takes the values as ON and OFF. The default value is OFF. To use this feature, provide the value of the `-SQLWidthUpdate` parameter as ON during startup. Once the value of the parameter is set, it is remembered for the lifetime of the server and for all connections. If the value of the `-SQLWidthUpdate` parameter is set to ON during startup, the Authorized Data Truncation (ADT) feature is also enabled irrespective of whether ADT was enabled during startup unless specified otherwise using connection URL. If for a specific connection, the ADT feature is disabled using connection URL and the ASU feature is enabled during startup, the ADT feature remains turned off for that specific connection.

**Note:** The ASU parameter can be provided only during server startup and not in the client connection URL.
Performing Multi-database Queries

OpenEdge SQL provides you with the ability to perform multi-database queries. Using a `SELECT` statement, you can retrieve information from multiple databases and view the retrieved information as a single result set.

For details, see the following topics:

- Multi-database query overview
- Connecting to multiple databases
- An example of a multi-database query

Multi-database query overview

Before you can retrieve information from multiple databases, it is important that you understand some basics of how a multi-database query works.

The process of multi-database queries

A multi-database query is performed when a `SELECT` statement retrieves information from multiple databases simultaneously and the retrieved information is presented in a single result set. The process for performing this task is completed in the following order:
1. Defining the databases

The databases to be queried consist of the primary database and auxiliary databases.

The primary database is the first database to which the client is connected. This is the database from which the OpenEdge SQL Server is generated. The catalog name of the primary database is the root name of the database. For example, in the database path /usr/databases/customer, the root name of the database is customer. Therefore, customer will be the catalog name of the primary database.

The auxiliary databases are the databases other than the primary database to which the client will connect in a multi-database environment.

2. Ensuring permissions to access databases to be queried

The first database to which the client is connected is identified as the primary database. The other databases to be included in the multi-database environment are designated as the auxiliary databases. A client must have a valid user identification and password and must be granted appropriate privileges to all databases in order to perform a multi-database query.

3. Connecting to the databases

In order to successfully connect to an auxiliary database, the administrator of that database must have given you permission to access that database. When connecting to an auxiliary database, the OpenEdge SQL Server will authenticate you as a valid user of the auxiliary database using the username and password that you specified when you attached to the OpenEdge SQL Server. Users who are already connected to a primary database can connect to the auxiliary databases. The connection is made by using the `CONNECT AS CATALOG` statement.

4. Performing the query

The multi-database query is performed using a `SELECT` statement. Specific databases are identified in the query by using their catalog names.

5. Disconnecting

Disconnect from the auxiliary databases using the `DISCONNECT CATALOG` statement.

**Note:** All databases used in an OpenEdge multi-database environment must reside in a single machine environment, such as a single computer system. The computers must have been started as servers by using the `proserve` command or an equivalent method.

---

**Working with catalogs in multi-database queries**

All data references in SQL follow a syntax where a reference has from one to four components of the form `catalog.schema.table.column`.

OpenEdge SQL uses database naming conventions shown in the following table, which are in full compliance with the SQL standard.
Table 11: OpenEdge SQL Database Naming Conventions

<table>
<thead>
<tr>
<th>Terms</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalog</td>
<td>A named collection of schemas. In OpenEdge SQL, a catalog logically corresponds to a database.</td>
</tr>
<tr>
<td>Schema</td>
<td>A collection of tables and other database objects.</td>
</tr>
<tr>
<td>Table</td>
<td>A collection of data organized into columns and rows.</td>
</tr>
</tbody>
</table>

Catalogs are used as name components to identify columns and tables in a multi-database query. A table name can have up to three components: catalog.schema.table-name. A column name can have up to four components: catalog.schema.table.column-name.

The primary database—already connected when an OpenEdge SQL process attaches to the OpenEdge SQL Server—is already assigned the name of the primary database as its catalog name. Each auxiliary database is assigned a name when the `CONNECT AS CATALOG` command is executed.

**Note:** Catalogs cannot be used in Data Definition Language statements such as `ALTER` and `CREATE`. Nor can they be used with sequences.

---

**Working with default catalogs**

When no catalog is specified in a command (three-level column identification), the OpenEdge SQL engine assumes the default catalog. Unless changed, the default is the initial connection or primary database.

**Working with catalogs and synonyms**

A public synonym, used without a qualifier, exists in the default catalog. The public synonym may be qualified by a catalog name, in which case the synonym must exist in the set of synonyms defined for that particular catalog.

A private synonym must always be qualified by at least the name of the schema where that synonym exists. A private synonym, used with only a schema qualifier, exists in the default catalog.

**Working with catalogs and stored procedures**

A stored procedure may be called during a client session with multiple catalogs active.

A stored procedure in an auxiliary catalog can be called by explicitly qualifying the procedure name by the catalog name. For example:

```
"call sales_db.smith.total_sales('Mary Jones')"
```
The SQL statements executed from within a stored procedure are interpreted in light of the multiple catalogs active. Consider a stored procedure in an auxiliary database. Tables referenced from statements in the stored procedure, if they are not qualified by a catalog name, will refer to tables in the default catalog, or primary database. This may not be what the author of the stored procedure intended. To avoid such problems, fully qualify table names if such usage is anticipated.

**Granting permissions to perform multi-database queries**

Privileges which exist in a database for a particular user apply only to the tables in that database. They do not apply to tables in another database. If a SQL query spans several databases, the administrators of all databases must grant access privileges to the user in order for the query to work.

**Limitations of the OpenEdge SQL multi-database environment**

OpenEdge SQL operations in a multi-database environment are limited to data retrieval from an auxiliary database. In other words, you cannot use any statement that writes to an auxiliary database. In a multi-database environment, the following limitations exist:

- Catalog names cannot be used with the following statements:
  - GRANT
  - CREATE INDEX
  - CREATE PROCEDURE
  - CREATE SYNONYM
  - CREATE TABLE
  - CREATE TRIGGER
  - CREATE VIE
  - UPDATE STATISTICS

- JTA transactions are not permitted when an auxiliary database connection exists.

- Catalog names cannot be used with sequences, including functions, such as CURRVAL and NEXTVAL.

- Although it is not possible to update data in an auxiliary database, you may use data from auxiliary databases to update data in the primary database. The following example uses the SELECT statement on an auxiliary database to enable an update in the primary database:

  ```sql
  DELETE FROM pub.Order
  WHERE OrderNum in (SELECT OrderNum FROM auxCatalog.pub.NewOrders WHERE OrderDate = SYSDATE);
  ```
Connecting to multiple databases

Connections to multiple databases may be made using one of two methods, either the use of SQL commands or the employment of a properties file that defines the connections.

Connecting to multiple databases using SQL commands

Connections to auxiliary databases can be established through the use of SQL commands. This section describes methods for specifying catalogs, determining catalog availability, connecting to databases, and disconnecting.

Specifying a database default catalog

The database catalog to be used for queries when the catalog is not given in a schema, table, or column specification is the primary database catalog unless the default is changed. The default catalog may be changed using the Progress Software Corporation specific SQL statement `SET CATALOG`.

**Note:** When you connect to auxiliary databases using a previously used pooled connection, you will inherit the database configurations as set by the previous user.

Using the SET CATALOG statement

Use the `SET CATALOG` statement to change the default catalog name to be used for schema, table, and column references.

The `SET CATALOG` statement uses the following syntax:

```
SET CATALOG catalog_name ;
```

The following example shows how the auxiliary database connection is identified by the catalog named `mydb1`.

```
SET CATALOG mydb1;
```

For more information on the `SET CATALOG` statement, see *OpenEdge Data Management: SQL Reference*.

Determining catalog availability

A list of available database catalogs may be obtained by using the Progress Software Corporation specific SQL statement `SHOW CATALOGS`. 
Using the SHOW CATALOGS statement

Use the SHOW CATALOGS statement to obtain a list of available database catalogs. The command returns a list of available catalog information with catalog name, catalog type (primary or auxiliary), and catalog status (default or notdefault).

The SHOW CATALOGS statement uses the following syntax:

```
SHOW CATALOGS [ ALL | { PRO_NAME | PRO_TYPE | PRO_STATUS } ], PRO_NAME | PRO_TYPE | PRO_STATUS }
```

The following example demonstrates the use of the SHOW CATALOGS statement.

```
SHOW CATALOGS PRO_NAME;
```

The SHOW CATALOGS statement is useful for obtaining the catalog names of databases currently connected for the user, for identifying the catalog name of the primary database and the current default catalog. Three columns of information can be returned by the SHOW CATALOGS statement and they describe the catalog's name, type (primary or auxiliary), and status (default or not default).

For more information on the SHOW CATALOG statement, see OpenEdge Data Management: SQL Reference.

Using the CONNECT AS CATALOG statement

Use the CONNECT AS CATALOG statement to connect to an auxiliary database.

The CONNECT AS CATALOG statement uses the following syntax:

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

In the following example, the database named customer in directory /usr/databases is connected as a catalog named 'mydb1'.

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

For more information on the CONNECT AS CATALOG statement, see OpenEdge Data Management: SQL Reference.

Disconnecting from catalogs

The DISCONNECT CATALOG statement removes a connection from an auxiliary read-only database.
Using the DISCONNECT CATALOG statement

Use the `DISCONNECT CATALOG` statement to remove a connection from an auxiliary read-only database.

The `DISCONNECT CATALOG` statement uses the following syntax:

```
DISCONNECT CATALOG catalog_name;
```

In the following example, an auxiliary database connection is removed.

```
DISCONNECT CATALOG mydb1;
```

For more information on the `DISCONNECT CATALOG` statement, see *OpenEdge Data Management: SQL Reference*.

Using properties files to enable multiple database connections

Many applications, such as Crystal Reports, may require the primary database to dynamically connect to auxiliary databases without the need to issue a `CONNECT` statement for each instance of an auxiliary connection. In instances such as this, you may create a properties file which will initiate the auxiliary database connections once an application makes the initial connection to the primary database.

Creating a properties file

The properties file must follow a specific format and contain information that appropriately defines the auxiliary databases. The following example provides an example of a multi-database connection properties file.
Example: Multi-database connection properties file

```
[sq1-configuration]
   configuration-names-list=NortheastSales, ALBSales
[configuration.NortheastSales]
   database-id-list=MA, NH
[database.MA]
   Name=Mass
   Catalog=Mass
   Location=/usr1/kjain/States/Mass
[database.NH]
   Name=NewHampshire
   Catalog=NH
   Location=/usr1/kjain/States/NewHamp
[configuration.ALBSales]
   database-id-list=Ny, Mal
[database.Ny]
   Name= NewYork
   Catalog=Lions
   Location=/usr1/kjain/States/NewYork
[database.Mal]
   Name= Mass
   Catalog=Bears
   Location=/usr1/kjain/States/Mass
```

The name of the multi-database connection properties file has the following format:

```
<database>.oesql.properties
```

SQL configuration properties are found after the [sql-configuration] directive in the OpenEdge SQL properties file. Property names and values are separated by an equal sign. For example, catalog=auto. The following table describes SQL configuration properties:

### Table 12: SQL configuration properties and their values

<table>
<thead>
<tr>
<th>Property</th>
<th>Type and length</th>
</tr>
</thead>
<tbody>
<tr>
<td>configuration-names-list</td>
<td>Character [1024]</td>
</tr>
</tbody>
</table>

The following specifics pertain to the SQL properties:

- The SQL properties specifies a comma-separated list of configuration names.
- The configuration name must match the configuration name specified in the properties section.
- Each configuration must have a unique name.
- The name cannot contain a hyphen (-).
- This property is used to list all available configurations.
- This property is **case-insensitive**.

Configuration properties are found after the configuration.configuration-name directive. The value of the configuration-name must exactly match one of the names specified for the configuration-names-list property under the sql-configuration directive. If you need two configurations, you need a configuration.configuration-name directive for each of the configurations. The following table describes the configuration properties:
Connecting to multiple databases

Table 13: Configuration properties and their values

<table>
<thead>
<tr>
<th>Property</th>
<th>Type and length</th>
</tr>
</thead>
<tbody>
<tr>
<td>database-id-list</td>
<td>Character [1024]</td>
</tr>
</tbody>
</table>

The following specifics pertain to the configuration properties:

- Specifies a list of database identifiers.
- Database identifiers need not be the name of the database. It is merely an identifier which must match the database name specified in the database properties section.
- Each database identifier must be unique. This property is used to list the database identifier sections available under this configuration. This property is **case-insensitive**.

