OpenEdge Data Management: SQL Development
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>1–1</td>
</tr>
<tr>
<td><strong>1. Introduction</strong></td>
<td>1–1</td>
</tr>
<tr>
<td>An overview of OpenEdge SQL</td>
<td>1–2</td>
</tr>
<tr>
<td>OpenEdge SQL client/server architecture</td>
<td>1–3</td>
</tr>
<tr>
<td>Multi-threaded architecture</td>
<td>1–4</td>
</tr>
<tr>
<td><strong>2. JDBC Client</strong></td>
<td>2–1</td>
</tr>
<tr>
<td>Introduction to the JDBC client</td>
<td>2–2</td>
</tr>
<tr>
<td>JDBC architecture</td>
<td>2–2</td>
</tr>
<tr>
<td>JDBC components</td>
<td>2–3</td>
</tr>
<tr>
<td>JDBC API support</td>
<td>2–3</td>
</tr>
<tr>
<td>Internet Protocol Support</td>
<td>2–4</td>
</tr>
<tr>
<td>Setting environment variables</td>
<td>2–5</td>
</tr>
<tr>
<td>Setting environment variables in a character environment</td>
<td>2–5</td>
</tr>
<tr>
<td>Setting environment variables in Windows</td>
<td>2–5</td>
</tr>
<tr>
<td>Connecting to an OpenEdge database with a JDBC driver</td>
<td>2–6</td>
</tr>
<tr>
<td>Connecting using SQL Explorer</td>
<td>2–6</td>
</tr>
<tr>
<td>Connecting from a Java application using a URL</td>
<td>2–7</td>
</tr>
<tr>
<td>Connecting from a Java application using a data source</td>
<td>2–11</td>
</tr>
<tr>
<td>Enabling encryption</td>
<td>2–11</td>
</tr>
<tr>
<td>JDBC connection parameters</td>
<td>2–13</td>
</tr>
<tr>
<td>Troubleshooting database connection problems</td>
<td>2–14</td>
</tr>
<tr>
<td><strong>3. ODBC Client</strong></td>
<td>3–1</td>
</tr>
<tr>
<td>Overview of ODBC</td>
<td>3–2</td>
</tr>
<tr>
<td>ODBC architecture</td>
<td>3–2</td>
</tr>
<tr>
<td>Configuring data sources</td>
<td>3–4</td>
</tr>
<tr>
<td>Internet Protocol support</td>
<td>3–4</td>
</tr>
<tr>
<td>Configuring Windows clients</td>
<td>3–4</td>
</tr>
<tr>
<td>Enabling encryption</td>
<td>3–8</td>
</tr>
<tr>
<td>Configuring UNIX clients</td>
<td>3–9</td>
</tr>
<tr>
<td>Definitions of ODBC.INI tags</td>
<td>3–11</td>
</tr>
<tr>
<td>Testing your ODBC connection on UNIX</td>
<td>3–13</td>
</tr>
</tbody>
</table>
## 4. Data Control Language and Security

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working with database security</td>
<td>4–1</td>
</tr>
<tr>
<td>Comparing OpenEdge SQL and ABL security</td>
<td>4–2</td>
</tr>
<tr>
<td>Comparing authentication and authorization</td>
<td>4–3</td>
</tr>
<tr>
<td>Creating users</td>
<td>4–4</td>
</tr>
<tr>
<td>Creating database administrators</td>
<td>4–4</td>
</tr>
<tr>
<td>Creating users</td>
<td>4–5</td>
</tr>
<tr>
<td>Granting privileges</td>
<td>4–6</td>
</tr>
<tr>
<td>Privilege basics</td>
<td>4–6</td>
</tr>
<tr>
<td>GRANT statement</td>
<td>4–6</td>
</tr>
<tr>
<td>Verifying privileges</td>
<td>4–9</td>
</tr>
<tr>
<td>Revoking privileges</td>
<td>4–10</td>
</tr>
</tbody>
</table>

## 5. OpenEdge SQL Data Definition Language

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Data Definition Language statements</td>
<td>5–1</td>
</tr>
<tr>
<td>Working with tables</td>
<td>5–2</td>
</tr>
<tr>
<td>Working with indexes</td>
<td>5–3</td>
</tr>
<tr>
<td>Working with views</td>
<td>5–7</td>
</tr>
<tr>
<td>Working with sequences</td>
<td>5–8</td>
</tr>
<tr>
<td>Maintaining data integrity</td>
<td>5–9</td>
</tr>
<tr>
<td>Need for integrity constraints</td>
<td>5–12</td>
</tr>
<tr>
<td>Types of integrity constraints</td>
<td>5–13</td>
</tr>
<tr>
<td>Referential constraints</td>
<td>5–16</td>
</tr>
<tr>
<td>Handling cycles in referential integrity</td>
<td>5–18</td>
</tr>
<tr>
<td>Working with SQL utilities</td>
<td>5–20</td>
</tr>
<tr>
<td>Using the SQLDUMP utility</td>
<td>5–20</td>
</tr>
<tr>
<td>Using the SQLLOAD utility</td>
<td>5–23</td>
</tr>
<tr>
<td>Using the SQLSCHEMA utility</td>
<td>5–25</td>
</tr>
</tbody>
</table>

## 6. OpenEdge SQL Data Manipulation Language

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using Data Manipulation Language statements</td>
<td>6–1</td>
</tr>
<tr>
<td>Using the SELECT statement</td>
<td>6–2</td>
</tr>
<tr>
<td>Using the INSERT statement</td>
<td>6–3</td>
</tr>
<tr>
<td>Using the UPDATE statement</td>
<td>6–4</td>
</tr>
<tr>
<td>Using the DELETE statement</td>
<td>6–5</td>
</tr>
<tr>
<td>Using indexes</td>
<td>6–6</td>
</tr>
<tr>
<td>Index system catalog tables</td>
<td>6–6</td>
</tr>
<tr>
<td>Working with join operations</td>
<td>6–7</td>
</tr>
<tr>
<td>Using inner joins</td>
<td>6–7</td>
</tr>
<tr>
<td>Using outer joins</td>
<td>6–8</td>
</tr>
</tbody>
</table>

## 7. OpenEdge SQL and Advanced Business Language Interoperability

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managing Open Edge SQL and ABL database connections</td>
<td>7–1</td>
</tr>
<tr>
<td>Determining database server requirements</td>
<td>7–2</td>
</tr>
<tr>
<td>Starting SQL and ABL brokers</td>
<td>7–2</td>
</tr>
<tr>
<td>Establishing user accounts and assigning privileges</td>
<td>7–5</td>
</tr>
<tr>
<td>Using authentication</td>
<td>7–5</td>
</tr>
<tr>
<td>Assigning privileges</td>
<td>7–5</td>
</tr>
<tr>
<td>ABL and OpenEdge SQL interaction in an OpenEdge application</td>
<td>7–7</td>
</tr>
<tr>
<td>Comparing ABL and OpenEdge SQL</td>
<td>7–7</td>
</tr>
<tr>
<td>Understanding OpenEdge SQL database structure</td>
<td>7–7</td>
</tr>
<tr>
<td>Comparing OpenEdge SQL and ABL database objects</td>
<td>7–8</td>
</tr>
<tr>
<td>Naming objects for OpenEdge SQL and ABL databases</td>
<td>7–9</td>
</tr>
</tbody>
</table>
8. Data Control Language and Transaction Behavior. ............... 8–1
   Working with transaction control ............................... 8–2
      COMMIT statement ........................................ 8–2
      ROLLBACK statement ...................................... 8–2
   Transactions and isolation levels ............................ 8–4
      Setting isolation levels .................................. 8–5
   Understanding transactions and locking .......................... 8–7
      Lock modes ............................................... 8–7
      How lock levels and lock modes interact ................. 8–8
      Understanding lock acquisition .......................... 8–9
   Enhancing performance with locking hints ..................... 8–10
      The READPAST locking hint .............................. 8–10
   Monitoring locking and database performance .................. 8–12
   Online schema changes ...................................... 8–13

9. Performing Multi-database Queries ............................... 9–1
   Multi-database query overview .................................. 9–2
      The process of multi-database queries ....................... 9–2
   Working with catalogs in multi-database queries .......... 9–3
   Connecting to multiple databases ............................ 9–6
      Connecting to multiple databases using SQL commands. .... 9–6
   Specifying a database default catalog ...................... 9–6
   Determining catalog availability ................................ 9–7
   Using the CONNECT AS CATALOG statement .................. 9–8
   Disconnecting from catalogs .................................. 9–8
   Using the DISCONNECT CATALOG statement .................. 9–8
   Using properties files to enable multiple database connections 9–8
   An example of a multi-database query .......................... 9–14
      Connecting to an auxiliary database ....................... 9–14
      Performing a multi-database query ......................... 9–14
      Disconnecting an auxiliary database ...................... 9–14

10. Working with JTA Transactions ................................ 10–1
    JTA’s role in J2EE ........................................... 10–2
    Understanding JTA architecture .............................. 10–3
    Understanding application interfaces ....................... 10–4
       XDataSource ............................................. 10–4
       XAConnection ............................................ 10–4
       XAResource ............................................. 10–4
       XAResource methods ..................................... 10–4
    JTA and the distributed transaction process .................. 10–6
       JTA transactions and two-phase commit protocol ........ 10–6
       JTA transactions and conventional transactions ........ 10–8
       JTA transactions and crash recovery ..................... 10–8
    Planning for JTA transaction support ......................... 10–9
       JTA transactions and database resource planning ........ 10–9
       Enabling JTA support ..................................... 10–10
       Disabling JTA support ..................................... 10–10
       Monitoring JTA transactions ............................. 10–10
11. Stored Procedures and Triggers ...................................... 11–1
   Setting up OpenEdge SQL for stored procedures and triggers ........ 11–2
      Enabling stored procedures on 64-bit platform databases ......... 11–2
   Basics of Java stored procedures .................................... 11–3
      Advantages of stored procedures .................................. 11–3
      How OpenEdge SQL interacts with Java .............................. 11–3
   Stored procedure fundamentals ....................................... 11–7
      Java snippet ............................................................ 11–7
   Writing stored procedures ............................................. 11–10
      Using the OpenEdge SQL Java classes .............................. 11–12
   Working with triggers .................................................. 11–27
      Creating triggers ...................................................... 11–27
      Structure of triggers ................................................ 11–28
      Triggers, stored procedures, and constraints ....................... 11–29
   Typical uses for triggers .............................................. 11–30
      OLDROW and NEWROW objects: passing values to triggers ....... 11–30

12. Optimizing Query Performance ....................................... 12–1
   Understanding optimization .......................................... 12–2
      How the query optimizer works ..................................... 12–2
      The statement parser ................................................. 12–4
      Optimizer phases ..................................................... 12–5
   Inspecting what the optimizer produces .............................. 12–11
      The _Sql_Qplan virtual system table .............................. 12–11
   Affecting what the optimizer produces ................................ 12–12
      Working with the UPDATE STATISTICS command ................. 12–12
      SQL use of index statistics ........................................ 12–12
      Updating index statistics .......................................... 12–13