Database properties are found after the database.database-identifier directive. The value of the database identifier must exactly match one of the names specified for the database-id-list property under the configuration.configuration-name directive. If you want three auxiliary databases under a certain configuration, you need a database.database-id directive for each of the databases. The following table describes database properties:

Table 14: Database properties and their values

<table>
<thead>
<tr>
<th>Property</th>
<th>Type and length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Character [32]</td>
<td>This is the name of the physical database whose properties will follow. This property is <strong>case-insensitive</strong>.</td>
</tr>
<tr>
<td>catalog</td>
<td>Character [32]</td>
<td>This is the catalog name under which the database will be available in the multi-database environment. This property is <strong>case-insensitive</strong>.</td>
</tr>
<tr>
<td>location</td>
<td>Character [300]</td>
<td>This is the full/relative path of the database's .db file. In case it is a relative path, it should be relative to the location of this properties files, which is the same as the location of the primary database. This property is <strong>case-sensitive</strong>.</td>
</tr>
</tbody>
</table>

The following are the characteristics of using a multi-database connection properties file:

- The properties file should have the same name as the primary database, without the database extension followed by the oesql.properties extension. It should be available in the same directory as the database .db file. For example, the properties file for /usr1/sales.db should be available in the /usr1 directory and it should be named sales.oesql.properties.
- The properties file is optional. The absence of a properties file indicates a single database connection environment. Similarly, an empty properties file with the correct name results a single database connection environment.
- It is important to name the file appropriately with the correct extension for each primary database that will support the automatic connection feature in a multi-database setting.
- The properties file may contain the same database in multiple configurations with different catalog names. For example, the database named Massachusetts appear in the properties file.
in the following example in two configurations with two different catalog names. Duplicate versions of the same physical database cannot exist within a single configuration.

- The properties file should contain only the list of databases to be connected as auxiliary connections. It should not contain the primary database. The primary connection would already have been established before the properties file is read.

- An error while reading/parsing the properties file will result in an appropriate error returned to the application. No auxiliary database connections will result if an error occurs while connecting to an auxiliary database.

- Once the primary connections is made, OpenEdge SQL Server will check for the presence of the properties file having the same name as the database followed by the required extension. If it cannot find the file, then a single database environment will result. If the file is found, the contents of the file will be parsed to create a list of database objects on the server and the appropriate configuration will be chosen to establish the connections to auxiliary databases.

- The complete properties file will be read at once after the primary connection is made.

- The auxiliary connections will be made in an all-or-none manner and in one configuration only. For example, from the properties file in the following example, if "ALB-sales" is chosen as the configuration, then connection to both "New York" and "Massachusetts" will be made as auxiliary connections. If an error occurs while making connections to any of the databases, then all other successful connections will be disconnected.

- The configuration file can be specified as part of the connection URL to the primary database. The file is named within square brackets after the primary database, as shown in the following example.

```
jdbc:datadirect:openedge://localhost:6790;databaseName=empty[-mdbq:AlbSales]
```

Similarly, the properties file can be specified when identifying the DSN used by an ODBC client, as shown in the following example.

```
Data Source Name: AlbTeams
Description: ALB Teams Databases
Host Name: albhost
Port Number: 3535
Database Name: empty[-mdbq:AlbSales]
User ID: bjones
```

Notes

- Once the properties file is read and the list of database objects is constructed, the database objects in the configuration is filtered on the basis of the configuration specified by the user in the connection URL. The connections to those databases are then established as auxiliary connections.

- A configuration specified in the connection URL that does not exist in the properties file will result in an appropriate error returned to the application and only the primary connection will exist. A message is also logged to the database log file (.lg).

- If a configuration is not specified on the connection URL, the primary database operates in single-database environment.

- The auxiliary database object list that is the parsed contents of the properties file will be maintained internally by the OpenEdge SQL Server and is shared process-wide. Each time a
new client connects to the server, the server will check the file modification timestamp. If the
file has not been changed since the last time the properties file was read and the DB object list
was created, the server will use that list with appropriate mutexing to read the list concurrently.
Otherwise, the server will refresh the list with the properties file contents appropriately mutexed
to prevent readers from accessing the list when it is being updated.

• Once the application/client disconnects from the server, all auxiliary connections initiated on
its behalf by the server through the properties file, will be disconnected in an orderly manner.

An example of a multi-database query

Performing a multi-database query consists of the steps outlined in the sections that follow:

Connecting to an auxiliary database

A client is connected to a general ledger database and would like to connect to the accounts payable database. The accounts payable database exists in the directory called /usr/data/accounting. The client executes the following statement:

```
CONNECT '/usr/data/accounting/apdb' AS CATALOG actpay;
```

Performing a multi-database query

The client is connected to the accounting databases. The catalog name for the first database is gledger and the catalog name for the second database is actpay. The client would like to execute a query statement against the accounts table of the pub schema in the gledger catalog and the accounts table of the pub schema in the actpay catalog. The client executes the following statement:

```
SELECT actpay.pub.accounts.name FROM gledger.accounts, actpay.accounts
WHERE actpay.pub.accounts.closed = 'n' AND gledger.pub.accounts.closed = 'n';
```

Disconnecting an auxiliary database

The client is connected to the general ledger database, gdb, and an accounts payable database, apdb. The client executes the following statement:

```
DISCONNECT CATALOG actpay;
```
The Java Transaction API (JTA) allows applications to perform distributed transactions on multiple networked computer resources. This chapter provides an overview of that process and how OpenEdge and DataDirect Connect for JDBC drivers relate to that.

For details, see the following topics:

- JTA's role in J2EE
- Understanding JTA architecture
- Understanding application interfaces
- JTA and the distributed transaction process
- Planning for JTA transaction support

**JTA's role in J2EE**

The OpenEdge platform has built-in interoperability based on industry standards and can easily integrate into a J2EE environment, adding value to an integrated enterprise.

Java applications almost always use the JDBC (Java Database Connectivity) specification to work with relational databases. JDBC is also part of the J2EE specification. The OpenEdge RDBMS includes the DataDirect JDBC driver, making the OpenEdge database an integration point for any kind of J2EE application.

As illustrated in the following figure, JTA defines transaction management between a transaction manager and a resource manager within the J2EE architecture. In this scenario, OpenEdge is the resource manager and an SQL server or application server is the transaction manager.
Understanding JTA architecture

The JTA architecture specifies standard Java interfaces between transaction managers and the parties involved in distributed transactions: the application, application server, and resource manager that controls access to shared resources affected by the transactions.

Transaction managers can be either a manager existing as part of an application server or as part of an application program. Resource managers are typically database storage engines. For example, a Java application server may serve as transaction manager while the OpenEdge Database storage engine may serve as resource manager.

The JTA architecture, depicted in the following figure, creates the environment in which distributed transactions are performed. The resource manager for OpenEdge consists of the OpenEdge SQL engine and the OpenEdge storage engine, working as a single component. The transaction manager, a component of a J2EE application server, is responsible for coordinating transactions that may span resource managers and application connections. The resource adapter for OpenEdge is the JDBC driver. It includes support for the XADataSource, XAConnection, and XAResource objects.
Understanding application interfaces

JTA specifies the Java interfaces between a transaction manager and the components of a distributed transaction system. These interfaces are implemented through the JDBC driver in cooperation with OpenEdge SQL.

XADatasource

An application that performs distributed transactions must use a JDBC driver that supports the XADatasource interface. The XADatasource interface provides a facility for creating and interacting with a physical connection to a data source. The connection is reusable and allows for participation in distributed transactions. The XADatasource object manages the interactions of the connection-pooling manager and the transaction manager components of the J2EE server.

XAConnection

In order to facilitate distributed transactions, the JDBC driver must also support the XAConnection interface. An XAConnection object is a PooledConnection object extended to participate in distributed transactions. An application that uses distributed transactions obtains a physical connection to the data source through an XAConnection object. The J2EE server is responsible for managing the connections and distributed transactions.
XAResource

The XAResource interface provides an object that the transaction manager uses to control distributed transactions. The transaction manager obtains an XAResource object for each connection in a global transaction. The transaction manager uses the `start` method to associate the global transaction with the resource, and it uses the `end` method to disassociate the transaction from the resource. The resource manager is responsible for associating the global transaction to all work performed on its data between the `start` and `end` method invocations. At transaction commit time, the resource managers are informed by the transaction manager to prepare, commit, or rollback a transaction according to the two-phase commit protocol.

XAResource methods

The methods of the XAResource interface provide the transaction manager with the ability to interact with the resource manager, which consists of the OpenEdge SQL server and the OpenEdge storage engine. The XAResource interface uses the following methods:

- **Start** — The `start` method works on behalf of a transaction branch specified by global transaction identifier, referred to as an XID. A start specifies that:
  - A new transaction branch is desired.
  - There was a request to join a transaction previously seen by the resource manager.
  - There was a request to resume a suspended transaction.

  A request to join or resume an existing transaction does not need to occur on the connection that the new transaction branch was requested. It may occur on a server other than the one that the new transaction branch was requested.

- **End** — The `end` method ends the work performed on behalf of a transaction branch. The resource manager disassociates the XA resource from the transaction branch specified and lets the transaction complete. The transaction completes in one of three ways:
  - A transaction is suspended temporarily in an incomplete state and must be resumed via the `start` method.
  - A transaction is flagged to indicate that a portion of work has failed. The resource manager marks the transaction as rollback-only.
  - A transaction ends if that the portion of work has completed successfully.

- **Prepare** — The resource manager is asked to prepare for a transaction commit of the transaction. The request occurs on a connection other than where the work for the transaction branch was executed.

- **Commit** — The resource manager is asked to commit the transaction. The request occurs on a connection other than where the work for the transaction branch was executed.

- **Rollback** — The `rollback` method prompts the resource manager to roll back the work done for the transaction. The request may occur on a connection other than where the work for the transaction branch was executed.

- **Recover** — The `recover` method prompts the resource manager for a list of prepared transaction branches. The transaction manager calls this method during recovery to obtain the list of transaction branches that are currently in prepared or in heuristically completed states.
JTA and the distributed transaction process

A distributed transaction occurs when the application sends a transaction request to the transaction manager. The transaction is completed by the final commit/rollback decision.

JTA transactions and two-phase commit protocol

Each transaction branch must be committed or rolled back by the local resource manager. The transaction manager controls the boundaries of the transaction and is responsible for the final decision as to whether or not the total transaction commits or is rolled back. This decision is made in a process commonly known as a two-phase commit protocol.

In the first phase, the transaction manager polls all of the resource managers involved in the distributed transaction to see if each one is ready to commit. If a resource manager cannot commit, it responds negatively and rolls back its particular part of the transaction so that data is not altered.

In the second phase, the transaction manager determines if any of the resource managers have responded negatively, and, if so, rolls back the whole transaction. If there are no negative responses, the translation manager commits the whole transaction, and returns the results to the application.

The following example demonstrates the use of a two-phase commit protocol to commit one transaction branch:

```java
XADatasource xaDS;
XAConnection xaCon;
XAResource xaRes;
Xid xid;
Connection con;
Statement stmt;
int ret;
xaDS = getDataSource();
xaCon = xaDS.getXAConnection("jdbc_user", "jdbc_password");
xaRes = xaCon.getXAResource();
con = xaCon.getConnection();
stmt = con.createStatement();
xid = new MyXid(100, new byte[]{0x01}, new byte[]{0x02});
try {
    xaRes.start(xid, XAResource.TMNOFLAGS);
    stmt.executeUpdate("insert into test_table values (100)");
    xaRes.end(xid, XAResource.TMSUCCESS);
    ret = xaRes.prepare(xid);
    if (ret == XAResource.XA_OK) {
        xaRes.commit(xid, false);
    }
} catch (XAException e) {
    e.printStackTrace();
} finally {
    stmt.close();
    con.close();
    xaCon.close();
}
```
JTA transactions and conventional transactions

When an OpenEdge database is configured as a resource manager for distributed JTA transactions, the transaction manager is responsible for establishing and maintaining the state of the transaction. The database receives an identifier for the global transaction. Multiple threads of execution process the transaction and have the following effects:

- Records can be locked by a JTA transaction with no user associated with the lock.
- Record locks can exist at database startup.
- Locks are owned by the transaction and not the user.

JTA transactions and crash recovery

During a typical database operation, crash recovery occurs whenever the database is started in single or multi-user mode. The process is completed in three stages—physical redo, physical undo and logical undo. At the end of this process, the database is presumed to have been made durable; all outstanding transactions have been committed or rolled back.

JTA transactions require that transactions must be able to be restored to a state pending the end of the two-phase commit. This means crash recovery logs what types of locks are acquired during runtime. The lock manager generates a new note indicating a table lock has been taken and identifies the lock strength. As record operations occur, recovery notes capture information on additional table, record or schema locks applied.

After the first physical redo phase of crash recovery has completed, the lock acquisition phase identifies JTA transactions that need transactional and locking information applied.

JTA transactions and OpenEdge Replication

When a JTA-enabled OpenEdge database is enabled for site replication with OpenEdge Replication, JTA transactions are replicated from the source database to the target database or databases. JTA transactions are started against the replication source database and replicated to the replication target database by the replication agent. Manual intervention to resolve JTA transactions should only be performed on a replication source database when there is an unrecoverable catastrophic failure to JTA transaction manager. On a target database JTA transactions can only be resolved when Replication Agent is in the "pre-transition" state of OpenEdge Replication.

For more information on OpenEdge Replication, see OpenEdge Replication: User Guide. For more information on manually resolving JTA transactions, see the .

Planning for JTA transaction support

JTA transaction support places increased demands on database resources. This section discusses those demands and tools for enabling JTA support.

For more information database resource management and database utilities, see OpenEdge Data Management: Database Administration.
JTA transactions and database resource planning

JTA transactions affect the way you structure and use your database as well as monitor its performance. The following issues must be considered when planning for JTA transaction support:

- **Space allocation** — The transactional requirements of JTA demand more usage of OpenEdge database before-image (BI) and after-image (AI) files. Therefore, BI and AI files should be increased in size by 30 percent to accommodate JTA transactions.

  The database needs to re-establish any JTA transactions which have reached a prepared state after a shutdown or crash. Database BI clusters are not reused until the prepared JTA transactions are committed or rolled back. Should this condition persist, BI files can grow considerably.

- **Transaction table usage** — The transaction table is increased by the Maximum Number of JTA Transactions (-maxxids) startup parameter. A JTA-enabled database uses more shared memory for the expanded transaction table and an additional table which holds information specific to each JTA transaction.

- **Size of the Xid Table** — JTA-enabled databases make use of the -maxxids startup parameter. This parameter is used to size both the transaction table and the Xid table which stores information specific to each active JTA transaction. The default value for the parameter is 100 for JTA-enabled databases. The database engine ensures that the number of JTA transactions is not exceeded by the limit specified by the -maxxids startup parameter. This also prevents JTA transactions from monopolizing the entire transaction table. Any JTA transactions left in the prepared state retain their locks until the transactions are completed or rolled back.

- **Size of the Lock Table** — JTA transactions do not use more locks than conventional transactions, but they can hold locks for longer periods. As a result, this might require a larger value for the Lock Table Entries (-L) startup parameter.

Enabling JTA support

The `proutil` utility is used to enable JTA support for your database. The command uses the following syntax:

### Syntax

```
proutil db-name -C enablejta
```

Disabling JTA support

Similarly, the `proutil` utility is used to disable JTA support. The command uses the following syntax:

```
proutil db-name -C disablejta
```
Monitoring JTA transactions

Database administrators of JTA-enabled databases must be able to monitor the status of those transactions. Both the promon utility and the virtual system tables are used to display JTA transaction and lock statuses.

JTA transaction locks belong to the transaction and not to any of the users who may have participated in the work of the transaction.

The promon utility identifies the JTA transaction states, as shown in the following table.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active JTA</td>
<td>The transaction is currently active</td>
</tr>
<tr>
<td>Idle JTA</td>
<td>The transaction is not currently executing</td>
</tr>
<tr>
<td>Prepared JTA</td>
<td>The transaction is prepared</td>
</tr>
<tr>
<td>Rollback-only JTA</td>
<td>The transaction has encountered an error</td>
</tr>
<tr>
<td>Commit JTA</td>
<td>The transaction is in the commit process</td>
</tr>
</tbody>
</table>

Resolving JTA transactions

Because OpenEdge relinquishes transaction control for JTA transactions, it is dangerous to intervene and manually resolve outstanding JTA transactions. Intervention could compromise the referential integrity of the database as a result. Resolve JTA transactions manually only when there has been an unrecoverable catastrophic failure to the JTA transaction manager. When necessary, you can use the promon utility to identify and resolve JTA transactions.
Stored Procedures and Triggers

Stored procedures are Java routines that are executed by the OpenEdge SQL engine. A stored procedure is explicitly invoked by a client application, another stored procedure, or a trigger procedure. SQL limits recursive invocation of a stored procedure to five levels.

This chapter describes when and how to use stored procedures, as detailed in the following sections. For details, see the following topics:

- Setting up OpenEdge SQL for stored procedures and triggers
- Basics of Java stored procedures
- Stored procedure fundamentals
- Writing stored procedures
- Working with triggers

Setting up OpenEdge SQL for stored procedures and triggers

OpenEdge ships with one of two scripts—$DLC/bin/sql_env or %DLC%/bin/sql_env.bat— to facilitate setting up the appropriate environment if you plan to use OpenEdge SQL to run stored procedures and triggers.

To run sql_env, the Java jdk1.4.x must be installed. The JDKHOME environment variable must be pointed to it on all platforms in which OpenEdge does not ship the jdk (IBM AIX, Unixware, Linuxx86).
The `sql_env` script should be executed after `proenv`.

**Enabling stored procedures on 64-bit platform databases**

Prior to OpenEdge Release 10.1C, databases created on 64-bit platforms did not have the capability to employ stored procedures and triggers. To enable stored procedures and triggers in databases created on 64-bit platforms prior to OpenEdge Release 10.1C, use the PROUTIL `enablestorproc` utility. For example:

```
proutil dbname -C enablestorproc
```

You can also enable stored procedures and triggers by using the PROUTIL `conv910` utility, which converts Progress® Version 9 databases to OpenEdge Release 10 databases. For more information on using the PROUTIL utility, see *OpenEdge Data Management: Database Administration*.

**Basics of Java stored procedures**

A stored procedure is a snippet of Java code embedded in a `CREATE PROCEDURE` statement. The Java snippet can use all standard Java features as well as use OpenEdge SQL-supplied Java classes for processing any number of SQL statements.

**Advantages of stored procedures**

Stored procedures and triggers expand the flexibility and performance of applications that access the OpenEdge SQL environment. They provide a mechanism for storing a collection of SQL statements and Java program constructs that enforce business rules and perform administrative tasks in a database.

Stored procedures and triggers enhance applications by:

- Enabling a client application to perform a procedure with a single request instead of multiple requests for each SQL statement.
- Executing faster than a corresponding SQL script.
- Implementing elaborate algorithms to enforce complex business rules. The details of the procedure implementation can change without requiring changes in an application that calls the procedure.

**How OpenEdge SQL interacts with Java**

OpenEdge SQL stored procedures allow the use of standard Java programming constructs along with standard SQL statements. To do this, the OpenEdge SQL Engine interacts with Java in the following ways:

- When you create a stored procedure, the SQL engine processes the Java code, submits it to the Java compiler, receives the compiled result, and stores the result in the database.
• When an application calls a stored procedure, the SQL engine interacts with the Java Virtual Machine (JVM) to execute the stored procedure and receive any result.

Creating stored procedures

The Java source text that makes up the body of a stored procedure is not a complete Java program, but a program fragment or snippet that the OpenEdge SQL Engine converts into a complete Java class when it processes a `CREATE PROCEDURE` statement. Creating a stored procedure involves the following steps:

1. A client application or tool issues a `CREATE PROCEDURE` statement that contains the Java source text.
2. The OpenEdge SQL Engine adds code to the Java snippet to create a complete Java class and submits the combined code to the Java compiler.
3. Presuming there are no compilation errors, the Java compiler returns compiled bytecode back to the OpenEdge SQL Engine. If there are compilation errors, the OpenEdge SQL Engine passes the first error message generated by the compiler back to the application or tool that issued the `CREATE PROCEDURE` statement.
4. The OpenEdge SQL Engine stores both the Java source text and the bytecode form of the procedure in the database.

The following figure illustrates the general steps for creating a Java stored procedure.

Figure 7: Creating Java stored procedures

Calling stored procedures

Once a stored procedure is created and stored in the database, any application or other stored procedure can execute it. You can call stored procedures from either ODBC applications or JDBC applications.

Example: Stored procedure using ODBC syntax
The following example shows an excerpt from an ODBC application that calls a stored procedure (order_parts) using the ODBC syntax

\{ call procedure_name ( param ) \}.

```c
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);
```

A stored procedure executes using the following process:

1. The application calls the stored procedure through its native calling mechanism. The previous example uses the ODBC call escape sequence.
2. The OpenEdge SQL retrieves the compiled bytecode form of the procedure and submits it to the Java Virtual Machine for execution.
3. For every SQL statement in the procedure, the Java Virtual Machine calls OpenEdge SQL.
4. OpenEdge SQL manages the interaction of the stored procedure with the database and execution of the SQL statements, and returns any result to the Java Virtual Machine.
5. The Java Virtual Machine returns result (output parameters and result sets) of the procedure to OpenEdge SQL, which in turn passes them to the calling application.

The following figure illustrates the steps in executing a stored procedure.

**Figure 8: Executing stored procedures**
Using stored procedures

Stored procedures extend the SQL capabilities of a database by adding control through Java program constructs that enforce business rules and perform administrative tasks.

Stored procedures can take advantage of the power of Java programming features. Stored procedures can:

• Receive and return input and output parameters
• Handle exceptions
• Include any number and kind of SQL statements to access the database
• Return a procedure result set to the calling application
• Make calls to other procedures
• Use predefined and external Java classes

OpenEdge SQL supports SQL statements in Java through several classes. See OpenEdge Data Management: SQL Reference for more information.

The following table summarizes the functionality of these OpenEdge SQL-supplied classes.

### Table 16: Summary of OpenEdge SQL Java classes

<table>
<thead>
<tr>
<th>Functionality</th>
<th>OpenEdge SQL Java class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate (one-time) execution of SQL statements</td>
<td>SQLIStatement</td>
</tr>
<tr>
<td>Prepared (repeated) execution of SQL statements</td>
<td>SQLPStatement</td>
</tr>
<tr>
<td>Retrieval of SQL Prepared (repeated) execution of SQL statements result sets</td>
<td>SQLCursor</td>
</tr>
<tr>
<td>Returning a procedure result set to the application</td>
<td>DhSQLResultSet</td>
</tr>
<tr>
<td>Exception handling for SQL statements</td>
<td>DhSQLException</td>
</tr>
</tbody>
</table>

Stored procedure fundamentals

This section discusses the fundamentals of writing stored procedures.

The SQL CREATE PROCEDURE statement provides the basic framework for stored procedures. Use the CREATE PROCEDURE statement to submit a Java code snippet that will be compiled and stored in the database.
The `CREATE PROCEDURE` statement uses the following syntax:

**Syntax**

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

**parameter_decl**

```
{ IN | OUT | INOUT } parameter_name data_type
```

**Java snippet**

The core of the stored procedure is the Java snippet. The snippet contains a sequence of Java statements. When it processes a `CREATE PROCEDURE` statement, OpenEdge SQL adds header and footer “wrapper” code to the Java snippet. This wrapper code:

- Declares a class with the name `username_procname_SP`, where `username` is the user name of the database connection that issued the `CREATE PROCEDURE` statement and `procname` is the name supplied in the `CREATE PROCEDURE` statement.

- Declares a method within that class that includes the Java snippet. When an application calls the stored procedure, the SQL engine calls the Java virtual machine to invoke the method of the `username_procname_SP` class.

**Structure of stored procedures**

There are two parts to any stored procedure:

- The procedure specification provides the name of the procedure and can include other optional clauses, such as:
  - Parameter declarations
  - Procedure result set declarations
  - Import clauses

- The procedure body contains the Java code that executes when an application invokes the procedure.

A simple stored procedure requires the procedure name in the specification and a statement requiring no parameters in the body. The following example assumes the existence of a table named `HellowWorld`, and inserts a quoted string into that table.
Example: Creating a stored procedure

```sql
CREATE PROCEDURE HelloWorld ()
BEGIN
    SQLIStatement Insert_HelloWorld = new SQLIStatement (
        "INSERT INTO HelloWorld(fld1) values ('Hello World!')");
    Insert_HelloWorld.execute();
END;
```

Example: Executing a stored procedure

```
SQLExplorer> CREATE TABLE helloworld (fld1 CHAR(100));
SQLExplorer> CALL HelloWorld();
0 records returned
SQLExplorer> SELECT * FROM helloworld;
FLD1
----
Hello World!
1 record selected
```

Subsequently, you can execute the procedure shown in the following example.

The procedure specification can also contain other clauses.

Parameter declarations specify the names and types of parameters that the calling application will pass and receive from the procedure. Parameters can be input, output, or both.

The procedure result set declaration details the names and types of fields in a result set the procedure generates. The result set is a set of rows that contain data generated by the procedure. If a procedure retrieves rows from a database table, for instance, it can store the rows in a result set for access by applications and other procedures. The names specified in the result-set declaration are not used within the stored procedure body. Instead, methods of the OpenEdge SQL Java classes refer to fields in the result set by ordinal number, not by name.

The import clause specifies which packages the procedure needs from the Java core API. By default, the Java compiler imports the `java.lang` package. The `IMPORT` clause must list any other packages the procedure uses. OpenEdge SQL automatically imports the packages it requires.

The following example shows a more complex procedure specification that contains these elements.

```sql
CREATE PROCEDURE new_sal (IN deptnum INTEGER, IN pct_incr INTEGER)
RESULT (empname CHAR (20), oldsal NUMERIC, newsal NUMERIC)
IMPORT
    import java.dbutils.SequenceType;
BEGIN
    .
    .
END
```
Writing stored procedures

Use any text editor to write the CREATE PROCEDURE statement and save the source text as a text file. That way, you can easily modify the source text and try again if it generates syntax or Java compilation errors.

From the command prompt, you can invoke SQL Explorer and submit the file containing the CREATE PROCEDURE statement as an input script, as shown in the following example.