Index ................................................................. Index–1
## Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Client/server architecture</td>
<td>1-3</td>
</tr>
<tr>
<td>2-1</td>
<td>OpenEdge JDBC Type 4 driver architecture</td>
<td>2-2</td>
</tr>
<tr>
<td>3-1</td>
<td>Components of an ODBC environment</td>
<td>3-3</td>
</tr>
<tr>
<td>7-1</td>
<td>SQL database table</td>
<td>7-8</td>
</tr>
<tr>
<td>10-1</td>
<td>J2EE architecture</td>
<td>10-2</td>
</tr>
<tr>
<td>10-2</td>
<td>JTA architecture</td>
<td>10-3</td>
</tr>
<tr>
<td>11-1</td>
<td>Creating Java stored procedures</td>
<td>11-4</td>
</tr>
<tr>
<td>11-2</td>
<td>Executing stored procedures</td>
<td>11-5</td>
</tr>
<tr>
<td>12-1</td>
<td>Query relational tree model</td>
<td>12-3</td>
</tr>
</tbody>
</table>
Tables

Table 2–1: JDBC driver component file locations ............................................. 2–3
Table 2–2: Data source settings ......................................................................... 2–11
Table 2–3: Encryption methods for the JDBC Driver .......................................... 2–13
Table 3–1: ODBC driver environment variables .................................................. 3–9
Table 3–2: Definitions of ODBC.ini tags ............................................................ 3–11
Table 7–1: ABL and OpenEdge SQL data types .................................................. 7–10
Table 7–2: DBTool option menu ......................................................................... 7–12
Table 8–1: Transaction isolation levels ............................................................... 8–6
Table 8–2: Insert, update, or delete record operations ......................................... 8–8
Table 8–3: Fetch or select record operations ...................................................... 8–8
Table 9–1: OpenEdge SQL Database Naming Conventions ................................. 9–3
Table 9–2: SQL configuration properties and their values .................................... 9–9
Table 9–3: Configuration properties and their values ........................................... 9–10
Table 9–4: Database properties and their values ................................................ 9–11
Table 10–1: JTA transaction states ..................................................................... 10–10
Table 11–1: Summary of OpenEdge SQL Java classes ........................................ 11–6
Table 11–2: Mapping between SQL and Java data types ..................................... 11–15
Table 11–3: getDiagnostics error-handling options ........................................... 11–24
Table 12–1: _SQL_Qplan Virtual System Table .................................................. 12–11
Preface

This Preface contains the following sections:

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

Chapter 1, “Introduction”

Provides an overview of OpenEdge SQL and the OpenEdge SQL client/server architecture.

Chapter 2, “JDBC Client”

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

Chapter 3, “ODBC Client”

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

Chapter 4, “Data Control Language and Security”

Reviews information on working with database security, creating users, and granting, modifying, and revoking privileges.

Chapter 5, “OpenEdge SQL Data Definition Language”

Furnishes information on OpenEdge SQL database structure and methods for creating, altering, and dropping database objects.

Chapter 6, “OpenEdge SQL Data Manipulation Language”

Provides information on the Data Manipulation Language statements, indexes, and join operations.

Chapter 7, “OpenEdge SQL and Advanced Business Language Interoperability”

Addresses the interoperability of ABL and OpenEdge SQL.

Chapter 8, “Data Control Language and Transaction Behavior”

Summarizes information on transactions, isolation levels, and locking.

Chapter 9, “Performing Multi-database Queries”

Provides information on the performance of multi-database queries.

Chapter 10, “Working with JTA Transactions”

Provides information on performing JTA transactions.

Chapter 11, “Stored Procedures and Triggers”

Provides information on using stored procedures and triggers.

Chapter 12, “Optimizing Query Performance”

Presents information on the operation of the SQL Query Optimizer and offers recommendations on how to get the best out of its performance.
## Typographical conventions

This manual uses the following typographical 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>
<tr>
<td>Syntax:</td>
<td></td>
</tr>
<tr>
<td>Fixed width</td>
<td>A fixed-width font is used in syntax statements, code examples, system output, and filenames.</td>
</tr>
<tr>
<td>Fixed-width italics</td>
<td>Fixed-width italics indicate variables in syntax statements.</td>
</tr>
<tr>
<td>Fixed-width bold</td>
<td>Fixed-width bold indicates variables with special emphasis.</td>
</tr>
<tr>
<td>UPPERCASE fixed width</td>
<td>Uppercase words are ABL keywords. Although these are always shown in uppercase, you can type them in either uppercase or lowercase in a procedure.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Large brackets indicate the items within them are optional.</td>
</tr>
<tr>
<td>{ }</td>
<td>Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Ellipses indicate repetition: you can choose one or more of the preceding items.</td>
</tr>
</tbody>
</table>
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:

**Syntax**

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

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

**Syntax**

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

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

**Syntax**

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

In this example, you must specify table_name or view_name:

**Syntax**

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

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

**Syntax**

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

Third party acknowledgements

OpenEdge includes AdventNet - Agent Toolkit licensed from AdventNet, Inc. http://www.adventnet.com. All rights to such copyright material rest with AdventNet.

OpenEdge includes ANTLR (Another Tool for Language Recognition) software Copyright © 2003-2006, Terence Parr All rights reserved. Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes software developed by the Apache Software Foundation (http://www.apache.org/). Copyright © 1999 The Apache Software Foundation. All rights reserved (Xerces C++ Parser (XML) and Xerces2 Java Parser (XML)); Copyright © 1999-2002 The Apache Software Foundation. All rights reserved (Xerces Parser (XML); and Copyright © 2000-2003 The Apache Software Foundation. All rights reserved (Ant). The names “Apache,” “Xerces,” “ANT,” and “Apache Software Foundation” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission of the Apache Software Foundation. For written permission, please contact apache@apache.org. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Concurrent Java software Copyright 1994-2000 Sun Microsystems, Inc. All Rights Reserved. -Neither the name of or trademarks of Sun may be used to endorse or promote products including or derived from the Java Software technology without specific prior written permission; and Redistributions of source or binary code must contain the above copyright notice, this notice and the following disclaimers: This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN Microsystems, INC. AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY

Preface–6
OPENEDGE INCLUDES DATADIRECT SOFTWARE COPYRIGHT © 1991-2007 PROGRESS SOFTWARE CORPORATION AND/OR ITS SUBSIDIARIES OR AFFILIATES. ALL RIGHTS RESERVED. (DATADIRECT CONNECT FOR JDBC TYPE 4 DRIVER); COPYRIGHT © 1993-2009 PROGRESS SOFTWARE CORPORATION AND/OR ITS SUBSIDIARIES OR AFFILIATES. ALL RIGHTS RESERVED. (DATADIRECT CONNECT FOR JDBC); COPYRIGHT © 1988-2007 PROGRESS SOFTWARE CORPORATION AND/OR ITS SUBSIDIARIES OR AFFILIATES. ALL RIGHTS RESERVED. (DATADIRECT CONNECT FOR ODBC); AND COPYRIGHT © 1988-2007 PROGRESS SOFTWARE CORPORATION AND/OR ITS SUBSIDIARIES OR AFFILIATES. ALL RIGHTS RESERVED. (DATADIRECT CONNECT64 FOR ODBC).

OPENEDGE INCLUDES DATADIRECT CONNECT FOR ODBC AND DATADIRECT CONNECT64 FOR ODBC SOFTWARE, WHICH INCLUDE ICU SOFTWARE 1.8 AND LATER - COPYRIGHT © 1995-2003 INTERNATIONAL BUSINESS MACHINES CORPORATION AND OTHERS. ALL RIGHTS RESERVED. PERMISSION IS HEREBY GRANTED, FREE OF CHARGE, TO ANY PERSON OBTAINING A COPY OF THIS SOFTWARE AND ASSOCIATED DOCUMENTATION FILES (THE "SOFTWARE"), TO DEAL IN THE SOFTWARE WITHOUT RESTRICTION, INCLUDING WITHOUT LIMITATION THE RIGHTS TO USE, COPY, MODIFY, MERGE, PUBLISH, DISTRIBUTE, AND/OR SELL COPIES OF THE SOFTWARE, AND TO PERMIT PERSONS TO WHOM THE SOFTWARE IS FURNISHED TO DO SO, PROVIDED THAT THE ABOVE COPYRIGHT NOTICE(S) AND THIS PERMISSION NOTICE APPEAR IN ALL COPIES OF THE SOFTWARE AND THAT BOTH THE ABOVE COPYRIGHT NOTICE(S) AND THIS PERMISSION NOTICE APPEAR IN SUPPORTING DOCUMENTATION.

OPENEDGE INCLUDES DATADIRECT CONNECT FOR ODBC AND DATADIRECT CONNECT64 FOR ODBC SOFTWARE, WHICH INCLUDE SOFTWARE DEVELOPED BY THE OPENSSL PROJECT FOR USE IN THE OPENSSL TOOLKIT (HTTP://WWW.OPENSSL.ORG/). COPYRIGHT © 1998-2006 THE OPENSSL PROJECT. ALL RIGHTS RESERVED. AND COPYRIGHT © 1995-1998 ERIC YOUNG (EAY@CRYPTSOFT.COM). ALL RIGHTS RESERVED.

OPENEDGE INCLUDES DATADIRECT PRODUCTS FOR THE MICROSOFT SQL SERVER DATABASE WHICH CONTAIN A LICENSED IMPLEMENTATION OF THE MICROSOFT TDS PROTOCOL.

OPENEDGE INCLUDES SOFTWARE AUTHORED BY DAVID M. GAY. COPYRIGHT © 1991, 2000, 2001 BY Lucent Technologies (dtoa.c); COPYRIGHT © 1991, 1996 BY Lucent Technologies (g_fmt.c); AND COPYRIGHT © 1991 BY Lucent Technologies (rnd_prod.s). PERMISSION TO USE, COPY, MODIFY, AND DISTRIBUTE THIS SOFTWARE FOR ANY PURPOSE WITHOUT FEE IS HEREBY GRANTED, PROVIDED THAT THIS ENTIRE NOTICE IS INCLUDED IN ALL COPIES OF ANY SOFTWARE WHICH IS OR INCLUDES A COPY OR MODIFICATION OF THIS SOFTWARE AND IN ALL COPIES OF THE SUPPORTING DOCUMENTATION FOR SUCH SOFTWARE. THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LuCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OPENEDGE INCLUDES SOFTWARE AUTHORED BY DAVID M. GAY. COPYRIGHT © 1998-2001 BY Lucent Technologies All Rights Reserved (decrsttodr.c; strtdlg.c); COPYRIGHT © 1998, 2000 BY Lucent Technologies All Rights Reserved (decrstofr.c; strtdlg.c); COPYRIGHT © 1998 BY Lucent Technologies All Rights Reserved (dmisc.c; gdtoa.h; gethex.c; gmisc.c; sum.c); COPYRIGHT © 1998, 1999 BY Lucent Technologies All Rights Reserved (gdtoa.c; misc.c; smisc.c; ulp.c); COPYRIGHT © 1998-2000 BY Lucent Technologies All Rights Reserved (gdtoaimp.h); COPYRIGHT
© 2000 by Lucent Technologies All Rights Reserved (hd_init.c). Full copies of these licenses can be found in the installation directory, in the c:/OpenEdge/licenses folder. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

OpenEdge includes http package software developed by the World Wide Web Consortium. Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All rights reserved. This work is distributed under the W3C® Software License [http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

OpenEdge includes ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.


OpenEdge includes Infragistics NetAdvantage for .NET v2009 Vol 2 Copyright © 1996-2009 Infragistics, Inc. All rights reserved.

OpenEdge includes JSTL software Copyright 1994-2006 Sun Microsystems, Inc. All Rights Reserved. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes OpenSSL software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2007 The OpenSSL Project. All rights reserved. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved. The names “OpenSSL Toolkit” and “OpenSSL Project” must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. Software distributed on an "AS IS" basis, WITHOUT
WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Quartz Enterprise Job Scheduler software Copyright © 2001-2003 James House. All rights reserved. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. This product uses and includes within its distribution, software developed by the Apache Software Foundation (http://www.apache.org/).

OpenEdge includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

OpenEdge includes the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright ©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.

OpenEdge includes Sonic software, which includes software developed by Apache Software Foundation (http://www.apache.org/). Copyright © 1999-2000 The Apache Software Foundation. All rights reserved. The names “Ant”, “Axis”, “Xalan,” “FOP,” “The Jakarta Project”, “Tomcat”, “Xerces” and/or “Apache Software Foundation” must not be used to endorse or promote products derived from the Product without prior written permission. Any product derived from the Product may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission. For written permission, please contact apache@apache.org.

OpenEdge includes Sonic software, which includes software Copyright © 1999 CERN - European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.

OpenEdge includes Sonic software, which includes software developed by ExoLab Project (http://www.exolab.org/). Copyright © 2000 Intalio Inc. All rights reserved. The names “Castor” and/or “ExoLab” must not be used to endorse or promote products derived from the Products without prior written permission. For written permission, please contact info@exolab.org. Exolab, Castor and Intalio are trademarks of Intalio Inc.

OpenEdge includes Sonic software, which includes software developed by IBM. Copyright © 1995-2003 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.
OpenEdge includes Sonic software, which includes the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

OpenEdge includes Sonic software, which includes software developed by the ModelObjects Group (http://www.modelobjects.com). Copyright © 2000-2001 ModelObjects Group. All rights reserved. The name “ModelObjects” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “ModelObjects”, nor may “ModelObjects” appear in their name, without prior written permission. For written permission, please contact djacobs@modelobjects.com.

OpenEdge includes Sonic software, which includes code licensed from Mort Bay Consulting Pty. Ltd. The Jetty Package is Copyright © 1998 Mort Bay Consulting Pty. Ltd. (Australia) and others.

OpenEdge includes Sonic software, which includes files that are subject to the Netscape Public License Version 1.1 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright 1998-1999 Netscape Communications Corporation. All Rights Reserved.

OpenEdge includes Sonic software, which includes software developed by the University Corporation for Advanced Internet Development (http://www.ucaid.edu) Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

OpenEdge includes the UnixWare platform of Perl Runtime authored by Kiem-Phong Vo and David Korn. Copyright © 1991, 1996 by AT&T Labs. Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T LABS MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes Vermont Views Terminal Handling Package software developed by Vermont Creative Software. Copyright © 1988-1991 by Vermont Creative Software.

OpenEdge includes XML Tools, which includes versions 8.9 of the Saxon XSLT and XQuery Processor from Saxonica Limited (http://www.saxonica.com/) which are available from SourceForge (http://sourceforge.net/projects/saxon/). The Original Code of Saxon
comprises all those components which are not explicitly attributed to other parties. The Initial Developer of the Original Code is Michael Kay. Until February 2001 Michael Kay was an employee of International Computers Limited (now part of Fujitsu Limited), and original code developed during that time was released under this license by permission from International Computers Limited. From February 2001 until February 2004 Michael Kay was an employee of Software AG, and code developed during that time was released under this license by permission from Software AG, acting as a "Contributor". Subsequent code has been developed by Saxonica Limited, of which Michael Kay is a Director, again acting as a "Contributor". A small number of modules, or enhancements to modules, have been developed by other individuals (either written especially for Saxon, or incorporated into Saxon having initially been released as part of another open source product). Such contributions are acknowledged individually in comments attached to the relevant code modules. All Rights Reserved. The contents of the Saxon files are subject to the Mozilla Public License Version 1.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the license can also be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

OpenEdge includes XML Tools, which includes Xs3P v1.1.3. The contents of this file are subject to the DSTC Public License (DPL) Version 1.1 (the "License"); you may not use this file except in compliance with the License. A copy of the license can be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is xs3p. The Initial Developer of the Original Code is DSTC. Portions created by DSTC are Copyright © 2001, 2002 DSTC Pty Ltd. All rights reserved.

OpenEdge includes YAJL software Copyright 2007, Lloyd Hilaiel. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. Neither the name of Lloyd Hilaiel nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Introduction

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:

- 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, and provides for query optimization. These features are designed to maximize database scalability and performance.

- **JDBC Driver for OpenEdge by DataDirect** — The JDBC Driver for OpenEdge is provided by DataDirect Technologies. The JDBC driver enables Java-based applications to access the OpenEdge RDBMS. The JDBC driver is installed as part of the OpenEdge SQL Client Access product.

- **ODBC Driver for OpenEdge by DataDirect** — The ODBC Driver for OpenEdge is provided by DataDirect Technologies. The ODBC driver provides access to the OpenEdge database from applications that support the ODBC interface. The ODBC driver is installed as part of the OpenEdge SQL Client Access product.
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.

Figure 1-1 illustrates the architecture of the ABL and OpenEdge SQL clients when connected to the database in a client/server configuration.

Figure 1–1: 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 Figure 1–1, 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:

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

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.
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. *Figure 2–1* shows the architecture of a Java application using a Type 4 JDBC driver.

![OpenEdge JDBC Type 4 driver architecture](image)

*Figure 2–1: OpenEdge JDBC Type 4 driver architecture*
In Figure 2–1, 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 `opendedge.jar` and `pool.jar` files. Table 2–1 lists the locations of the files.

**Table 2–1: JDBC driver component file locations**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Files and locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>%DLC%/java/opendedge.jar</td>
</tr>
<tr>
<td>Sun Solaris SPARC (32 bit and 64 bit)</td>
<td>$DLC/java/opendedge.jar</td>
</tr>
<tr>
<td>Compaq Tru64 UNIX</td>
<td></td>
</tr>
<tr>
<td>Linux X86</td>
<td></td>
</tr>
<tr>
<td>IBM AIX</td>
<td></td>
</tr>
<tr>
<td>HP-UX (32 bit and 64 bit)</td>
<td></td>
</tr>
</tbody>
</table>

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:

```
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:

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:

   ![New System Variable dialog box]

4. In the Variable name field, type CLASSPATH. In the Variable value field, type:
   
   %CLASSPATH%;%DLC%\java\openedge.jar; %DLC%\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. This section covers:

- Connecting using SQL Explorer
- Connecting from a Java application using a URL
- Connecting from a Java application using a data source
- Enabling encryption
- JDBC connection parameters
- Troubleshooting database connection problems

For information on connecting to multiple databases in order to perform multiple-database queries, see Chapter 9, “Performing Multi-database Queries.”

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:

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

```bash
sqlexp -url jdbc:datadirect:openedge://host:port;databaseName=db_name;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.
- 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” section on page 2–9 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:

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:

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:

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

This is an example of a connection string:

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

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

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 “JDBC connection parameters” section on page 2–13.

- **CacheSize**
  
  Indicates how many entries will be in the statement cache. For more information about the `statementCacheSize` connection parameter, see the “JDBC connection parameters” section on page 2–13.
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. Table 2–2 lists the settings.

Table 2–2: Data source 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.

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.
**ValidateServerCertificate**—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.
Table 2–3 provides descriptions of the methods used to enable encryption in the driver.

**Table 2–3: 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
      
      jdbc:datadirect:openedge://host_name:port;databaseName=database_name
    
    Or
    
    jdbc:datadirect:openedge://host_name:port;databaseName=database_name;
    servicename=service_name
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:

- 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 specific information about the DataDirect ODBC driver, see http://www.datadirect.com/techres/progressdoc/index.ssp.

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.
Figure 3–1 shows the 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.

This section includes the following topics:

- Internet Protocol support
- Configuring Windows clients
- Enabling encryption
- Configuring UNIX clients
- Definitions of ODBC.INI tags
- Testing your ODBC connection on UNIX

For information on connecting to multiple databases in order to perform multiple-database queries, see Chapter 9, “Performing Multi-database Queries.”

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.
   - **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 Chapter 8, “Data Control Language and Transaction Behavior,” 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: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.

8. 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*.

To enable encryption:

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

- **Encryption Method** — This option 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. This section provides the following information:

- **Setting environment variables**

- **Configuring data sources on a UNIX environment**

#### Setting environment variables

*Table 3–1* lists the environment variables you must set to configure the ODBC driver and the location where you must set them.

#### Table 3–1: 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>
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:

```ini
[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

[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
```

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`.

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:

```ini
[ODBC Data Sources]
sports=Progress_SQL92_Driver
[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

Table 3–2 provides definitions of the ODBC.ini tags.

#### Table 3–2: Definitions of ODBC.ini tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Driver</strong></td>
<td>The data-source-specific library used by the Driver Manager.</td>
</tr>
<tr>
<td><strong>Database name (mandatory)</strong></td>
<td>The name of the database used by the DSN.</td>
</tr>
<tr>
<td><strong>PortNumber (mandatory)</strong></td>
<td>The Network Service number on which the database server is listening for connections.</td>
</tr>
<tr>
<td><strong>HostName (mandatory)</strong></td>
<td>The name of the host machine on which the database is running.</td>
</tr>
<tr>
<td><strong>LogonID</strong></td>
<td>The default user ID for logon to the database server.</td>
</tr>
<tr>
<td><strong>Password</strong></td>
<td>The password for the default user ID.</td>
</tr>
<tr>
<td><strong>APILevel</strong></td>
<td>A number indicating the ODBC interface conformance level supported by the 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><strong>ConnectFunctions</strong></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 “Y”; otherwise, it is “N”.</td>
</tr>
<tr>
<td></td>
<td>• If the driver supports SQLDriverConnect, the second character is “Y”; otherwise, it is “N”.</td>
</tr>
<tr>
<td></td>
<td>• If the driver supports SQLBrowseConnect, the third character is “Y”; otherwise, it is “N”.</td>
</tr>
<tr>
<td></td>
<td>• For example, if a driver supports SQLConnect and SQLDriverConnect but not SQLBrowseConnect, the three-character string is “YYN”.</td>
</tr>
<tr>
<td><strong>CPTimeout</strong></td>
<td>The time interval setting for Connection Pooling Timeout. Not supported for UNIX.</td>
</tr>
<tr>
<td><strong>DriverODBCVer</strong></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>
</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.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileUsage</td>
<td>A number indicating how a file-based driver directly treats files in a data source:</td>
</tr>
<tr>
<td>0 — The driver is not a file-based driver. For example, a driver may be a DBMS-based driver.</td>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<td>SQLLevel</td>
<td>A number indicating the SQL-92 grammar supported by the driver:</td>
</tr>
<tr>
<td>0 — SQL-92 Entry</td>
<td></td>
</tr>
<tr>
<td>1 — FIPS127-2 Transitional</td>
<td></td>
</tr>
<tr>
<td>2 — SQL-92 Intermediate</td>
<td></td>
</tr>
<tr>
<td>3 — SQL-92 Full</td>
<td></td>
</tr>
<tr>
<td>UsageCount</td>
<td>Indicates driver libraries are in use. This value should not be modified.</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>(mandatory)</td>
<td></td>
</tr>
<tr>
<td>StaticCursorLongColBuffLen</td>
<td>The default size for retrieving sections of LVARBINARY data.</td>
</tr>
</tbody>
</table>
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 testconnect.c sample program to test your ODBC connection.