Example: CREATE PROCEDURE input script

```bash
$ sqlexp -infile hello_world_script.sql example_db
```

Example: CREATE PROCEDURE in context of application call

The following example illustrates the use of the CREATE PROCEDURE statement in the context of an application call.

```sql
-- File name: hello_world_script.sql
-- Purpose: Illustrate a CREATE PROCEDURE statement.
@echo true;
@autocommit true;
CREATE PROCEDURE HelloWorld ()
BEGIN
    SQLIStatement Insert_HelloWorld = new SQLIStatement (
        "INSERT INTO HelloWorld(fld1) values ('Hello World!'));
    Insert_HelloWorld.execute();
END
;COMMIT WORK;
```

The Java snippet within the CREATE PROCEDURE statement does not execute as a stand-alone program. Instead, it executes in the context of an application call to the method of the class created by the OpenEdge SQL Engine. This characteristic has the following implications:

- If the snippet declares any classes, it must instantiate them within the snippet to invoke their methods.
- It is not possible to make use of stdout messages in stored procedures. This means method invocations such as System.out.println will not display messages, because stdout cannot be used in a server process where stored procedures are executed. If you would like to put tracing code in your stored procedures, it is recommended that you open and close a regular text file via Java and write your messages to that file.

Invoking stored procedures

The manner in which applications call stored procedures depends on their environment.

From ODBC

From ODBC, applications use the syntax in the following ODBC call escape sequence:
Writing stored procedures

Syntax

```
{ CALL proc_name [ (parameter[ , ...]) ] }
```

Use parameter markers (question marks used as placeholders) for input or output parameters to the procedure. You can also use literal values for input parameters only. OpenEdge stored procedures do not support return values in the ODBC escape sequence.

Embed the escape sequence in an ODBC SQLExecDirect call to execute the procedure.

Example: Stored procedure passing a single output parameter

The following example shows a call to a stored procedure named `order_parts` that passes a single input parameter using a parameter marker.

```sql
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);
```

From JDBC

The JDBC call escape sequence is the same as in ODBC. For example:

```
{ CALL proc_name [ (parameter[ , ...]) ] }
```

Embed the escape sequence in a JDBC `CallableStatement.prepareCall` method invocation.

The following example shows the JDBC code parallel to the ODBC code excerpt shown in the previous example.

Table 17: JDBC stored procedure code

```java
try {
    CallableStatement statement;
    int Part_num = 318;
    // Associate the statement with the procedure call
    // (conn is a previously-instantiated connection object)
    statement = conn.prepareCall("call order_parts(?)");
    // Bind the parameter.
    statement.setInt(1, Part_num);
    // Execute the statement.
    statement.execute();
}```
Modifying and deleting stored procedures

To modify a procedure, you must drop and re-create it. To re-create the procedure, you need the original source of the `CREATE PROCEDURE` statement. Query system tables to extract the source of the `CREATE PROCEDURE` statement to a file.

The SQL `DROP PROCEDURE` statement deletes stored procedures from the database. Exercise care in dropping procedures, since any procedure that calls the dropped procedure will raise an error condition when the now nonexistent stored procedure is invoked.

Stored procedure security

The following guidelines apply to stored procedure security:

- To create a stored procedure, a user must have `RESOURCE` or `DBA` privileges.
- The `DBA` privilege entitles a user to execute any stored procedure.
- The `DBA` privilege entitles a user to drop any stored procedure.
- The owner of a stored procedure is given `EXECUTE` privilege on that procedure at creation time, by default.
- The privileges on a procedure can be granted to another user or to public either by the owner of that procedure or by the DBA.
- Stored procedures are executed with the definer’s rights, not the invoker’s. In other words, when a procedure is being executed on behalf of a user with `EXECUTE` privilege on that procedure, for the objects that are accessed by the procedure, the procedure owner’s privileges are checked and not the user’s. This enables a user to execute a procedure successfully even when the user does not have the privileges to directly access the objects that are accessed by the procedure, as long as the user has `EXECUTE` privilege on the procedure.

Using the OpenEdge SQL Java classes

This section describes how you use the OpenEdge SQL Java classes to issue and process SQL statements in Java stored procedures.

To process SQL statements in a stored procedure, you must know whether the SQL statement generates output (in other words, if the statement is a query) or not. `SELECT` statements, for example, generate results: they retrieve data from one or more database tables and return the results as rows in a table.

Whether a statement generates such an SQL result set determines which OpenEdge SQL Java classes you should use to issue it. For example:

- To issue SQL statements that do not generate results (such as `INSERT`, `GRANT`, or `CREATE`), use the `SQLIStatement` class for one-time execution, or the `SQLPStatement` class for repeated execution.
- To issue SQL statements that generate results (`SELECT` and, in some cases, `CALL`), use the `SQLCursor` class to retrieve rows from a database or another procedure’s result set.
In either case, if you want to return a result set to the application, use the DhSQLResultSet class to store rows of data in a procedure result set. You must use DhSQLResultSet methods to transfer data from an SQL result set to the procedure result set for the calling application to process it. You can also use DhSQLResultSet methods to store rows of data generated internally by the procedure.

In addition, OpenEdge SQL provides the DhSQLException class so procedures can process and generate Java exceptions through the try, catch, and throw constructs.

Passing values to SQL statements

Stored procedures must be able to pass and receive values from SQL statements they execute. They do this through the setParam and getValue methods.

setParameter method: pass input values to SQL statements

The setParam method sets the value of an SQL statement's parameter marker to the specified value (a literal value, a procedure variable, or a procedure input parameter).

The setParam method takes two arguments. This is the syntax for setParam:

Syntax

```
setParameter ( marker_num, value );
```

**marker_num**

Specifies the ordinal number of the parameter marker in the SQL statement that is to receive the value as an integer. 1 denotes the first parameter marker, 2 denotes the second, n denotes the nth.

**value**

Specifies a literal, variable name, or input parameter that contains the value to be assigned to the parameter marker.

The following example shows a segment of a stored procedure that uses setParam to assign values from two procedure variables to the parameter markers in an SQL INSERT statement. When the procedure executes, it substitutes the value of the cust_number procedure variable for the first parameter marker and the value of the cust_name variable for the second parameter marker.

```
SQLIStatement insert_cust = new SQLIStatement ( "INSERT INTO customer VALUES (?,?) ");
insert_cust.setParam (1, cust_number);
insert_cust.setParam (2, cust_name);
```
The following example shows a procedure using the `is.NULL` method. The `is.NULL` method is used to check for a null value.

```java
if (!NEWROW.isNULL(1))
callStmt.setParam(1, (Integer) NEWROW.getValue(1, INTEGER));
if (!NEWROW.isNULL(2))
callStmt.setParam(2, (String) NEWROW.getValue(2, VARCHAR));
if (!NEWROW.isNULL(3))
callStmt.setParam(3, (java.math.BigDecimal) NEWROW.getValue(3, DECIMAL))
```

**getValue method: pass values from SQL result sets to variables**

The `getValue` method of the `SQLCursor` class assigns a single value from an SQL result set (returned by an SQL query or another stored procedure) to a procedure variable or output parameter using the following syntax:

**Syntax**

```java
getValue ( col_num , sql_data_type ) ;
```

*col_num*

Specifies the desired column of the result set as integer. `getValue` retrieves the value in the currently fetched record of the column denoted by `col_num`. 1 denotes the first column of the result set, 2 denotes the second, n denotes the nth.

*sql_data_type*

Specifies the corresponding SQL data type.

This method returns a Java object that must be cast to the corresponding SQL data type. This example shows how the `getValue()` method works:

```java
cnum = (Integer) NEWROW.getValue(1, INTEGER);
cname = (String) NEWROW.getValue(1, CHARACTER);
```

**Passing values to and from stored procedures: input and output parameters**

Applications need to pass and receive values from the stored procedures they call. They do this through input and output parameters. When applications process the `CREATE PROCEDURE` statement, the SQL engine declares Java variables of the same name. Therefore, the stored procedure can refer to input and output parameters as if they were Java variables declared in the body of the stored procedure.

Procedure result sets are another way for applications to receive output values from a stored procedure. Procedure result sets provide output in a row-oriented tabular format.
Parameter declarations include the parameter type (IN, OUT, or INOUT), the parameter name, and SQL data type.

Declare input and output parameters in the specification section of a stored procedure, as shown in the following example.

```sql
CREATE PROCEDURE order_entry (  
  IN cust_name  CHAR(20),  
  IN item_num   INTEGER,  
  IN quantity   INTEGER,  
  OUT status_code INTEGER,  
  INOUT order_num INTEGER
)
```

When the `order_entry` stored procedure executes, the calling application passes values for the `cust_name`, `item_num`, `quantity`, and `order_num` input parameters. The body of the procedure refers to them as Java variables. Similarly, Java code in the body of `order_entry` processes and returns values in the `status_code` and `order_num` output parameters.

**Implicit data type conversion between SQL and Java types**

When the OpenEdge SQL Engine creates a stored procedure, it converts the type of any input and output parameters.

The `java.lang` package, part of the Java core classes, defines classes for all the primitive Java types that "wrap" values of the corresponding primitive type in an object. The OpenEdge SQL Engine converts the SQL data types declared for input and output parameters to one of these wrapper types, as shown in the following table.

Be sure to use wrapper types when declaring procedure variables to use as arguments to the `getValue`, `setParam`, and `set` methods. These methods take objects as arguments and will generate compilation errors if you pass a primitive type to them.

The following example illustrates the use of the Java wrapper type `Long` for a SQL type `INTEGER`:

```sql
CREATE PROCEDURE proc1(INOUT f1 char(50), INOUT f2 integer)  
BEGIN  
f1 = new String("new rising sun");  
f2 = new Integer("999");  
END
CREATE PROCEDURE proc2()  
BEGIN  
String in1 = new String("String type");  
String out1 = new String();  
Long out2 = new Long("0");  
SQLCursor call_proc = new SQLCursor("call proc1(?, ?)");  
call_proc.setParameter(1,in1);  
// In setParam you can use either String or String type  
// for SQL types CHAR, and VARCHAR  
call_proc.setParameter(2,new Long("111"));  
call_proc.open();  
out1 = (String)call_proc.get Param(1,CHAR);  
// getParam requires String type for CHAR  
out2 = (Long)call_proc.get Param(2,INTEGER);  
call_proc.close();  
END
```
When the OpenEdge SQL Engine submits the Java class it creates from the stored procedure to the Java compiler, the compiler checks for data-type consistency between the converted parameters and variables you declare in the body of the stored procedure.

To avoid type mismatch errors, use the data-type mappings shown in the following table for declaring parameters and result-set fields in the procedure specification and the Java variables in the procedure body.

### Table 18: Mapping between SQL and Java data types

<table>
<thead>
<tr>
<th>SQL type</th>
<th>Java methods</th>
<th>Java wrapper type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR, VARCHAR</td>
<td>All</td>
<td>String</td>
</tr>
<tr>
<td>CHAR, VARCHAR</td>
<td>set, setParam</td>
<td>String</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>All</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>All</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>BIT</td>
<td>All</td>
<td>Boolean</td>
</tr>
<tr>
<td>TINYINT</td>
<td>All</td>
<td>Byte[1]</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>All</td>
<td>Integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>All</td>
<td>Integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>All</td>
<td>Integer</td>
</tr>
<tr>
<td>REAL</td>
<td>All</td>
<td>Float</td>
</tr>
<tr>
<td>FLOAT</td>
<td>All</td>
<td>Double</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>All</td>
<td>Double</td>
</tr>
<tr>
<td>BINARY</td>
<td>All</td>
<td>Byte[ ]</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>All</td>
<td>Byte[ ]</td>
</tr>
<tr>
<td>DATE</td>
<td>All</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>TIME</td>
<td>All</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>All</td>
<td>java.sql.Timestamp</td>
</tr>
</tbody>
</table>

### Executing an SQL statement

If an SQL statement does not generate a result set, stored procedures can execute the statement in one of two ways:

- **Immediate execution** — Using methods of the SQLIStatement class, the procedure executes a statement once.
• **Prepared execution** — Using methods of the `SQLPStatement` class, the procedure prepares a statement for multiple executions in a procedure loop.

## Immediate execution

Use immediate execution when a procedure must execute an SQL statement only once.

This stored procedure in this sample script inserts a row in a table. The constructor for `SQLIStatement` takes the SQL `INSERT` statement as its only argument. In the following example, the statement includes five parameter markers.

```sql
CREATE PROCEDURE insert_team(
    IN empnum INTEGER not null,
    IN FirstName VARCHAR(30) not null,
    IN LastName VARCHAR(50) not null,
    IN State VARCHAR(50) not null,
    IN Sport CHAR(20)
) BEGIN
    SQLIStatement insert_team = new SQLIStatement ("INSERT INTO team (empnum, FirstName, LastName, State, Sport) VALUES (?,?,?,? ,?)");
    insert_team.setParam (1, empnum);
    insert_team.setParam (2, FirstName);
    insert_team.setParam (3, LastName);
    insert_team.setParam (4, State);
    insert_team.setParam (5, Sport);
    insert_team.execute();
END
COMMIT WORK;
```

## Prepared execution

Use prepared execution when you must execute the same SQL statement repeatedly. Prepared execution avoids the overhead of creating multiple `SQLIStatement` objects for a single statement.

There is an advantage to prepared execution when you execute the same SQL statement from within a loop. Instead of creating an object with each iteration of the loop, prepared execution creates an object once and supplies input parameters for each execution of the statement.

Once a stored procedure creates a `SQLPStatement` object, you can execute the object multiple times, supplying different values for each execution.
The following example extends the previous example to use prepared execution.

```sql
CREATE PROCEDURE prepared_insert_customer (  
    IN cust_number INTEGER,  
    IN cust_name CHAR(20)  
) BEGIN  
    SQLPStatement p_insert_cust = new SQLPStatement (  
        "INSERT INTO customer VALUES (?,?) ");  
    .  
    int i;  
    for (i = 0; i < new_custs.[length]; i++)  
    {  
        p_insert_cust.setParam (1, new_custs[i].cust_number);  
        p_insert_cust.setParam (2, new_custs[i].cust_name);  
        p_insert_cust.execute ();  
    }END
```

Retrieving data: the SQLCursor class

Methods of the SQLCursor class let stored procedures retrieve rows of data. When stored procedures create an object from the SQLCursor class, they pass as an argument an SQL statement that generates a result set. The SQL statement is either a SELECT or a CALL statement:

- 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.

Either way, once the procedure creates an object from the SQLCursor class, the processing of result sets follows the same steps.

To process result sets:

1. Open the cursor by using the SQLCursor.open method.
2. Check whether there are any records in the result set by using the SQLCursor.found method.
3. If there are records in the result set, loop through the result set to:
   - Fetch a record by using the SQLCursor.fetch method.
   - Check whether the fetch returned a record with the SQLCursor.found method.
   - Assign values from the result-set record's fields to procedure variables or procedure output parameters by using the SQLCursor.getValue method.
   - Process the data, or
   - Exit the loop if the fetch operation did not return a record.
4. Close the cursor by using the SQLCursor.close method.
The following example uses SQLCursor to process the result set returned by an SQL SELECT statement.

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

Stored procedures also use SQLCursor objects to process a result set returned by another stored procedure. Instead of a SELECT statement, the SQLCursor constructor includes a CALL statement that invokes the desired procedure.

The following example shows an excerpt from a stored procedure that processes the result set returned by another procedure, get_customers.

```sql
SQLCursor cust_cursor = new SQLCursor ( "CALL get_customers (?) ");
cust_cursor.setParam (1, "NE");
cust_cursor.open () ;
for (;;) {
    cust_cursor.fetch ();
    if (cust_cursor.found ()){
        cust_number = (Integer) cust_cursor.getValue (1, INTEGER);
        cust_name = (String) cust_cursor.getValue (2, CHAR) ;
    }
    else
        break;
}
cust_cursor.close () ;
```

Returning a procedure result set to applications: the RESULT clause and DhSQLResultSet

The `get_sal` procedure in the previous example with a CREATE PROCEDURE uses the SQLCursor.getValue method to store the values of a database record in individual variables. The procedure did not, however, do anything with those values and they will be overwritten in the next iteration of the loop that fetches records.

The DhSQLResultSet class provides a way for a procedure to store rows of data in a procedure result set so that the rows can be returned to the calling application. There can only be one procedure result set in a stored procedure.
A stored procedure must explicitly process a result set to return it to the calling application. For example:

- Declare the procedure result set through the `RESULT` clause of the procedure specification.
- Populate the procedure result set in the body of the procedure using the methods of the `DhSQLResultSet` class.

When the SQL engine creates a Java class from a `CREATE PROCEDURE` statement that contains the `RESULT` clause, it implicitly instantiates an object of type `DhSQLResultSet`, and calls it `SQLResultSet`. Invoke methods of the `SQLResultSet` instance to populate fields and rows of the procedure result set.

This example extends the `get_sal` procedure to return a procedure result set:

```java
CREATE PROCEDURE get_sal2 ()
RESULT (
  empname CHAR(20),
  empsal NUMERIC
) 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 () ;
do {
  empcursor.fetch () ;
  if (empcursor.found () ) {
    ename = (StringBuffer) empcursor.getValue (1, CHAR);
    esal = (BigDecimal) empcursor.getValue (2, NUMERIC);
    // NUMERIC and DECIMAL are synonyms
    SQLResultSet.set (1, ename);
    SQLResultSet.set (2, esal);
    SQLResultSet.insert () ;
  }
} while (empcursor.found () ) ;
empcursor.close () ;
END
```

For each row of the SQL result set assigned to procedure variables, the procedure:

- Assigns the current values in the procedure variables to corresponding fields in the procedure result set with the `DhSQLResultSet.Set` method
- Inserts a row into the procedure result set with the `DhSQLResultSet.Insert` method

### Handling null values

Stored procedures routinely must set and detect null values. For example:

- Stored procedures might need to set the values of SQL statement input parameters or procedure result fields to null.
- Stored procedures must check if the value of a field in an SQL result set is null before assigning it through the `SQLCursor.getValue` method. The OpenEdge SQL Engine generates a run-time error if the result-set field specified in `getValue` is `NULL`. 

Setting SQL statement input parameters and procedure result set fields to null

Both the `setParam` method and `set` method take objects as their value arguments. You can pass a `NULL` reference directly to the method or pass a variable that has been assigned the null value.

The following example uses both techniques to set an SQL input parameter to `NULL`.

```
CREATE TABLE t1 (
  c1 INTEGER,
  c2 INTEGER,
  c3 INTEGER);
CREATE PROCEDURE test_nulls( )
BEGIN
  Integer pvar_int;
  pvar_int = null;
  SQLIStatement insert_t1 = new SQLIStatement
    ("INSERT INTO t1 (c1, c2, c3) values (?,?,?)");
  // Set to non-null value
  insert_t1.setParam(1, new Integer(1));
  // Set directly to null
  insert_t1.setParam(2, null);
  // Set indirectly to null
  insert_t1.setParam(3, pvar_int);
  insert_t1.execute();
END
```

Assigning null values from SQL result sets: the `SQLCursor.wasNULL` method

If the value of the field argument to the `SQLCursor.getValue` method is `NULL`, the SQL engine returns a run-time error.

The following example illustrates the error returned when the argument to `SQLCursor.getValue` is `NULL`.

```
(error(-20144): Null value fetched.)
```

This means you must always check whether a value is null before attempting to assign a value in an SQL result set to a procedure variable or output parameter. The `SQLCursor` class provides the `wasNULL` method for this purpose.

The `SQLCursor.wasNULL` method returns `TRUE` if a field in the result set is null. It takes a single integer argument that specifies which field of the current row of the result set to check.
The following example illustrates using the \texttt{wasNULL} method.

```
CREATE PROCEDURE test_nulls2( )
RESULT ( res_int1 INTEGER ,
res_int2 INTEGER ,
res_int3 INTEGER )
BEGIN
Integer pvar_int1 = new Integer(0);
Integer pvar_int2 = new Integer(0);
Integer pvar_int3 = new Integer(0);
SQLCursor select_t1 = new SQLCursor
("SELECT c1, c2, c3 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 pvar_intx procedure variables. Must first check
  // whether the values fetched are null: if they are, must set
  // pvars explicitly to null.
  if ((select_t1.wasNULL(1)) == true)
    pvar_int1 = null;
  else
    pvar_int1 = (Integer) select_t1.getValue(1, INTEGER);
  if ((select_t1.wasNULL(2)) == true)
    pvar_int2 = null;
  else
    pvar_int2 = (Integer) select_t1.getValue(2, INTEGER);
  if ((select_t1.wasNULL(3)) == true)
    pvar_int3 = null;
  else
    pvar_int3 = (Integer) select_t1.getValue(3, INTEGER);
  // Transfer the value from the procedure variables to the
  // columns of the current row of the procedure result set.
  SQLResultSet.set(1,pvar_int1);
  SQLResultSet.set(2,pvar_int2);
  SQLResultSet.set(3,pvar_int3);
  // Insert the row into the procedure result set.
  SQLResultSet.insert();
  select_t1.fetch();
} // Close the SQL result set.
select_t1.close();
END
```

Handling errors

OpenEdge SQL stored procedures use standard Java try/catch constructs to process exceptions. Any errors in SQL statement execution result in the creation of a \texttt{DhSQLException} class object. When OpenEdge SQL detects an error in an SQL statement, it throws an exception. The stored procedure should use try/catch constructs to process such exceptions. The \texttt{getDiagnostics} method of the \texttt{DhSQLException} class object provides a mechanism to retrieve different details of the error.

The \texttt{getDiagnostics} method takes a single argument whose value specifies which error message detail it returns. The following table shows the explanations of the \texttt{getDiagnostics} error-handling options.
Table 19: getDiagnostics error-handling options

<table>
<thead>
<tr>
<th>Argument value</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETURNED_SQLSTATE</td>
<td>The SQLSTATE returned by execution of the previous SQL statement</td>
</tr>
<tr>
<td>MESSAGE_TEXT</td>
<td>The condition indicated by RETURNED_SQLSTATE</td>
</tr>
<tr>
<td>CLASS_ORIGIN</td>
<td>Not currently used; always returned null</td>
</tr>
<tr>
<td>SUBCLASS_ORIGIN</td>
<td>Not currently used; always returned null</td>
</tr>
</tbody>
</table>

The following example shows an excerpt from a stored procedure that uses `DhSQLException.getDiagnostics`.

```java
try {
   SQLIStatement insert_cust = new SQLIStatement ("INSERT INTO customer VALUES (1,2)");
} catch (DhSQLException e) {
   errstate = e.getDiagnostics (DhSQLException.RETURNED_SQLSTATE) ;
   errmesg = e.getDiagnostics (DhSQLException.MESSAGE_TEXT) ;
   .
   .
}
```

Stored procedures can also throw their own exceptions by instantiating a `DhSQLException` object and throwing the object when the procedure detects an error in execution. The conditions under which the procedure throws the exception object are completely dependent on the procedure.

The following example illustrates using the `DhSQLException` constructor to create an exception object called `excep`. It then throws the `excep` object under all conditions.

```java
CREATE PROCEDURE sp1_02()
BEGIN
   // raising exception
   DhSQLException excep =
      new DhSQLException(777,new String("Entered the tst02 procedure"));
   if (true)
      throw excep;
END
```

**Calling stored procedures from other stored procedures**

Stored procedures and triggers can call other stored procedures. Nesting procedures lets you take advantage of existing procedures. Instead of rewriting the code, procedures can simply issue `CALL` statements to the existing procedures.
Another use for nesting procedures is for assembling result sets generated by queries on different databases into a single result set. With this technique, the stored procedure processes multiple SELECT statements through multiple instances of the SQLCursor class. For each of the instances, the procedure uses the DhSQLResultSet class to add rows to the result set returned by the procedure.

**Stored procedure parameter requirements and usage**

When one stored procedure is calling another stored procedure, the following requirements must be met for using the three parameter types in order to properly allocate the SQLDA structure to the correct size:

- An **IN** parameter calls only the `SetParam` function
- An **OUT** parameter calls only the `RegisterOutParam` function
- An **INOUT** parameter calls both the `SetParam` and `RegisterOutParam` functions in any order

**INOUT and OUT parameters when one Java stored procedure calls another**

If an **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.

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.

These are the general steps to follow when calling one Java stored procedure from another:

1. Register **OUT** parameters in the calling stored procedure.
2. Declare Java variables in the snippet of the calling procedure.
3. Invoke the other stored procedure.
The following example illustrates the steps required for calling one Java stored procedure from another.