```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");
        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");
        return sqlReturn;
    }
    /* set the ODBC application version to 3.x */
    sqlReturn = SQLSetEnvAttr(environmentHandle,
                               SQL_ATTR_ODBC_VERSION,
                               (SQLPOINTER) SQL_OV_ODBC3,
                               SQL_IS_UIINTEGER);
    if (sqlReturn != SQL_SUCCESS)
    {
        printf("Unable to set ODBC Versoin to 3.x, exiting.\n");
        SQLFreeHandle(SQL_HANDLE_ENV, environmentHandle);
        return sqlReturn;
    }
    /* make sure we got at least 3 arguments to the exe */
    if (argc < 4)
    {
        printf("Insufficient parameters provided.\n");
        printf("Usage - %s dsn userid password\n", argv[0]);
        return 1;
    }
    /* 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");
        return sqlReturn;
    }
    /* set the ODBC application version to 3.x */
    sqlReturn = SQLSetEnvAttr(environmentHandle,
                               SQL_ATTR_ODBC_VERSION,
                               (SQLPOINTER) SQL_OV_ODBC3,
                               SQL_IS_UIINTEGER);
    if (sqlReturn != SQL_SUCCESS)
    {
        printf("Unable to set ODBC Versoin to 3.x, exiting.\n");
        SQLFreeHandle(SQL_HANDLE_ENV, environmentHandle);
        return sqlReturn;
    }
    /* make sure we got at least 3 arguments to the exe */
    if (argc < 4)
    {
        printf("Insufficient parameters provided.\n");
        printf("Usage - %s dsn userid password\n", argv[0]);
        return 1;
    }
    /* 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");
        return sqlReturn;
    }
    /* set the ODBC application version to 3.x */
    sqlReturn = SQLSetEnvAttr(environmentHandle,
                               SQL_ATTR_ODBC_VERSION,
                               (SQLPOINTER) SQL_OV_ODBC3,
                               SQL_IS_UIINTEGER);
    if (sqlReturn != SQL_SUCCESS)
    {
        printf("Unable to set ODBC Versoin to 3.x, exiting.\n");
        SQLFreeHandle(SQL_HANDLE_ENV, environmentHandle);
        return sqlReturn;
    }
    /* make sure we got at least 3 arguments to the exe */
    if (argc < 4)
    {
        printf("Insufficient parameters provided.\n");
        printf("Usage - %s dsn userid password\n", argv[0]);
        return 1;
    }
    /* 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");
        return sqlReturn;
    }
    /* set the ODBC application version to 3.x */
    sqlReturn = SQLSetEnvAttr(environmentHandle,
                               SQL_ATTR_ODBC_VERSION,
                               (SQLPOINTER) SQL_OV_ODBC3,
                               SQL_IS_UIINTEGER);
    if (sqlReturn != SQL_SUCCESS)
    {
        printf("Unable to set ODBC Versoin to 3.x, exiting.\n");
        SQLFreeHandle(SQL_HANDLE_ENV, environmentHandle);
        return sqlReturn;
    }
    /* make sure we got at least 3 arguments to the exe */
    if (argc < 4)
    {
        printf("Insufficient parameters provided.\n");
        printf("Usage - %s dsn userid password\n", argv[0]);
        return 1;
    }
    /* 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");
        return sqlReturn;
    }
    /* set the ODBC application version to 3.x */
    sqlReturn = SQLSetEnvAttr(environmentHandle,
                               SQL_ATTR_ODBC_VERSION,
                               (SQLPOINTER) SQL_OV_ODBC3,
                               SQL_IS_UIINTEGER);
    if (sqlReturn != SQL_SUCCESS)
    {
        printf("Unable to set ODBC Versoin to 3.x, exiting.\n");
        SQLFreeHandle(SQL_HANDLE_ENV, environmentHandle);
        return sqlReturn;
    }
    /* make sure we got at least 3 arguments to the exe */
    if (argc < 4)
    {
        printf("Insufficient parameters provided.\n");
        printf("Usage - %s dsn userid password\n", argv[0]);
        return 1;
    }
    /* 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");
        return sqlReturn;
    }
    /* set the ODBC application version to 3.x */
```
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 Chapter 8, “Data Control Language and Transaction Behavior.”

This chapter contains the following sections:

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

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

This section contains the following information:

- Comparing OpenEdge SQL and ABL security
- Comparing authentication and authorization

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, the ABL database administrator can use the Data Administration tool to restrict access by creating users and assigning passwords. Once the first ABL user has been created in the User table, no other users may access the database until they, too, have been assigned a User ID and password.

The database administrator can further enhance security by assigning user permissions to define actions which may be taken on specific database objects. ABL permissions are assigned through the Data Administration tool. For information on using the Data Administration tool, see *OpenEdge Data Management: Database Administration*.

Any security permissions defined for users through the Data Administration tool apply only to users accessing the database through an ABL client. If users are able to access a database through both SQL and ABL clients, then DBAs should ensure that identical 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 via a client. The server will then verify your user identification. The client may be either an ABL client or an OpenEdge SQL (JDBC or ODBC) client.

From OpenEdge SQL, authentication is enabled 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. ABL DBAs may create entries in the same User table by assigning user IDs and passwords through the Data Administration tool.

Although OpenEdge SQL and ABL security models are independent of each other, they do share one common aspect of security—the User table. User IDs and passwords created in ABL are also recognized by OpenEdge SQL. Likewise, user IDs and passwords created using OE SQL are also recognized by ABL.

SQL DBAs create new users and assign user IDs through the CREATE statement. ABL DBAs create users and assign user IDs using the OpenEdge Data Administration tool. For information on using the Data Administration tool, see OpenEdge Data Management: Database Administration.

Note: When initially populating the User table from the SQL side, ensure the first table entry is that of the DBA and ensure that particular username is granted DBA and RESOURCE privileges in the same transaction. Otherwise, you may potentially lock yourself out of the database.

Authorization

Authorization is the process by which DBAs enable users to perform tasks on the database, such as retrieving, updating and deleting information. OpenEdge SQL and ABL each employ their own method of performing authorization.

OpenEdge SQL performs authorization through the GRANT statement. ABL DBAs authorize users to perform tasks by using the Data Administration tool.

For more on performing authorization with OpenEdge SQL, see the “Granting privileges” section on page 4–6. For more information on granting permissions through the Data Administration tool, see OpenEdge Data Management: Database Administration.
Creating users

This section covers the following information:

- Creating database administrators
- Creating users

Note: 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 Chapter 5, “OpenEdge SQL Data Definition Language.”

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 uses the CREATE statement to create a user and then uses the GRANT statement to provide the user with administrative privileges.
- In ABL, a DBA uses the Data Administration Tool to create other administrators.
Creating users

Use the following syntax to employ the CREATE USER statement:

**Syntax**

```
CREATE USER 'username', 'password';
```

In Example 4–1, an account with DBA privileges creates the 'username' 'GPS' with 'password' 'star'.

**Example 4–1: CREATE USER statement**

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

A user’s password can be changed easily by using the ALTER USER statement, as shown:

**Syntax**

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

**Example 4–2 demonstrates the use of the ALTER USER statement.**

**Example 4–2: ALTER USER statement**

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

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*.
Granting privileges

This section covers the following topics:

- Privilege basics
- GRANT statement

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

**Syntax**

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

**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 4–3 demonstrates the use of the `GRANT RESOURCE` statement.

**Example 4–3: 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.

**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.

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

**Syntax**

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

This is the syntax for the `privilege` value:

**Syntax**

```
{ 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 Example 4–4, ‘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.

Example 4–4 illustrates the granting of table-specific privileges.

**Example 4–4: GRANT UPDATE statement**

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

The GRANT UPDATE statement has limited GSP’s ability to interact with the item table. Now, if GSP attempts to update a column to which he has not been granted access, the database will return the error message in Example 4–5.

**Example 4–5: SQL error message**

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

**Granting public access**

The GRANT statement can be easily modified to make previously restricted columns accessible to the public, as in Example 4–6.

**Example 4–6: Granting update privilege to public**

```sql
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.

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

**Syntax**

```sql
REVOKE { RESOURCE, DBA, AUDIT_ADMIN, AUDIT_ARCHIVE, AUDIT_INSERT }
FROM { username [, username ] };
```

**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:

**Syntax**

```sql
REVOKE [ GRANT OPTION FOR ] { privilege [, privilege ] } [ ALL PRIVILEGES ]
ON table_name
FROM { username [, username ] } [ PUBLIC ];
```

where privilege is:

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

The `REVOKE` statement can be used to remit the privileges previously granted to 'GPS', as shown in Example 4–7.

**Example 4–7: REVOKE statement**

```sql
REVOKE UPDATE
ON Item (ItemNum, ItemName, CatDescription)
FROM "GPS";
```
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*. 
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:

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

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

This section covers the following information:

- Working with tables
- Working with indexes
- Working with views
- Working with sequences

**Note:** When creating a table, view, or index in the SQL engine, choose table, view, index, 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
Working with tables

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

Using the CREATE TABLE statement

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 statement uses the following syntax:

**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
;
```

The following syntax is used to define an LOB column in CREATE TABLE statement:

**Syntax**

```
{ LVARCHAR | CLOB | LVARBINARY | BLOB } [ ( length ) ]
[ AREA areaname ]
[ 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*.
Example 5–1 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.

Example 5–1: CREATE TABLE statement

```sql
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:

**Syntax**

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

The following CREATE TABLE statement shows how to use the DEFAULT clause. Example 5–2 sets a default value of 10 for the deptno column.

Example 5–2: CREATE TABLE statement with DEFAULT clause

```sql
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*.


**ALTER TABLE**

The ALTER TABLE statement lets you add new columns to a table, delete columns from a table, or change the format and labels associated with an existing column.

---

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

---

The ALTER TABLE statement has the following syntax:

```
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 }
};
```

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*.

Example 5–3 shows how the ALTER TABLE statement is used to add a column to a table.

**Example 5–3: ALTER TABLE statement**

```sql
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:

**Syntax**

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

Example 5–4 shows how the ALTER TABLE statement is used to rename an existing table.