```java
create procedure lotusp(
    IN f1 char(50),
    INOUT f2 char(50),
    OUT f3 char(50)
) RESULT(f4 char(50))
BEGIN
    f2 = new String("new rising sun");
    f3 = new String("new rising lotus");
    SQLResultSet.set(1, new String("the fog - the snow - the ice"));
    SQLResultSet.insert();
END
commit work;
create procedure proc1()
BEGIN
    String inout_param = new String("sun");
    String out_param = new String();
    SQLCursor call_proc = new SQLCursor("call lotusp(?,?,?)");
    call_proc.setParam(1, new String("moon"));
    call_proc.registerOutParam(3, CHAR);
    // OR you can specify the optional scale parameter
    // call_proc.registerOutParam(3, CHAR, 15);
    call_proc.open();
inout_param = (String)call_proc.getParam(2, CHAR);
    out_param = (String)call_proc.getParam(3, CHAR);
    call_proc.close();
END
```

**Working with triggers**

Database triggers are not part of the SQL standard, but are supported in the OpenEdge environment. Triggers are a special type of stored procedure used to maintain database integrity by enforcing specific business logic.

Database triggers are supported using Java. The OpenEdge SQL Engine adds wrapper code around a trigger to create a Java class and method that is invoked when the trigger fires. When creating a database trigger, the compiled Java class generated is stored within the database as well as the original source.

Triggers are a special type of stored procedure used to maintain database integrity.

Like stored procedures, triggers also contain Java code (embedded in a `CREATE TRIGGER` statement) and use OpenEdge SQL Java classes. However, triggers are automatically invoked (fired) by certain SQL operations (an insert, update, or delete operation) on the trigger's target table.

This section provides a general description of triggers and discusses in detail where trigger procedures differ from stored procedures. Unless otherwise noted, the material in earlier sections of this chapter also applies to triggers.
# Creating triggers

Use the SQL `CREATE TRIGGER` statement to create a trigger. This is the syntax for the `CREATE TRIGGER` statement:

```sql
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
```

## Structure of triggers

Like a stored procedure, a trigger has a specification and a body.

The body of a trigger is the same as that of a stored procedure: `BEGIN` and `END` delimiters enclosing a Java snippet. The Java code in the snippet defines the triggered action that executes when the trigger is fired.

As with stored procedures, when it processes a `CREATE TRIGGER` statement, OpenEdge SQL adds wrapper code to create a Java class and method that is invoked when the trigger is fired.

The trigger specification, however, is different from a stored procedure specification. It contains the following elements:

- The `CREATE` clause specifies the name of the trigger. OpenEdge SQL stores the `CREATE TRIGGER` statement in the database under `trigname`. It also uses `trigname` in the name of the Java class that OpenEdge SQL declares to wrap around the Java snippet. The class name uses the format `username_trigname_TP`, where `username` is the user name of the database connection that issued the `CREATE TRIGGER` statement.

- The `BEFORE` or `AFTER` keywords specify the trigger action time: whether the triggered action implemented by `java_snippet` executes before or after the triggering `INSERT`, `UPDATE`, or `DELETE` statement.

- The `INSERT`, `DELETE`, or `UPDATE` keyword specifies which data modification command activates the trigger. If `UPDATE` is the trigger event, this clause can include an optional column list. Updates to any of the specified columns will activate the trigger. (Updates to other columns in the table will not activate the trigger.) If `UPDATE` is the triggering statement and does not include the optional column list, then the `UPDATE` statement must specify all the table columns in order to activate the trigger.

- The `ON table_name` clause specifies the table for which the specified trigger event activates the trigger. The `ON` clause cannot specify a view or a remote table.

- The optional `REFERENCING` clause is allowed only if the trigger also specifies the `FOR EACH ROW` clause. It 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.
• The **FOR EACH** clause specifies the frequency with which the triggered action implemented by
  java_snippet executes.

• **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.

• The **IMPORT** clause is the same as in stored procedures. It specifies standard Java classes to
  import.

The following example shows the elements of a trigger.

```
CREATE TRIGGER BUG_UPDATE_TRIGGER
  AFTER
  UPDATE OF STATUS REPORT, PRIORITY
  ON BUG_INFO
  REFERENCING OLDROW, NEWROW
  FOR EACH ROW
  IMPORT
  import java.sql.*;
BEGIN
  ...
END
```

---

### Triggers, stored procedures, and constraints

Triggers are identical to stored procedures in many respects. There are three main differences:

• Triggers are automatic. When the trigger event (an **INSERT**, **UPDATE**, or **DELETE** statement)
  affects the specified table (and, optionally in **UPDATE** operations, the specified columns), the
  Java code contained in the body of the trigger executes. Stored procedures, on the other hand,
  must be explicitly invoked by an application or another procedure.

• Triggers cannot have output parameters or a result set. Since triggers are automatic, there is
  no calling application to process any output they might generate. The practical consequence
  of this is that the Java code in the trigger body cannot invoke methods of the **DhSQLResultSet**
  class.

• Triggers have limited input parameters. The only possible input parameters for triggers are
  values of columns in the rows affected by the trigger event. If the trigger includes the
  **REFERENCING clause**, OpenEdge SQL passes the values (either as they existed in the database
  or are specified in the **INSERT** or **UPDATE** statement) of each row affected. The Java code in
  the trigger body can use those values in its processing by invoking the **getValue** method of the
  **OLDROW** and **NEWROW** objects.

The automatic nature of triggers makes them well suited for enforcing referential integrity. In this
regard they are like constraints, since both triggers and constraints can help ensure that a value
stored in the foreign key of a table must either be null or be equal to some value in the matching
unique or primary key of another table. However, triggers differ from constraints in the following
ways:

• Triggers are active, while constraints are passive. Constraints prevent updates that violate
  referential integrity, and triggers perform explicit actions in addition to the update operation.
Triggers can do much more than enforce referential integrity. Because they are passive, constraints are limited to preventing updates in a narrow set of conditions. Triggers are more flexible.

Typical uses for triggers

Typical uses for triggers include combinations of the following:

- **Cascading deletes** — A delete operation on one table causes additional rows to be deleted from other tables that are related to the first table by key values. This is an active way of enforcing referential integrity that a table constraint enforces passively.

- **Cascading updates** — An update operation on one table causes additional rows to be updated in other tables that are related to the first table by key values. These updates are commonly limited to the key fields themselves. This is an active way of enforcing referential integrity that a table constraint enforces passively.

- **Summation updates** — An update operation in one table causes an update operation in a row of another table. The second value is increased or decreased.

- **Automatic archiving** — A delete operation on one table creates an identical row in an archive table that is not otherwise used by the database.

OLDROW and NEWROW objects: passing values to triggers

The OLDROW and NEWROW objects allow SQL to pass row values as input parameters to the stored procedure in a trigger that executes once for each affected row. If the CREATE TRIGGER statement contains the REFERENCING clause, the SQL server implicitly instantiates an OLDROW or NEWROW object (or both, depending on the arguments to the REFERENCING clause) when it creates the Java class.

This allows the Java code in the snippet to use the getValue method of those objects to retrieve values of columns in rows affected by the trigger event and store them in procedure variables, and use the setValue method of those objects to set the values to be stored in the database before the trigger event. For example:

- The OLDROW object contains values of a row as it exists in the database before an update or delete operation. It is instantiated when triggers specify an UPDATE...REFERENCINGOLDROW or DELETE...REFERENCING OLDROW clause. It is meaningless and not available for insert operations. The getValue method is valid on OLDROW before or after an update or delete and the setValue method is not valid on OLDROW at all.

- The NEWROW object contains values of a row as specified in an INSERT or UPDATE statement. It is instantiated when triggers specify an UPDATE...REFERENCING NEWROW or INSERT...REFERENCING NEWROW clause. It is meaningless and not available for delete operations. The getValue method is valid on NEWROW before or after an update or insert and the setValue method is only valid on NEWROW before insert or update.

UPDATE is the only triggering statement that allows both NEWROW and OLDROW in the REFERENCING clause.
Triggers use the `OLDROW.getValue` and `NEWROW.getValue` methods to assign a value from a row being modified to a procedure variable. The format and arguments for `getValue` are the same as in other OpenEdge SQL Java classes. This is the syntax for `getValue`:

```
getValue ( col_num , sql_data_type ) ;
```

col_num

Specifies the integer column number of the affected row. `getValue` retrieves the value in the column denoted by `col_num`. 1 denotes the first column of the table that the trigger is for. 2 denotes the second, \( n \) denotes the \( n \)th.

sql_data_type

Specifies the corresponding SQL data type. For a complete list of appropriate data types, see the Mapping between SQL and Java data types table.

The following example shows an excerpt from a trigger that uses `getValue` to assign values from both `OLDROW` and `NEWROW` objects.

```java
CREATE TRIGGER BUG_UPDATE_TRIGGER
AFTER UPDATE OF STATUS, PRIORITY ON BUG_INFO
REFERENCING OLDROW, NEWROW
FOR EACH ROW
IMPORT
import java.sql.* ;
BEGIN
try
{
    // column number of STATUS is 10
    String old_status, new_status;
    old_status = (String) OLDROW.getValue(10, CHAR);
    new_status = (String) NEWROW.getValue(10, CHAR);
    if ((old_status.CompareTo("OPEN") == 0) &&
        (new_status.CompareTo("FIXED") == 0))
    {
        // If STATUS has changed from OPEN to FIXED
        // increment the bugs_fixed_cnt by 1 in the
        // row corresponding to current month
        // and current year
        SQLStatement update_stmt (" update BUG_STATUS set bugs_fixed_cnt = bugs_fixed_cnt + 1 "
        " where month = ? and year = ?"
    );
    .
    .
    .
```
Optimizing Query Performance

The OpenEdge SQL server consists of several components, including the query optimizer. The query optimizer maximizes the server’s efficiency by determining the quickest way to execute a statement to produce the exact data requested. For details, see the following topics:

- Understanding optimization
- Inspecting what the optimizer produces
- Affecting what the optimizer produces

Understanding optimization

The OpenEdge SQL Engine contains a query optimizer that analyzes SQL queries and produces a plan for how SQL should best execute the query. The plan contains information such as which tables to access, in what order, and with which indexes. To produce a good query plan, the optimizer analyzes the query and considers many methods for each query execution step.

For instance, a table of customer orders might have eight different indexes: for accessing orders, order number, customer number, order date, delivery date, suppliers, plant number, sales person, and by combinations of those attributes. For example:

```
SELECT a,b,c, FROM pub.Orders
    WHERE CustNum = 1234
    AND OrderDate = '01-04-2003'
    AND Supplier = 'Whittle Widgets';
```
Two candidate indexes might be `XCust_Num` and `XSupplier`. To choose one of these indexes, the optimizer estimates the cost to access data using that specific index. The optimizer measures cost in terms of time. The optimizer then chooses the least costly index. Index `XCust_Num`, for instance, might have an estimated cost of 25 milliseconds for the predicate `CustNum = 1234`, and the index `XSupplier` might have an estimated cost of 35 milliseconds for predicate `Supplier = 'Whittle Widgets'`.

Clearly, then, estimating costs as accurately as possible is crucial to choosing the best index, and for all other choices the optimizer makes. For database tables, the optimizer’s cost estimates are based on how the table is accessed, and on the number of rows it expects to access. To estimate the number of rows, the optimizer uses statistics which the database owner has created using the `SQL UPDATE STATISTICS` command. It also uses rules about the type of index considered, such as a unique index, and about the type of predicate (such as `=` or `BETWEEN`) used to access an index.

**How the query optimizer works**

The optimizer works not only on statements reading data, but also on statements writing data. For any SQL statement, there are many possible methods to compute results. The optimizer decides which methods to use, the order in which to apply the methods, and the characteristics of each method. The optimization model used by the OpenEdge SQL Engine is a synthesis of:

- **Decomposition** — Statements are broken into elementary pieces such as tables, columns, and predicates.

- **Relational algebra operations** — This includes operations such as project, restrict, join, and sort.

- **Composition** — Primitive operations, such as restrict or join, are composed into a sequence of steps.

- **Cost-based analysis and decision making** — Alternative operations are cost estimated, and the least costly operation is chosen.

- **Rule-based analysis and decision making** — Rules expressing proven, efficient statement execution methods determine how operations and their attributes are built and combined.

**Representing the statement as a query tree**

The query processor makes extensive use of a relational algebra tree representation to model and manipulate SQL queries. At various points within the tree, operations are performed on the data. Each operation is represented as a node in the tree. Nodes can have one or more expressions associated with them to specify columns, conditions, and calculations associated with the operation.

Some of the operators that might be present in the tree are:

- **Restrict** — Reduces the number of output rows by eliminating those that fail to satisfy some condition applied to the input. Restrict operators appear in the tree from `WHERE` clauses and `JOIN`es.

- **Project** — Reduces the number of output columns by eliminating columns not present in a project list. Projection operators appear in the tree from `SELECT` statements, from the list of columns needed for a table, and for aggregations such as `SUM`.

- **Join** — Combines two input tables into a single output table that contains some combination of rows from the inputs. Joins appear in the tree from the use of `FROM` clauses and from `JOIN` clauses.
• **Sort** — Changes the ordering of rows in an input table to produce an output table in the desired order.

• **Table** — Represents a table scan or an index scan, reading data from a given table by either its default index (table scan) or a specific index (index scan).

Leaf nodes of the tree are always references to database tables. The following figure illustrates a tree produced for the query.

**Figure 9: Query relational tree model**

![Query relational tree model]

This query lists the names and order dates for all customers whose orders were shipped on the same day the order was placed:

```
SELECT Name, OrderDate
FROM Customer, Order
WHERE Order."OrderDate" = Order."ShipDate";
```

**The statement parser**

The *statement parser*, a component of the SQL engine, performs the initial analysis of an SQL statement. It checks for correct syntax and transforms the statement from a character string into a query tree. The parser also performs several transformations of the query in order to simplify subsequent analysis and optimization steps. Among these transformations are translations of quantified predicates and taking into account references to views.

**Quantified predicates and other subqueries**

In OpenEdge SQL, subqueries are low-cost because they are folded into the query tree as joins. The parser translates subqueries, such as predicates preceded by `ANY` and `ALL`, to an equivalent form that does not contain these keywords. Usually, the new form is a join between the data in the subquery and the data in the remainder of the SQL statement.
Views

In OpenEdge SQL, views are low-cost because their definitions in terms of base tables are substituted into the query tree. The initial tree created by the parser treats views as though they were base tables. Before the query can be optimized, the view references must be resolved and applied to the tree. View resolution replaces each view reference with a subtree corresponding to the query expression found in the view definition.

Optimizer phases

The optimization process is divided into several phases. Some phases deal with internal infrastructure, such as minimizing data handling or temp-table usage. Others deal with significant cost factors and are straightforward to understand. Each phase addresses a specific type of optimization, as explained in the following sections.

The optimizer follows a cost-based model. In each stage, whenever multiple alternatives are available, the optimizer estimates the cost for each and selects the cheapest. The cost computation takes into account:

- Cost metrics for operations performed by the SQL engine’s storage manager and query processor components
- Index definitions
- Properties of join algorithms
- Column selectivity
- Filter factors
- Table cardinality

The following sections provide details on the optimization phases.

Early evaluation of constant expressions

This is the first phase of optimization (introduced from 11.6). In this phase query tree is iterated over recursively to find the constant expressions. If the expression can be evaluated then values for the constants are peeked and the expression is evaluated. The expression is replaced with the result of the constant expression. This reduces execution time considerably as it eliminates the evaluation of expression for each row during the execution.

The following example shows evaluation of constant expression in peek_evaluate_consts phase. Assuming that the Where condition is as follows:

```sql
Select * from sales where ('yearly' = 'yearly' and sales_date between '01-01-2001' and '10-10-2012')
OR ('yearly' = 'Monthly' and sales_date between '01-01-2002' and '10-10-2012')
OR ('yearly' = 'daily' and sales_date between '01-01-2009' and '10-10-2012')
```
After evaluating the constant expressions, the Where condition is transformed as shown below.