**Example 5–4: Using ALTER TABLE statement to rename table**

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

**DROP TABLE**

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

**Syntax**

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

Example 5–5 illustrates the use of a DROP TABLE statement.

**Example 5–5: DROP TABLE statement**

```sql
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.

Using the CREATE INDEX statement

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`).

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

The `CREATE INDEX` statement uses the following syntax:

**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'} ];
```

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 **Example 5–6**.

**Example 5–6: CREATE INDEX statement**

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

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

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. **Example 5–7** demonstrates the use of a `DROP INDEX` statement.

**Example 5–7: DROP INDEX statement**

```
DROP INDEX idx_cust ON customer;
```
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 Chapter 6, “OpenEdge SQL Data Manipulation Language.”

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`.

Example 5–8 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.

**Example 5–8: CREATE VIEW statement**

```
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 Example 5–9.

**Example 5–9: DROP VIEW statement**

```
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.

The CREATE SEQUENCE statement uses the following syntax:

**Syntax**

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

In Example 5–10, a sequence is used to generate unique customer numbers when a new row is inserted into the table `pub.customer`.

**Example 5–10: CREATE SEQUENCE statement**

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

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 5–11 uses NEXTVAL to assign a new customer number into a customer table.

**Example 5–11: INSERT statement using NEXTVAL**

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

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

**SELECT statement using CURRVAL**

```
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**

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

In Example 5–12, the DROP SEQUENCE statement removes the sequence named customer from the pub schema.

**Example 5–12: DROP SEQUENCE statement**

```
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.

The syntax of the `ALTER SEQUENCE` statement is:

**Syntax**

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

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 5–13 modifies a sequence that is used to generate unique customer numbers when a new customer is inserted into the table `pub.customer`.

**Example 5–13: ALTER SEQUENCE statement**

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

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:

**Syntax**

```
proutil database_name -C enablesq64
```

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.
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.

This section covers the following information:

- Need for integrity constraints
- Types of integrity constraints
- Referential constraints
- Handling cycles in referential integrity

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 Example 5–14.

Example 5–14: UNIQUE constraint

```sql
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.

Example 5–15 shows the assignment of table constraint `prim_constr` on table `supplier_item`. You specify a constraint name with the `CONSTRAINT` keyword.

**Example 5–15: Constraint modifier**

```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.

Example 5–16 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.

**Example 5–16: Checking a constraint**

```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.

In Example 5–17, 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.

**Example 5–17: Column-level check constraint**

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

In Example 5–18 and Example 5–19, the INSERT statement results in an error, and the corresponding row is not inserted into the table.

**Example 5–18: INSERT failure due to check constraint**

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

**Example 5–19: Constraint violation error message**

```sql
=== 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.
In **Example 5–20**, 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.

**Example 5–20: Table-level check constraint**

```sql
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 5–21** shows an INSERT statement for the supplier table created in the previous example. This INSERT operation results in a check constraint violation.

**Example 5–21: Table-level check constraint violation**

```sql
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 **Example 5–22**, 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.

**Example 5–22: Column-level primary key**

```sql
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 Example 5–23, 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.

Example 5–23: Candidate key

```
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 NOT NULL 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 Example 5–24, 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.

Example 5–24: Referential constraint

```
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 **Example 5–25**, `item_no` is the foreign key referencing the item table, and the foreign key is specified at the column level.

**Example 5–25: Foreign key constraint**

```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.

**Example 5–26** 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.

**Example 5–26: Referential and foreign key constraints**

```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.

In Example 5–27, 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.

Example 5–27: Table reference cycle

```
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 );
```

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. Example 5–28 shows this single-table cycle.

Example 5–28: Single-table reference cycle

```
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.
Maintaining data integrity

Inserting rows in a cycle

To 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.

Example 5–29 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.

Example 5–29: Updating data in a 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)  
) ;
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

This section covers the following topics:

- Using the SQLDUMP utility
- Using the SQLLOAD utility
- Using the SQLSCHEMA utility

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.

The SQLDUMP utility has the following syntax:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX Windows</td>
<td><code>sqldump -u user_name [-a password] [-C code-page-name] -t [owner_name.]table_name1 [owner_name.]table_name2 ... database_name</code></td>
</tr>
</tbody>
</table>

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 5–30 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.

**Example 5–30: SQLDUMP from selected tables**

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

Example 5–31 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.

**Example 5–31: SQLDUMP based on table names**

```
sqldump -u tucker -a sulky -t%.cust%,%.invent%,%.sales%
progress:T:thunder:4077:salesdb
```
Example 5–32 directs the SQLDUMP utility to write the data from all tables for all owner names in the salesdb database.

**Example 5–32: SQLDUMP of entire database**

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

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

Example 5–33 depicts a dump file.

**Example 5–33: 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:

**Syntax**

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

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

By default, SQLDUMP displays promsigs 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 promsigs messages using the Russian code page (unless you specify a different code page by setting the client’s SQL_CLIENT_CHARSET_PROMSIGS environment variable).

### 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.

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

Use the following syntax for the SQLLOAD utility:

**Syntax**

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| UNIX Windows     | sqlload -u user_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 maxrows ]
|                  | [ -F comma | quote ]
|                  | 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 5–34** 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.

**Example 5–34: SQLLOAD of two dump files**

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

**Example 5–35** directs SQLLOAD to load the data from all appropriately named dump files into the specified tables in the salesdb database.

**Example 5–35: SQLLOAD from appropriately named files**

```plaintext
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.

To load the table **Yearly Profits**, use the UNIX command-line syntax, as shown in Example 5–36.

Example 5–36: SQLLOAD of files with delimited identifiers

```bash
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.

### 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.

Use the following syntax for the SQLSCHEMA utility:

**Syntax**

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX Windows</td>
<td>sqlschema -u user_name [-a password] [-t owner_name.]table_name1,owner_name.]table_name2, ... [-t owner_name.]view_name1,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] database_name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td></td>
</tr>
</tbody>
</table>
The SQLSCHEMA utility cannot write definitions for ABL tables. 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 5–37 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.

Example 5–37: SQLSCHEMA utility for writing object definitions

```
sqlschema -t tucker.customers,tucker.products -T
          tucker.customers,tucker.products progress:T:thunder:4077:salesdb
```

Example 5–38 directs the SQLSCHEMA utility to write table definitions to an output file named salesdbschema.dfsql.

Example 5–38: SQLSCHEMA for writing object definitions to output file

```
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.
OpenEdge SQL Data Manipulation Language

The Data Manipulation Language (DML) is used to select, insert, delete, or update information of a database, and is described in the following sections:

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

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

- Using the SELECT statement
- Using the INSERT statement
- Using the UPDATE statement
- Using the DELETE statement

Using the SELECT statement

Use the SELECT statement to retrieve information from a database.

The SELECT statement uses the following syntax:

**Syntax**

```sql
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 Example 6–1, the simple SELECT statement retrieves all columns from the Customer table.

**Example 6–1: SELECT statement on entire table**

```sql
SELECT * FROM Customer;
```
In Example 6–2, the statement is easily modified to identify columns from which the data will be retrieved.

**Example 6–2: SELECT statement with columns identified**

```
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. Example 6–3 retrieves the customer and order information from both the Customer and Order tables.

**Example 6–3: SELECT statement from multiple tables**

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

The WHERE clause can be used to further refine your query. In Example 6–4, 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.

**Example 6–4: SELECT statement using WHERE clause**

```
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.

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*.

### Using the INSERT statement

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

The INSERT statement uses the following syntax:

**Syntax**

```
INSERT INTO [owner_name.]table_name|view_name
[ ( column_name [, column_name ] , ... ) ]
{ VALUES ( value [, value ] , ... ) | query_expression } ;
```
Example 6–5 depicts a single row being added to a table.

**Example 6–5: INSERT statement**

```
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 Example 6–6, rows are created in a table of New York customers based on a query of the Customer table.

**Example 6–6: INSERT statement based on query expression**

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

**Notes:** 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 `VALUES` to specify columns will insert one row into the table. Use the query expression to insert multiple rows.

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

**Using the UPDATE statement**

The UPDATE statement updates the rows and columns of the specified table with the given values for rows that satisfy the `search_condition`. The UPDATE statement uses the following syntax:

**Syntax**

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

**assignment:**

**Syntax**

```
column = { expr | NULL }
| ( column [, column ] , ... ) = ( expr [, expr ] )
| ( column [, column ] , ... ) = ( query_expression )
```
In Example 6–7, a simple UPDATE statement is used to revise the credit limit of all customers in the Customer table.

**Example 6–7: UPDATE statement**

```
UPDATE Customer
SET CreditLimit = 50000;
```

Use the WHERE clause to identify a specific column and row to be updated, as shown in Example 6–8.

**Example 6–8: UPDATE statement with WHERE clause**

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

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

**Using the DELETE statement**

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.

The DELETE statement uses the following syntax:

**Syntax**

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

Example 6–9 depicts a simple delete on a single table.

**Example 6–9: DELETE statement**

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

**Note:** A DELETE statement will fail in the event it attempts to remove rows from primary or candidate keys that reference other tables.
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 Chapter 12, “Optimizing Query Performance.”

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 outer 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:

**Syntax**

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

Example 6–10 demonstrates a join that retrieves information from two tables relating customers and their order information.

**Example 6–10: Inner join**

```
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>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
<td>. . .</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. Example 6–11 modifies the inner join in Example 6–10 by employing table aliases.

Example 6–11: Inner join using table aliases

```sql
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:

Syntax

```sql
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:

Syntax

```sql
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 \textit{WHERE} clause:

\begin{verbatim}
SELECT Customer.Custnum, Customer.Name, Order.Ordernum, Order.Orderdate
FROM Customer, Order
WHERE Customer.CustNum = Order.CustNum (+) ;
\end{verbatim}

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

\begin{verbatim}
<table>
<thead>
<tr>
<th>CustNum</th>
<th>Name</th>
<th>Ordernum</th>
<th>OrderDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lift Tours</td>
<td>6</td>
<td>2006-02-11</td>
</tr>
<tr>
<td>1</td>
<td>Lift Tours</td>
<td>1</td>
<td>2006-03-17</td>
</tr>
<tr>
<td>1</td>
<td>Lift Tours</td>
<td>36</td>
<td>2006-05-01</td>
</tr>
<tr>
<td>1</td>
<td>Lift Tours</td>
<td>79</td>
<td>2006-06-22</td>
</tr>
<tr>
<td>. . .</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Ace Tennis</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>7</td>
<td>Xtreme Surf</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{Example 6–12} uses the \texttt{LEFT OUTER JOIN} phrase in the \texttt{FROM} clause.

\textbf{Example 6–12: Left outer join using FROM clause}

\begin{verbatim}
SELECT e.firstname, e.lastname, e.deptcode, d.deptname
FROM employee e, department d
WHERE e.deptcode = d.deptcode
ORDER BY d.deptcode, e.lastname;
\end{verbatim}

This query produces the following results:

\begin{verbatim}
<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>
<tr>
<td>. . .</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{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.
Example 6–13 offers an example of a right outer join.

**Example 6–13: Right outer join**

```sql
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.
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:

- 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

This section provides information on managing OpenEdge SQL and ABL database connections, including:

- Determining database server requirements
- Starting SQL and ABL brokers

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:
   
   \[-Mn = (\text{Sum of all } Mpb \text{ values}) + \text{one per each additional broker}\]

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 \(-Ma\).