```sql
Select * from sales where sales_date between '01-01-2001' and '10-10-2012';
```

**Note:** If any of the constant expressions are evaluated for a query, the query plan is marked as not re-usable. Due to the evaluation of constant expressions the query plan has become valid only for a set of constants hence it cannot be re-used for another set of constant values.

---

**GROUP BY optimization**

GROUP BY optimization allows the SQL engine to select index scan and to perform STREAM aggregation for GROUP BY columns. The two techniques of GROUP BY optimizations are as follows:

- 1. Re-order GROUP BY columns
- 2. Considering Equi- Constant Predicates

**Re-order GROUP BY columns**

The SQL engine checks whether the GROUP-BY columns are the leading prefix components of an index, by using re-ordering of GROUP BY column. If they are the leading prefix components of an index, the SQL engine performs stream aggregation for these queries. If this optimization technique is not used, the SQL engine optimizer may choose hash aggregation. An index can be selected to perform GROUP BY c2, c1 if the index is defined on c1 and c2 as leading components. The SQL engine chooses stream aggregation as follows:

```sql
Create index idx2 on test1(c1,c2);
Select c1,c2,sum(c1), sum(c2) from test1 GROUP BY c2,c1;
```

**Considering Equi- Constant Predicates**

Using the Equi-Constant Predicates optimization technique, sql engine recognizes leading index prefix components with equi-constant predicates and tries to select index scan if the remaining index components (may include leading components also) are specified in group-by columns. For example:

```sql
Create index idx1 on test1(c1,c2, c3)
Select c2,c3 from Test1 where c1=100 GROUP BY c2,c3;
Select c2,c3 from Test1 where c1=100 GROUP BY c3,c2;
```

In these two queries, index idx1 can be chosen with the index predicate c1=100. Since the rows are in order and the Considering Constant Predicates optimization technique is applied, STREAM aggregation is chosen. If this optimization technique is not applied, the SQL optimizer will pick the index scan but will choose the HASH aggregation.
Pushing restrict operations close to the data origin

This stage consists of moving restrict operators as far down the query tree as possible. This reduces
the number of tuples moving up the tree for further processing and minimizes the amount of data
handled. When restrict operations on a join node cannot be moved below the join node, they are
set as join conditions. When multiple predicates are moved down the tree to the same relative
position, they are reassembled into a single restrict operation, as shown in the following example.

```
SELECT Name FROM Employee
WHERE Salary > 4000 AND Salary <= 6000
AND Employee.DeptNum = Department.DeptNum;
```

The optimizer takes the input tree and transforms it as shown below. The restrictions Salary >
4000 and Salary <= 6000 are moved down the tree, below the join node, since they apply
to a single table. The restriction Employee.DeptNum = Department.DeptNum, applying to
two tables, stays above the join node.

Using indexes for restrictions

This optimization phase consists of recognizing those cases where an existing index can be used
to evaluate a restriction and converting a table scan into an index bracket or set of contiguous
index entries. An index bracket is extremely effective in limiting the number of rows that must be
processed.

Choosing the best index

To choose an index, the optimizer performs several stages. These stages determine whether an
index can be used to process a restrict operation and, if there are multiple indexes to choose from,
which index will be used:

- Transform expressions in predicates
- Generate a list of candidate indexes
- Select an index to use

Predicate expressions

When the predicates of an SQL statement use the OR logical operator to combine expressions
that compare the same column with a constant, the optimizer converts these expressions to a
single IN predicate. The purpose of these transformations is, where possible, to combine multiple
predicates into a single predicate for simpler evaluation in this and later stages.

Similarly, a LIKE predicate on an index key, where the LIKE pattern has a prefix of fixed characters,
is converted to a BETWEEN predicate.

Generating candidate indexes

For every predicate in an SQL statement, the optimizer checks to see if there are indexes that
include the columns referenced in the predicate.
Once the optimizer knows which indexes exist on the relevant tables, it generates a list of all the possible index predicates that could be used. For each predicate for which there is an index, the optimizer checks whether:

- The predicate’s relational operator (=, <, <=, etc.) can be performed by the index
- The index has multiple components, and if so, that the key components with predicates form a sequence of leading components of the index

### Selecting an index

When the list of candidate index predicates has been determined, the optimizer selects which, if any, it will use for an index scan operation.

This selection is cost-based. The optimizer computes the cost for each of the index candidates and the cost for a table scan using the default index. The candidate with the lowest cost is chosen.

### Providing index hints

You can specify an index for each table in the `FROM` clause of a `SELECT` query. For example:

```sql
SELECT column_list FROM table_name [[ AS ]table_alias] [ WITH (INDEX (index_val)) ]...WHERE ...
```

`index_val` is a string that indicates the name of the index.

If a candidate plan is generated with the specified index, the optimizer will use it. If the optimizer is unable to generate a candidate plan with the specified index, it ignores the hint.

### Join optimization

There are two distinct optimization tasks done as part of the join optimization stage:

- **Join ordering** — Determines the most efficient order for performing joins among adjacent join nodes
- **Join algorithm selection** — Determines the best join algorithm to use for each join node

### Determining join order among adjacent join nodes

After identifying a set of adjacent join nodes, the optimizer uses the available statistics to estimate the cardinality (the number of rows in the table or intermediate result) and selectivity (percentage of rows a predicate returns) for each subtree of the join nodes. It then uses the following criteria to determine the join order:

- The subtree with the lowest estimated cardinality is taken first. The SQL engine’s cost manager estimates the cardinality of each subtree by multiplying table cardinality by the selectivity of the predicates applied to the table.
- The subtree that has the lowest estimated join cardinality (number of output rows produced by a join with the first subtree) is taken second. When determining join cardinality, the optimizer considers whether there is a join condition between the two subtrees. It gives preference to subtree pairs that have join conditions.
• The subtree with the next lowest estimated join cardinality is taken next, and so on.

Choosing the join algorithm

Once the join order has been established, each join node is analyzed to select from among the following algorithms:

• Augmented nested loop (ANL) join
• Merge join
• Nested loop join

The optimizer generates, when possible, candidates for each algorithm. For each join node, candidates are generated by:

• Checking whether the algorithm's requirements are satisfied. For example, the ANL join needs an index on one of the join columns.
• Assuming the algorithm is usable, when multiple predicates reference the two tables being joined, choosing a predicate (or set of predicates) with the lowest cost.

Once a set of candidates exists, the optimizer selects the least costly candidate.

Augmented nested loop join

The augmented nested loop (ANL) is by far the most common join method. An augmented nested loop join is performed by doing a scan over the left subtree and for each row in it, performing an index bracket scan on a portion of the right subtree. The right subtree is read as many times as there are rows in the left subtree.

To be a candidate for an ANL join, the subtree pair for a join node must meet the following criteria:

• There must be an index or indexes defined on the join columns for the table in the right subtree.
• No other scan on that index has already been set.

When an ANL join is possible on several indexes, the least-cost index is chosen.

When there is an index defined on the left subtree's table instead of on the right, the optimizer analyzes the cost of swapping the subtrees to make an ANL join possible.

When neither subtree's table has an index defined on the join column, the optimizer analyzes the cost of creating a dynamic index on one or both of the subtrees.

Merge join

A merge join is performed by opening simultaneous scans on both the left and right subtrees. Each row that satisfies the join condition is output by the join algorithm. Depending upon the result of the join column comparison, either the left or right scan pointer is advanced. The left and right subtrees are each read once. A merge join is almost never chosen because its cost invariably exceeds an ANL join.

Nested loop join

A nested loop join is performed by doing a scan over the left subtree and for each row in it performing a full scan of the right subtree.
This is the default join algorithm, which can be used for any join. However, it is usually less efficient than the other methods. Usually, either an existing index or a dynamic index, used in an ANL join, will cost much less. Occasionally, when subtree cardinalities are very low, possibly because of index bracketing, nested loop will be the method with the least cost.

**Sort optimization**

The optimizer performs two optimizations designed to avoid sort operations. The first optimization is to eliminate redundant sorts. The second optimization is to convert table scans into index bracket scans.

**Eliminating redundant sorts**

The optimizer checks whether the query tree contains unnecessary sort nodes. For example, when an SQL statement contains both a `GROUP BY` clause and an `ORDER BY` clause that refers to the same column, at most one sort is needed.

A sort node is also redundant when the immediate descendant node of the sort node is an index bracket scan on the sort column. That is, the sort is redundant when the data input to the sort was read using an index with the needed sort order.

**Converting table scans to index bracket scans**

When a leaf node of a subtree is a table scan, the optimizer checks whether any indexes that exist on the table match the sort columns. If so, it analyzes the cost of each possible index bracket scan and compares the least of those with the sum of the cost of the table scan and sort operation.

If the analysis shows an index bracket scan as having less cost than the table scan and sort operation, the optimizer converts the table scan to the index bracket scan and removes the sort node.

**Indexes to evaluate MAX/MIN functions**

This stage of optimization examines subtrees that contain `MIN` and `MAX` aggregate functions. The optimizer checks if any index on the table matches the column specified in the function. If so, it replaces the table scan at the leaf node with an index bracket scan.

The index bracket scan looks up the first or last value of the relevant index key. The first and last values represent the `MIN` and `MAX` values, respectively, for ascending indexes, and the `MAX` and `MIN` values for descending indexes.

Evaluating aggregate functions without fetching the table rows is not possible for indexed character columns because index entries for character data contain the "sort-weight" form of the column value, not the actual column value.

**Index bracket scan optimization**

This stage checks whether a table scan can be replaced by an index bracket scan. This is possible when a subtree meets the following criteria:

- The subtree has a table scan as its leaf node.
- An index that includes all the projected columns as part of the index key exists.
• Analysis indicates the index bracket scan cost is less than the table scan cost. This is nearly always the case.

Inspecting what the optimizer produces

The OpenEdge SQL Engine provides the capability to examine the query trees produced by the optimizer, which it has used to actually execute a query. The engine’s internal cache of recent query plans is available as a series of text columns in a virtual system table called _Sql_Qplan. Note that like all other virtual system tables, the data in this table do not actually exist as a table. The table rows are constructed on demand from data structures in the server’s memory.

The _Sql_Qplan virtual system table

The virtual system table (VST) called _Sql_Qplan contains query plans for the last 10 queries that were executed. The CREATE TABLE statement shows you its definition in the following example.