For the broker, invoke PROSERVE with the following:

- Set \(-Ma\) to the maximum number of users
- Set \(-MPb\) to the maximum number of servers for this broker
- Set \(-ServerType\)
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
```

**Notes:** 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.”

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 `-ss` startup parameter. For more information on creating SSL brokers, see *OpenEdge Getting Started: Core Business Services* 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.

Creating, altering, or dropping a user via SQL is equivalent to creating, maintaining, or deleting a user with the OpenEdge Data Administration tool. The user list updated for the OpenEdge SQL interface is 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** — The ABL interface will not display an authentication dialog box, but the OpenEdge SQL interface will. All SQL users will be required to enter a valid username and password into the authentication dialog box before they will be permitted access to the database.

- **If users have been created** — Users accessing the database through the ABL interface are required to provide an identification and password. 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 Chapter 4, “Data Control Language and Security.” For more information on and 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. Specifically, this section covers the following topics:

- Comparing ABL and OpenEdge SQL
- Understanding OpenEdge SQL database structure
- Comparing OpenEdge SQL and ABL database objects
- Naming objects for OpenEdge SQL and ABL databases
- Working with data type compatibility
- Working with SQL column widths
- Working with triggers
- Working with locking behavior and isolation levels

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.
Figure 7–1 offers an example of an SQL database table.

![SQL database table](image)

**Figure 7–1: 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, Table 7–1 illustrates the compatible data types of the two languages.

Table 7–1: 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
- The DBTool utility

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:

\[\text{Column column in table table has value exceeding its max length or precision (7864)}\]

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:

   ![DBTool option menu](image)

2. Select an option from the menu and press **Enter**.

   Table 7–2 describes the options available in the 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>
</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 ABL-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, see Chapter 11, “Stored Procedures and Triggers.”
**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.

For ABL clients, there is a lock wait timeout 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 will time-out and give up waiting long before the ABL client. SQL can be set by using the environment variable PRO_LOCKWAIT_TIMEOUT.

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 Chapter 8, “Data Control Language and Transaction Behavior.”
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:

- 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
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:

**Syntax**

```
COMMIT [ WORK ];
```

The following example shows a COMMIT statement:

```
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:

**Syntax**

```
ROLLBACK [ WORK ];
```
The following example shows how to use the ROLLBACK statement:

```
ROLLBACK WORK ;
```

**Notes:** The ROLLBACK statement does not affect the contents of any host variables or the flow of control in the program.

The database issues a ROLLBACK operation automatically when there is an abnormal termination of the application program.

For more on the ROLLBACK statement, see *OpenEdge Data Management: SQL Reference.*
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 Example 8–1.

Example 8–1: Dirty read

<table>
<thead>
<tr>
<th>User A executes:</th>
<th>User B executes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT INTO State (state, state_name, region) VALUES ('ME', 'Maine', 'Northeast');</td>
<td>SELECT * FROM State;</td>
</tr>
<tr>
<td>User B sees:</td>
<td>state ‘ME’</td>
</tr>
<tr>
<td>User A executes:</td>
<td>ROLLBACK WORK;</td>
</tr>
<tr>
<td>User B has seen data that really did not exist.</td>
<td></td>
</tr>
</tbody>
</table>

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 Example 8–2.

Example 8–2: Nonrepeatable read

<table>
<thead>
<tr>
<th>User A executes:</th>
<th>User B executes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM State;</td>
<td>UPDATE State SET state_name = 'Arkansas' WHERE state = 'AK'; COMMIT WORK; SELECT * FROM pub.State</td>
</tr>
<tr>
<td>User A re-executes:</td>
<td>SELECT * FROM State;</td>
</tr>
<tr>
<td>User A has now updated records in the result set.</td>
<td></td>
</tr>
</tbody>
</table>
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 Example 8–3.

Example 8–3: Phantom read

<table>
<thead>
<tr>
<th>User A executes:</th>
<th>SELECT * FROM State;</th>
</tr>
</thead>
<tbody>
<tr>
<td>User B executes:</td>
<td>INSERT INTO pub.State (state, state_name, region) VALUES ('CT', 'Connecticut', 'Northeast'); COMMIT WORK;</td>
</tr>
<tr>
<td>User A re-executes:</td>
<td>SELECT * FROM pub.State;</td>
</tr>
<tr>
<td>User 1 has new records in the results set.</td>
<td></td>
</tr>
</tbody>
</table>

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;
```

*isolation_level_name*

**Syntax**

| READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE |

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.

Table 8–1 identifies which phenomena are either permitted or prevented by each isolation level.

Table 8–1: 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>

Notes: 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.

This section covers:

- Lock modes
- How lock levels and lock modes interact
- Understanding lock acquisition

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

Table 8–2 and Table 8–3 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 8–2: 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 8–3: 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 **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 **READ UNCOMMITTED**.
Enhancing performance with locking hints

A range of table-level locking hints can be specified using the SELECT, INSERT, UPDATE, and 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 READPAST locking hint

The READPAST locking hint skips locked rows. This option causes a transaction to skip rows locked by other transactions that would ordinarily appear in the result set, rather than block the transaction waiting for the other transactions to release their locks on these rows. The READPAST lock hint applies only to transactions operating at READ COMMITTED isolation level and will read only past row-level locks. This only applies to the SELECT statement.

The locking hint clause, such as READPAST, can only be specified in the main SELECT statement, but not in the subquery SELECT statement in the search condition of the WHERE clause.

The SELECT statement uses the following syntax:

Syntax

```
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:

Syntax

```
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 Example 8–4 and Example 8–5 illustrate the use of the READPAST locking hint.

**Example 8–4: READPAST locking hint with NOWAIT specified**

```sql
SELECT * FROM Customer WHERE “CustNum” < 100 ORDER BY “CustNum” FOR UPDATE
WITH (READPAST NOWAIT);
```

**Example 8–5: READPAST locking hint with WAIT time-out specified**

```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. With 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
- ALTER TABLE ALTER COLUMN
- ALTER TABLE ALTER INDEX
- CREATE INDEX
- CREATE SEQUENCE
- CREATE STORED PROCEDURE
- CREATE SYNONYM
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DROP SYNONYM
- DROP VIEW
- GRANT
- REVOKE
- SET ENCRYPT
- SET DECRYPT
- SET BUFFER_POOL

**Note:** Inactive indexes can be 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 syntax and specific information about each command, see *OpenEdge Data Management: SQL Reference.*
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. Specifically, this chapter covers:

- 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. This section offers an overview of the multi-database query, including information on:

- The process of multi-database queries
- Working with catalogs in multi-database queries

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 Table 9–1, which are in full compliance with the SQL standard.

<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.
Performing Multi-database Queries

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:

```
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. This section covers the following:

- Connecting to multiple databases using SQL commands
- Specifying a database default catalog
- Determining catalog availability
- Using the CONNECT AS CATALOG statement
- Disconnecting from catalogs
- Using the DISCONNECT CATALOG statement
- Using properties files to enable multiple database 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:

**Syntax**

```
SET CATALOG catalog_name;
```

Example 9–1 shows how the auxiliary database connection is identified by the catalog named mydb1.

**Example 9–1: SET CATALOG statement**

```
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:

**Syntax**

```
SHOW CATALOGS [ ALL | { PRO_NAME | PRO_TYPE | PRO_STATUS } |
[ , PRO_NAME | PRO_TYPE | PRO_STATUS } ];
```

Example 9–2 demonstrates the use of the SHOW CATALOGS statement.

**Example 9–2: 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:

**Syntax**

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

In Example 9–3, the database named customer in directory /usr/databases is connected as a catalog named ‘mydb1’.

**Example 9–3: CONNECT AS CATALOG statement**

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

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

Disconnected 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:

**Syntax**

```
DISCONNECT CATALOG catalog_name;
```

In Example 9–4, an auxiliary database connection is removed.

**Example 9–4: DISCONNECT CATALOG statement**

```
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. Example 9–5 provides an example of a multi-database connection properties file.

**Example 9–5: Multi-database connection properties file**

```
[sql-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, Ma1

[database.Ny]
  Name=NewYork
  Catalog=Lions
  Location=/usr1/kjain/States/NewYork

[database.Ma1]
  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. Table 9–2 describes SQL configuration properties:

**Table 9–2: 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>
Performing Multi-database Queries

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. Table 9–3 describes the configuration properties:

<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. Table 9–4 describes database properties:

### Table 9–4: 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 Example 9–5 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 are 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 Example 9–5, 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 Example 9–6.

**Example 9–6: Property file identified in connection URL**

```
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 Example 9–7.

**Example 9–7: Property file identified in ODBC driver settings**

```
Data Source Name: AlbTeams
Description: ALB Teams Databases
Host Name: albhost
Port Number: 3535
Database Name: empty][-mdbq:AlbSales]
User ID: bjones
```

Once the properties file is read and the list of database objects is constructed, the database objects in the configuration are 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 muting to read the list concurrently. Otherwise, the server will refresh the list with the properties file contents appropriately muted 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, gldb, 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. Specifically, this chapter covers:

- 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, thus 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 Figure 10–1, 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 or application server is the transaction manager.

Figure 10–1: J2EE architecture
Understanding JTA architecture

The JTA 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 Figure 10–2, 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.

Figure 10–2:  JTA architecture
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. This section covers:

- XADataSource
- XAConnection
- XAResource
- XAResource methods

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 can specify 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 can complete in one of three ways:

  – A transaction might be suspended temporarily in an incomplete state and must be resumed via the start method.

  – A transaction might be flagged to indicate that a portion of work has failed. The resource manager may mark the transaction as rollback-only.

  – A transaction might end 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 may occur 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 may occur 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. This section covers the following transaction information:

- JTA transactions and two-phase commit protocol
- JTA transactions and conventional transactions
- JTA transactions and crash recovery

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 could commit or rollback. 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 transaction 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 will receive an identifier for the global transaction. Multiple threads of execution process the transaction and have the following effects:

- Records may be locked by a JTA transaction with no user associated with the lock.
- Record locks may 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 present a new recovery requirement in that transactions must be able to be restored to a state pending the end of the two-phase commit. This requirement means enhancing crash recovery by logging 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, a new phase, lock acquisition, is introduced to identify those JTA transactions that need transactional and locking information applied.
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. Specifically, this section covers:

- JTA transactions and database resource planning
- Enabling JTA support
- Disabling JTA support
- Monitoring JTA transactions

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 might not be reused until the prepared JTA transactions have been committed or rolled back. Should this condition persist, BI files might grow considerably.

- **Transaction table usage** — In a database that is not JTA-enabled, the size of the transaction table is a function of the Number of Users (-n) startup parameter. In a JTA-enabled database, the transaction table is made larger by the Maximum Number of JTA Transactions (-maxxids) startup parameter. A JTA-enabled database will use 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 200 for JTA-enabled databases. For tables that are not JTA-enabled, the default is 0. 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
```

The database must be offline when enabling JTA support. Enabling JTA disables after-imaging. This will need to be re-enabled after JTA support is enabled.

For more information on the `proutil` utility, see *OpenEdge Data Management: Database Administration*.

Disabling JTA support

Similarly, the `proutil` utility is used to disable JTA support. The command uses the following syntax:

**Syntax**

```
proutil db-name -C enablejta
```

The database must be offline when disabling JTA support.

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 Table 10–1.

**Table 10–1: JTA transaction states**

<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>

For more information on database utilities and database resource management, see *OpenEdge Data Management: Database Administration*. 

```
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.

This chapter describes when and how to use stored procedures, as detailed in the following sections:

- 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.

Figure 11–1 illustrates the general steps for creating a Java stored procedure.

**Figure 11–1: 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 11–1 shows an excerpt from an ODBC application that calls a stored procedure (order_parts) using the ODBC syntax `{call procedure_name( param)}`.

**Example 11–1: Stored procedure using ODBC syntax**

```c
SQLINTEGER 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.

Figure 11–2 illustrates the steps in executing a stored procedure.
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.

Table 11–1 summarizes the functionality of these OpenEdge SQL-supplied 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 SQLPrepared (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>
</tbody>
</table>
| Exception handling for SQL statements                                        | DhSQLException
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_dec1[ , ... ] ] )
   [ RESULT ( column_name data_type [ , ... ] ) ]
   [ IMPORT java_import_clause ]
BEGIN
   java_snippet
END
```

**Parameter declaration**

```
{ 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. Example 11–2 assumes the existence of a table named HelloWorld, and inserts a quoted string into that table.

Example 11–2: Creating a stored procedure

```java
CREATE PROCEDURE HelloWorld ()
BEGIN
   SQLIStatement Insert_HelloWorld = new SQLIStatement (
      "INSERT INTO HelloWorld(fld1) values ('Hello World!')");
   Insert_HelloWorld.execute();
END;
```

Subsequently, you can execute the procedure shown in Example 11–3.

Example 11–3: Executing a stored procedure

```sql
SQLExplorer> CREATE TABLE helloworld (fld1 CHAR(100));
SQLExplorer> CALL HelloWorld();
0 records returned
SQLExplorer> SELECT * FROM helloworld;
FLD1
----
Hello World!
1 record selected
```

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.

Example 11–4 shows a more complex procedure specification that contains these elements.

Example 11–4: CREATE PROCEDURE statement

```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 Example 11–5.

Example 11–5: CREATE PROCEDURE input script

```
$ sqlexp -infile hello_world_script.sql example_db
```

Example 11–6 illustrates the use of the CREATE PROCEDURE statement in the context of an application call.

Example 11–6: CREATE PROCEDURE in context of application call

```
-- 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:

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 11–7 shows a call to a stored procedure named order_parts that passes a single input parameter using a parameter marker.

**Example 11–7: Stored procedure passing a single output parameter**

```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);
```

**From JDBC**

The JDBC call escape sequence is the same as in ODBC. For example:

**Syntax**

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

Embed the escape sequence in a JDBC CallableStatement.prepareCall method invocation.

Example 11–8 shows the JDBC code parallel to the ODBC code excerpt shown in the previous example.

**Example 11–8: 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 \texttt{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.

\textbf{Stored procedure security}

The following guidelines apply to stored procedure security:

- To create a stored procedure, a user must have \texttt{RESOURCE} or \texttt{DBA} privileges.
- The \texttt{DBA} privilege entitles a user to execute any stored procedure.
- The \texttt{DBA} privilege entitles a user to drop any stored procedure.
- The owner of a stored procedure is given \texttt{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 \texttt{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 \texttt{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 \texttt{EXECUTE} privilege on the procedure.

\textbf{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. \texttt{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 \texttt{INSERT}, \texttt{GRANT}, or \texttt{CREATE}), use the \texttt{SQLStatement} class for one-time execution, or the \texttt{SQLPStatement} class for repeated execution.
- To issue SQL statements that generate results (\texttt{SELECT} and, in some cases, \texttt{CALL}), use the \texttt{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 \texttt{DhSQLResultSet} class to store rows of data in a procedure result set. You must use \texttt{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 \texttt{DhSQLResultSet} methods to store rows of data generated internally by the procedure.

In addition, OpenEdge SQL provides the \texttt{DhSQLException} class so procedures can process and generate Java exceptions through the \texttt{try}, \texttt{catch}, and \texttt{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.

**setParam 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**

```java
setParam ( 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 *n*th.

*value*

Specifies a literal, variable name, or input parameter that contains the value to be assigned to the parameter marker.

Example 11–9 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.

**Example 11–9: Stored procedure using setParam**

```java
SQLIStatement insert_cust = new SQLIStatement ( "INSERT INTO customer VALUES (?,?) ");
insert_cust.setParam (1, cust_number);
insert_cust.setParam (2, cust_name);
```

Example 11–10 shows a procedure using the `is.NULL` method. The `is.NULL` method is used to check for a null value.

**Example 11–10: procedure using is.NULL**

```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**

```
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:

```
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 Example 11–11.

**Example 11–11: Stored procedures input and output parameters**

```java
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 procedures 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 Table 11–2.

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`:

```java
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.setParam(1, in1);
    // In setParam you can use either String or String type
    // for SQL types CHAR, and VARCHAR
    call_proc.setParam(2, out2);
    call_proc.open();
    out1 = (String)call_proc.getParam(1, CHAR);
    // getParam requires String type for CHAR
    out2 = (Long)call_proc.getParam(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 Table 11–2 for declaring parameters and result-set fields in the procedure specification and the Java variables in the procedure body.

**Table 11–2:  Mapping between SQL and Java data types (1 of 2)**

<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><code>set</code>, <code>setParam</code></td>
<td>String</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>All</td>
<td><code>java.math.BigDecimal</code></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>All</td>
<td><code>java.math.BigDecimal</code></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.

---

**Table 11–2: Mapping between SQL and Java data types (2 of 2)**

<table>
<thead>
<tr>
<th>SQL type</th>
<th>Java methods</th>
<th>Java wrapper type</th>
</tr>
</thead>
<tbody>
<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><code>java.sql.Date</code></td>
</tr>
<tr>
<td>TIME</td>
<td>All</td>
<td><code>java.sql.Time</code></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>All</td>
<td><code>java.sql.Timestamp</code></td>
</tr>
</tbody>
</table>
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 Example 11–12, the statement includes five parameter markers.

**Example 11–12: Stored procedure using INSERT statement**

```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 an SQLPStatement object, you can execute the object multiple times, supplying different values for each execution.
Example 11–13 extends the previous example to use prepared execution.

Example 11–13: Stored procedure with prepared execution

```
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.

Example 11–14 uses `SQLCursor` to process the result set returned by an SQL `SELECT` statement.

**Example 11–14: Stored procedure using SQLCursor**

```java
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.

Example 11–15 shows an excerpt from a stored procedure that processes the result set returned by another procedure, `get_customers`.
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.

Example 11–15: Stored procedure processing results of another procedure

```java
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();
```
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.

Example 11–16 uses both techniques to set an SQL input parameter to `NULL`.

Example 11–16: Stored procedure setting input parameter to `NULL`

```java
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.

Example 11–17 illustrates the error returned when the argument to `SQLCursor.getValue` is `NULL`.

Example 11–17: Stored procedure error message

```
(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.
Example 11–18 illustrates using the `wasNULL` method.

Example 11–18: Stored procedure using `wasNULL` method

```java
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 `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 `getDiagnostics` method of the `DhSQLException` class object provides a mechanism to retrieve different details of the error.
The `getDiagnostics` method takes a single argument whose value specifies which error message detail it returns. Table 11–3 shows the explanations of the `getDiagnostics` error-handling options.

### Table 11–3: `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>

**Example 11–19** 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);
    errmsg  = 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.

**Example 11–20** 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.
Example 11–21 illustrates the steps required for calling one Java stored procedure from another.

**Example 11–21: Stored procedure calling 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.setParam(2, inout_param);
    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:

Syntax

```
CREATE TRIGGER [ owner_name. ]trigname
  { BEFORE | AFTER }
  { INSERT | DELETE | UPDATE [ OF ( column_name [ , ... ] ) ] }
ON table_name
  [ REFERENCING { OLDRow | NEWRow | NEWRow , OLDRow | OLDRow , NEWRow } ]
  [ 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.
Example 11–22 shows the elements of a trigger.

**Example 11–22: Trigger elements**

```sql
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...REFERENCING OLDROW 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.getMethod and NEWROW.getMethod 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:

**Syntax**

```java
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 nth.

**sql_data_type**

Specifies the corresponding SQL data type. For a complete list of appropriate data types, see Table 11–2.

Example 11–23 shows an excerpt from a trigger that uses getValue to assign values from both OLDROW and NEWROW objects.

**Example 11–23: Trigger assigning OLDROW and NEWROW values**

```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
        SQLIStatement  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. This chapter contains the following sections:

- 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 JOINs.

- **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. Figure 12–1 illustrates a tree produced for the query.

![Query relational tree model](image-url)
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:

- Pushing restrict operations close to the data origin
- Using indexes for restrictions
- Choosing the best index
- Predicate expressions
- Generating candidate indexes
- Selecting an index
- Join optimization
- Determining join order among adjacent join nodes
- Choosing the join algorithm
- Augmented nested loop join
- Merge join
- Nested loop join
- Sort optimization
- Eliminating redundant sorts
- Converting table scans to index bracket scans
- Indexes to evaluate MAX/MIN functions
- Index bracket scan optimization

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.
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 Example 12–1.

Example 12–1: Query statement optimization

```
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 \(\text{Salary} > 4000\) and \(\text{Salary} \leq 6000\) are moved down the tree, below the join node, since they apply to a single table. The restriction \(\text{Employee.DeptNum} = \text{Department.DeptNum}\) 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:

```
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 Example 12–2.