```sql
CREATE TABLE "_Sql_Qplan" (
    "_Pnumber" INTEGER NOT NULL,
    "_Ptype" INTEGER NOT NULL,
    "_Dtype" INTEGER NOT NULL,
    "_Description" VARCHAR (255) NOT NULL,
    "_Dseq" INTEGER NOT NULL
);
```

The following table offers a description for the columns in the _Sql_Qplan Virtual System Table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Pnumber</td>
<td>Query plan number, in descending order. Has no inherent significance. It merely groups together all the rows for a query plan.</td>
</tr>
<tr>
<td>_Ptype</td>
<td>Query plan type. Is &gt; 0 for an application query and &lt; 0 for an internally generated query.</td>
</tr>
<tr>
<td>_DType</td>
<td>Not used. In the future, this column will provide descriptive information about the plan.</td>
</tr>
<tr>
<td>_Description</td>
<td>Contains a description of part of the query plan.</td>
</tr>
<tr>
<td>_Dseq</td>
<td>Query plan row number, ordering the rows describing the plan for a particular table.</td>
</tr>
</tbody>
</table>
Affecting what the optimizer produces

The query plan produced by the optimizer is determined by optimizer algorithms and by:

• The SQL statement itself, especially its predicates.
• Information from the schema, such as the definition of tables used by the statement. Index definitions are especially important.
• Information from the statistics tables about the cardinalities of the tables used, about data frequency within indexes, and about data distribution within columns.

The SQL statement itself is of utmost importance. The predicates from the WHERE clause form the basis for estimating how much data will be read for each table, and how much data each join will produce. Supplying as many precise and accurate predicates as possible is vital to good optimization.

Working with the UPDATE STATISTICS command

Indexes are the database's fast-access path to a table's data. SQL will choose an index when its key components have matching predicates and it is estimated as the least costly access path. An index is usually regarded as least costly when its predicates select the least amount of data.

Cost estimates are most accurate when SQL has good statistics with which to work. Statistics are created by the UPDATE STATISTICS command and its various options.

The most basic statistic is table cardinality. This is vital to join order determination and to estimating the number of rows that will be selected by a set of predicates.

Column statistics give information about the distribution of data within a column. The column statistics are a sample of data across the entire range of a column's data. They especially enable SQL to estimate costs for range predicates, such as BETWEEN, and ">" and "<". They are useful for equality predicates also, but are much less precise than index statistics.

When there are no statistics at all, the optimizer uses certain default values for the selectivity of a column. Selectivity expresses what fraction of a column's data will be selected by some set of predicates.

SQL use of index statistics

SQL uses new index statistics whenever they exist. Statistics are not automatically created. Index statistics are used to estimate costs for "=" predicates and for joins on "=" predicates. In these situations, the column selectivity statistics will no longer be used. Statistic counts are used to estimate the number of rows that will be read using an index for a given set of key component values, as shown:

#rows = table cardinality / index count of unique values
The statistic count used depends on the number of key components with corresponding predicates. If an index has more than three key components and the query needs a count for a key subset without an explicit count, SQL uses interpolation to produce an estimated count. For instance, suppose an index has six components, and there are predicates for the first three components only. Then SQL will use the new index counts to interpolate an estimate for three components. SQL uses linear interpolation. The linear interpolation, in effect, draws a straight line through statistics’ counts, and estimates where the count would be for only three components. This is an estimation, of course, and subject to some inaccuracy.

In the majority of cases, the new index statistics will outperform the older statistics and their associated rules.

This means it is much more likely that queries will use the best index. Therefore, whenever there is a problem with index performance, or with choice of an index by the optimizer, create new index statistics. Note that algorithms used for cost estimating with new index statistics assume relatively uniform data distribution. If data distribution is highly skewed so that some key values have more instances than others, cost estimates will be less accurate.

### Updating index statistics

The `UPDATE STATISTICS` command uses the following syntax:

```
UPDATE ([ TABLE | INDEX | [ ALL ] COLUMN ] STATISTICS [ AND ]) ... [ FOR table_name[ FOR PARTITIONS partition_name, [partition_name_1, ... ]]]
```

The following example demonstrates the use of the `UPDATE STATISTICS` statement for a single table:

```
UPDATE INDEX STATISTICS FOR Employee;
```

The following example updates statistics for indexes and columns for a single employee:

```
UPDATE TABLE STATISTICS AND INDEX STATISTICS AND COLUMN STATISTICS FOR Employee;
```

The following example updates index statistics of partitions `USA_Customer`, `EUROPE_Customer` and `ASIA_Customer` for table `Customer` by using `FOR PARTITIONS` which updates only the local index statistics and not the global index statistics:

To create the new index statistics for all tables in a database, simply use the statement shown in the following example.

```
Table 21: UPDATE INDEX STATISTICS statement
```

```
UPDATE INDEX STATISTICS;
```

### Notes
• To create the new index statistics, SQL makes one pass over each index, reading every index entry and counting unique values. This is usually a CPU-intensive operation. When a table has many indexes, this operation can take quite a bit more time than the default UPDATE STATISTICS.

• UPDATE STATISTICS does not lock user data. It only locks the output statistics rows (and also acquires a shared lock on the schema). This means that user-level transactions can freely run concurrently with UPDATE STATISTICS.

• 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 executing:

  • UPDATE TABLE STATISTICS AND INDEX STATISTICS AND ALL COLUMN STATISTICS;
Index

A
ABL brokers 132
ALTER TABLE statement 72
ALTER TENANT statement 86
Altering tables 72
Application performance using stored procedures 182
Architecture
  client/server 20
  J2EE 174
  JDBC 24
  JTA 174
  multi-threaded 21
  ODBC 37
Audit Administrator 61
Automatic archiving 206
Auxiliary databases
  connecting 171
  disconnecting 171

C
Calling stored procedures
  from JDBC 189
  from ODBC 188
Candidate keys 95
Cascading deletes 206
Cascading privileges 63
Cascading updates 206
Catalogs
  disconnecting from 166
  in multi-database queries 162
CLASSPATH 26
Clients
  JDBC 23
  ODBC 37
Column authorization system table 63
Column widths 139
Columns 137
COMMIT statement 146
CONNECT AS CATALOG statement 166
Connecting
  database servers 132
  multi-databases 165
Constraints
  check 93
  column-level check 93
  foreign key 96
  integrity 92
  primary key 95
  referential 96
  table-level check 94
Conventional identifiers 138
Crash recovery
  JTA transactions 178
CREATE INDEX statement 76
CREATE PROCEDURE statement 182, 185
CREATE SUPER-TENANT statement 86
CREATE TABLE statement 68
CREATE TENANT statement 85
CREATE TRIGGER
  in procedures 204
  syntax 204
CREATE USER statement 57
CREATE VIEW statement 79–80
Creating
  domain 84
  indexes 76
  stored procedures 183
  tables 68
  users in the _User table 57
  views 79–80

D
Data Control Language 53
Data Definition Language 67
Data integrity 83, 85
Data Manipulation Language 113
Data sources
  adding 39
  administration 39
  configuring 39
  configuring for ODBC 39
  ODBC UNIX configuration 44
  ODBC Windows configuration 39
Data type compatibility 139
Data types
  conversion between SQL and Java 193
  mapping between SQL and Java 193
Database
  connecting from Java application 28
  connecting using SQL Explorer 27
  connecting with JDBC driver 27
  connection problems 28
  multiple connections 165
  Progress 4GL objects 137
  SQL objects 137
  structure 136

OpenEdge Data Management: SQL Development 223
G

getValue
  definition 191
getValue method 191
GRANT RESOURCE statement 61
GRANT statement 60
Granting
  public access 63
  resource privileges 61
  update privileges 62

I

Identifiers
  conventional 138
  delimited 138

Import clauses 186

Index statistics
  overview 219
  updating 220

Indexes
  creating 76
  dropping 78
  system catalogs 122
  using 121

Inner joins 122

Input parameters
  in stored procedures 192

Insert statements 116

Inserting data 116

Internet protocol support 25

Invoking stored procedures 188

Isolation levels
  ABL 142
  dirty read 147
  nonrepeatable read 147
  phantom read 148
  setting 148

J

J2EE 173

Java
  application programming interface 23
  database connection 28
  methods 193
  native interface 24
  source text 183
  using with stored procedures 182
  wrapper types 193

Java classes
  SQLCursor 190
  SQLStatement 190
  summary 185

Java Database Connectivity 23
Java snippets 186
Java Transaction API 173
Java Virtual Machine 183
JDBC
  API support 25
  architecture 24
  client 23
  components 24
  driver 24
  files 24
  invoking stored procedures 189
JDBC Driver
  architecture 24
  components 24
  files and locations 24
  overview 24
jdbc.jar file 24
JdbcProgress file 24
Join operations 122
Join optimization 215
JTA
  data resource planning 179
  distributed transaction process 177
  monitoring transactions 180
  role in J2EE 173

L
Library path 26
Loading
  ABL 104
Lock modes
  overview 149
Locking
  ABL 142
    acquisition 151
    database performance 153
    hints 152
    levels 150
    lock modes 149
    record locks 152
    schema locks 151
    table locks 152
Logon validation 135

M
Modifying
  stored procedures 190
Multi-database queries
  limitations 164
  overview 161
  process 161
  working with catalogs 162
Multiple-database queries
  using properties files 167

Multithreaded architecture 21

N
NEWROW objects 206
Nonrepeatable read 147

O
Objects
  NEWROW 206
  OLDROW 206
ODBC
  applications 37
  architecture 37
  connection testing in UNIX 48
  connection testing in Windows 39
  data source administrator 39
  driver manager 38
  driver overview 37
  driver set up 39
  invoking stored procedures 188
ODBC driver 37
ODBC.INI tags 46
OLDROW objects 206
OpenEdge SQL
  client/server architecture 19
  engine 19
  interaction with Java 182
  overview 19
OpenEdge SQL brokers 132
Optimization
  affecting output 219
  inspecting 218
  joins 215
  methods 210
  phases 212
  sorting 217
Outer joins 123
Output parameters
  in stored procedures 192

P
Parameter declarations 186
Phantom read 148
Primary key 95
Privileges
  database-wide 60
  granting 58
  revoking 63
  table-specific 60
  verifying 63
Procedure result set declarations 186
PROMON utility 153
Properties files
creating 167
using with multi-database queries 167
PUB schema 113

Q
Query optimizer 210

R
READPAST 152
Reference cycle
inserting rows 98
single table 98
table 97
Referential constraints 96
Resources
granting privileges 61
Restricted privileges 63
REVOKE statement 63
Revoking privileges 63
ROLLBACK statement 146
Rows 137

S
Schemas 137
Security
OpenEdge SQL 54
using stored procedures 190
SELECT statement 113
Selecting data 113
SET CATALOG statement 165
setParam
definition 191
syntax 191
SHOW CATALOG statement 166
SHOW TENANT statement 88
Single table reference cycle 97
SQL Engine
using stored procedures 183
SQL Explorer 27
SQL statements
executing 194
immediate execution 195
multi-database queries 165
prepared execution 195
SQL utilities 99
SQLDUMP Utility 99
SQLLOAD Utility 104
SQLSchema Utility 109
SSL 134
Statement parser 211

Statements
ALTER TABLE 72
ALTER USER 57, 59
COMMIT 146
CONNECT AS CATALOG 166
CREATE PROCEDURE 182, 185
CREATE TABLE 68
CREATE TRIGGER 204
CREATE VIEW 79
DISCONNECT CATALOG 167
DROP TABLE 76
DROP VIEW 79
GRANT 61
GRANT RESOURCE 61
INSERT 165–167
REVOKE 63
ROLLBACK 146
SET CATALOG 166
SHOW CATALOGS 166
UPDATE STATISTICS 220

Stored procedures
advantages 182
basics 185
calling 183
creating 182
deleting 190
enabling on 64-bit platforms 182
executing 183
handling errors 200
handling NULL values 198
input and output parameters 192
invoking 188
Java definition 182
jdk requirements 181
modifying 190
passing values 191
result sets 197
retrieving data 191
security 190
structure 186
using 185
writing 188

Summation updates 206

Syntax
ALTER TABLE 72
ALTER USER 57, 59
CALL to stored procedure (JDBC) 189
CALL to stored procedure (ODBC) 188
COMMIT 146
CONNECT AS CATALOG 166
CREATE INDEX 76
CREATE PROCEDURE 185
CREATE TABLE 68
CREATE USER 57
DELETE 121, 126
DISCONNECT CATALOG 167
Syntax (continued)
- DROP INDEX 78
- DROP TABLE 76
- getValue 192
- GRANT 61
- INSERT 118, 165–167
- joins 122
- REVOKE 63
- ROLLBACK 147
- SELECT 113
- SET CATALOG 165
- setParam 191
- SHOW CATALOGS 166
- UPDATE STATISTICS 220

System tables
- column authorization 63
- database authorization 63
- indexes 122
- table authorization 63

T
- Table authorization system table 63
- Table reference cycle 97

Tables
- altering 72
- creating 68
- dropping 76

Transaction control 145

Transactions
- committing 146
- conventional versus JTA 178
- JTA 177
- JTA and crash recovery 178
- locking 149
- rolling back 146
- with isolation levels 147

Triggers
- and stored procedures 205
- creating 204
- overview 203

Triggers (continued)
- passing values 206
- setting up OpenEdge SQL 181
- structure 204
- types 206
- using with ABL 142
- with constraints 205

Type 2 driver 24

U

UNIX
- ODBC configuration 44
- testing ODBC connections 48

UPDATE STATISTICS command 219

User accounts 134

User privileges 134

Users
- creating 56

Utilities
- SQLDUMP 99
- SQLLOAD 99
- SQLSCHEMA 99

W

WebSpeed Messenger Application 155, 157, 159

Windows
- adding new data sources 39
- JDBC environment variables 26
- ODBC administrator 39
- testing ODBC connections 39

Wrapper code 186

X

XAConnection 175
XDataSource 175
XAResource 176
XAResource methods 176