Example 12–2: Virtual system table query plan

```
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
);
```

Table 12–1 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:

\[ \#\text{rows} = \frac{\text{table cardinality}}{\text{index count of unique values}} \]
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:

**Syntax**

```
UPDATE ( [ TABLE | INDEX | [ ALL ] COLUMN ] STATISTICS [ AND ] ) ... 
[ FOR table_name ]
```

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 ;
```
To create the new index statistics for all tables in a database, simply use the statement shown in Example 12–3.

Example 12–3: 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.
Index

A

ABL brokers 7–2
ALTER TABLE statement 5–5
Altering
   tables 5–5
Application performance
   using stored procedures 11–3
Architecture
   client/server 1–3
   J2EE 10–2
   JDBC 2–2
   JTA 10–3
      multi-threaded 1–4
   ODBC 3–2
Audit Administrator 4–6
Automatic archiving 11–30
Auxiliary databases
   connecting 9–14
      disconnecting 9–14

C

Calling stored procedures 11–4
   from JDBC 11–11
   from ODBC 11–10
Candidate keys 5–16
Cascading deletes 11–30
Cascading privileges 4–10
Cascading updates 11–30
Catalogs
   disconnecting from 9–8
      in multi-database queries 9–3
CLASSPATH 2–5
Clients
   JDBC 2–2
   ODBC 3–1
Column authorization system table 4–9
Column widths 7–11
Columns 7–8
CONNECT AS CATALOG statement 9–8
Connecting
   database servers 7–2
      multi-databases 9–6
Constraints
   check 5–13
      column-level check 5–14
   foreign key 5–16
   integrity 5–12
   primary key 5–15
      referential 5–16
      table-level check 5–14
Conventional identifiers 7–9
Crash recovery
   JTA transactions 10–8
CREATE INDEX statement 5–7
Index

CREATE PROCEDURE statement 11–3, 11–7
CREATE TABLE statement 5–3
CREATE TRIGGER
  in procedures 11–28
  syntax 11–27
CREATE USER statement 4–5
CREATE VIEW statement 5–8, 5–9

Creating
  indexes 5–7
  stored procedures 11–3
  tables 5–3
  users 4–5
  views 5–8, 5–9

D
Data Control Language 4–1
Data Definition Language 5–1
Data integrity 5–12
Data Manipulation Language 6–2
Data sources
  adding 3–4
  administration 3–4
  configuring 3–4
  configuring for ODBC 3–4
  ODBC UNIX configuration 3–9
  ODBC Windows configuration 3–4

Data type compatibility 7–10
Data types
  conversion between SQL and Java 11–15
  mapping between SQL and Java 11–15

Database
  connecting from Java application 2–7
  connecting using SQL Explorer 2–6
  connecting with JDBC driver 2–6
  connection problems 2–7
  multiple connections 9–6
  Progress 4GL objects 7–8
  SQL objects 7–8
  structure 7–7

Database administrators
  creating 4–4
  responsibilities 4–4

Database authorization system table 4–9

Database objects
  naming 7–9

Database security
  basics 4–2
  using stored procedures 11–12

Database servers
  connection requirements 7–2

DBTool utility 7–11

Default Isolation Levels
  setting for ODBC in Windows 3–7

DELETE statement 6–5

Deleting
  data 6–5
  stored procedures 11–11

Delimited identifiers 7–10

Dirty Read 8–4

DISCONNECT CATALOG statement 9–8

DLC directory 2–5

Drivers
  JDBC 2–2
  ODBC 3–2

DROP INDEX statement 5–7

DROP TABLE statement 5–6

Dropping
  indexes 5–7
  tables 5–6
  views 5–8, 5–9

Dumping
  SQLDUMP utility 5–20

Encrypted connections 7–4

Environment variables
  Java character environment 2–5
  Java Windows environment 2–5
  setting for JDBC driver 2–5
  setting for ODBC driver 3–9

Errors
  in stored procedures 11–23

Executing
  SQL statements 11–16
  stored procedures 11–4
F
Fetch array size 3–7
Foreign key constraints 5–16

G
getValue
    definition 11–13
getValue method 11–13
GRANT RESOURCE statement 4–7
GRANT statement 4–6
Granting
    public access 4–8
    resource privileges 4–7
    update privileges 4–8

I
Identifiers
    conventional 7–9
    delimited 7–10
Import clauses 11–8
Index statistics
    overview 12–12
    updating 12–13
Indexes
    creating 5–7
    dropping 5–7
    system catalogs 6–6
    using 6–6
Inner joins 6–7
Input parameters
    in stored procedures 11–14
INSERT statement 6–3
Inserting data 6–3
Internet protocol support 2–4
Invoking stored procedures 11–10
Isolation levels
    ABL 7–14
    dirty read 8–4
    nonrepeatable read 8–4
    phantom read 8–5
    setting 8–5

J
J2EE 10–2
Java
    application programming interface 2–2
    database connection 2–7
    methods 11–15
    native interface 2–2
    source text 11–3
    using with stored procedures 11–3
    wrapper types 11–15
Java classes
    SQLCursor 11–12
    SQLIstatement 11–12
    summary 11–6
Java Database Connectivity 2–2
Java snippets 11–7
Java Transaction API 10–1
Java Virtual Machine 11–4
JDBC
    API support 2–3
    architecture 2–2
    client 2–2
    components 2–3
    driver 2–2
    files 2–3
    invoking stored procedures 11–11
JDBC Driver
    architecture 2–2
    components 2–3
    files and locations 2–3
    overview 2–2
jdbc.jar file 2–3
JdbcProgress file 2–3
Join operations 6–7
Join optimization 12–7
JTA 10–1
    data resource planning 10–9
    distributed transaction process 10–6
    monitoring transactions 10–10
    role in J2EE 10–2

L
Library path 2–5
Loading
    ABL 5–23
Index

Lock modes
overview 8–7

Locking
ABL 7–14
acquisition 8–9
database performance 8–12
hints 8–10
levels 8–8
lock modes 8–7
record locks 8–9
schema locks 8–9
table locks 8–9

Logon validation 7–5

M

Modifying
stored procedures 11–11

Multi-database queries
limitations 9–4
overview 9–2
process 9–2
working with catalogs 9–3

Multiple-database queries
using properties files 9–8

Multithreaded architecture 1–4

N

NEWROW objects 11–30

Nonrepeatable read 8–4

O

Objects
NEWROW 11–30
OLDROW 11–30

ODBC
applications 3–2
architecture 3–2
connection testing in UNIX 3–13
connection testing in Windows 3–4
data source administrator 3–4
driver manager 3–2
driver overview 3–2
driver set up 3–4
invoking stored procedures 11–10
ODBC driver 3–2
ODBC.INI tags 3–11
OLDROW objects 11–30

OpenEdge SQL
client/server architecture 1–2
gine 1–2
interaction with Java 11–3
overview 1–2

OpenEdge SQL brokers 7–2

Optimization
affecting output 12–12
inspecting 12–11
joins 12–7
methods 12–2
phases 12–5
sorting 12–9

Outer joins 6–8

Output parameters
in stored procedures 11–14

P

Parameter declarations 11–8

Phantom read 8–5

Primary key 5–15

Privileges
database-wide 4–6
granting 4–6
revoking 4–10
table-specific 4–6
verifying 4–9

Procedure result set declarations 11–8

PROMON utility 8–12

Properties files
creating 9–9
using with multi-database queries 9–8

PUB schema 6–3

Q

Query optimizer 12–2

R

READPAST 8–10

Reference cycle
inserting rows 5–18
single table 5–18
table 5–18

Referential constraints 5–16

Index–4
Resources
  granting privileges 4–7
Restricted privileges 4–10
REVOKE statement 4–10
Revoking privileges 4–10
ROLLBACK statement 8–2
Rows 7–8

S
Schemas 7–8

Security
  OpenEdge SQL 4–2
  using stored procedures 11–12
SELECT statement 6–2
Selecting data 6–2
SET CATALOG statement 9–7
setParam
  definition 11–13
  syntax 11–13
SHOW CATALOG statement 9–7
Single table reference cycle 5–18
SQL Engine 1–2
  using stored procedures 11–3
SQL Explorer 2–6
SQL statements
  executing 11–16
  immediate execution 11–16
  multi-database queries 9–6
  prepared execution 11–17
SQL utilities 5–20
SQLDUMP Utility 5–20
SQLLOAD Utility 5–23
SQLSchema Utility 5–25
SSL 7–4
Statement parser 12–4

Statements
  ALTER TABLE 5–5
  ALTER USER 4–5
  COMMIT 8–2
  CONNECT AS CATALOG 9–8
  CREATE PROCEDURE 11–3, 11–7
  CREATE TABLE 5–3
  CREATE TRIGGER 11–27
  CREATE VIEW 5–8
  DELETE 6–5
  DISCONNECT CATALOG 9–8
  DROP TABLE 5–6
  DROP VIEW 5–8
  GRANT 4–7
  GRANT RESOURCE 4–7
  INSERT 6–3, 9–7, 9–8
  REVOKE 4–10
  ROLLBACK 8–2
  SELECT 6–2
  SET CATALOG 9–7
  SHOW CATALOGS 9–7
  UPDATE 6–4
  UPDATE STATISTICS 12–13

Stored procedures
  advantages 11–3
  basics 11–7
  calling 11–4
  creating 11–3
  deleting 11–11
  enabling on 64-bit platforms 11–2
  executing 11–4
  handling errors 11–23
  handling NULL values 11–21
  input and output parameters 11–14
  invoking 11–10
  Java definition 11–3
  jdk requirements 11–2
  modifying 11–11
  passing values 11–13
  result sets 11–20
  retrieving data 11–18
  security 11–12
  structure 11–8
  using 11–6
  writing 11–10

Summation updates 11–30

Syntax
  ALTER TABLE 5–5
  ALTER USER 4–5
  CALL to stored procedure (JDBC) 11–11
  CALL to stored procedure (ODBC) 11–10
  COMMIT 8–2
  CONNECT AS CATALOG 9–8
  CREATE INDEX 5–7
  CREATE PROCEDURE 11–7
  CREATE TABLE 5–3
  CREATE USER 4–5
  DELETE 6–5
  DISCONNECT CATALOG 9–8
  DROP INDEX 5–7
  DROP TABLE 5–6
  getValue 11–14
  GRANT 4–7
  INSERT 6–3, 9–7, 9–8
  joins 6–7
Index

REVOKE 4–10
ROLLBACK 8–2
SELECT 6–2
SET CATALOG 9–7
setParam 11–13
SHOW CATALOGS 9–7
UPDATE 6–4
UPDATE STATISTICS 12–13

System tables
   column authorization 4–9
database authorization 4–9
indexes 6–6
table authorization 4–9

T

Table authorization system table 4–9
Table reference cycle 5–18
Tables 7–8
   altering 5–5
   creating 5–3
   dropping 5–6
Transaction control 8–2

Transactions
   committing 8–2
   conventional versus JTA 10–8
   JTA 10–6
   JTA and crash recovery 10–8
   locking 8–7
   rolling back 8–2
   with isolation levels 8–4

Triggers
   and stored procedures 11–29
   creating 11–27
   overview 11–27
   passing values 11–30
   setting up OpenEdge SQL 11–2
   structure 11–28
   types 11–30
   using with ABL 7–13
   with constraints 11–29

Type 2 driver 2–2

U

UNIX
   ODBC configuration 3–9
      testing ODBC connections 3–13
   UPDATE statement 6–4
   UPDATE STATISTICS command 12–12
   Updating data 6–4
   User accounts 7–5
   User privileges 7–5
   Users
      creating 4–4

Utilities
   SQLDUMP 5–20
   SQLLOAD 5–20
   SQLSCHEMA 5–20

W

Windows
   adding new data sources 3–4
   JDBC environment variables 2–5
   ODBC administrator 3–4
   testing ODBC connections 3–4
   Wrapper code 11–7

X

XAConnection 10–4
XDataSource 10–4
XAResource 10–4
XAResource methods 10–4