Pervasive.SQL 2000i

SQL Engine Reference
Reference for Using SQL with Pervasive.SQL 2000 Service Pack 3

Pervasive Software, Inc.
12365 Riata Trace Parkway
Building II
Austin, TX 78727 USA

Telephone: +1 512 231 6000 or 800 287 4383
Fax: +1 512 231 6010
E-Mail: info@pervasive.com
Web: http://www.pervasive.com
disclaimer

PERVASIVE SOFTWARE INC. LICENSES THE SOFTWARE AND DOCUMENTATION PRODUCT TO YOU OR YOUR COMPANY SOLELY ON AN "AS IS" BASIS AND SOLELY IN ACCORDANCE WITH THE TERMS AND CONDITIONS OF THE ACCOMPANYING LICENSE AGREEMENT. PERVASIVE SOFTWARE INC. MAKES NO OTHER WARRANTIES WHATSOEVER, EITHER EXPRESS OR IMPLIED, REGARDING THE SOFTWARE OR THE CONTENT OF THE DOCUMENTATION; PERVASIVE SOFTWARE INC. HEREBY EXPRESSLY STATES AND YOU OR YOUR COMPANY ACKNOWLEDGES THAT PERVASIVE SOFTWARE INC. DOES NOT MAKE ANY WARRANTIES, INCLUDING, FOR EXAMPLE, WITH RESPECT TO MERCHANTABILITY, TITLE, OR FITNESS FOR ANY PARTICULAR PURPOSE OR ARISING FROM COURSE OF DEALING OR USAGE OF TRADE, AMONG OTHERS.

trademarks

Btrieve, Tango, Client/Server in a Box, and the Pervasive Software logo are registered trademarks of Pervasive Software Inc.
Built on Pervasive, Built on Pervasive Software, Extranet in a Box, Pervasive.SQL, Jtrieve, Plug n’ Play Databases, SmartScout, Solution Network, Ultra-light Z-DBA, Z-DBA, ZDBA, UltraLight, MicroKernel Database Engine, and MicroKernel Database Architecture are trademarks of Pervasive Software Inc.
Microsoft, MS-DOS, Windows, Windows NT, Win32, Win32s, and Visual Basic are registered trademarks of Microsoft Corporation.
Windows 95 is a trademark of Microsoft Corporation.
NetWare and Novell are registered trademarks of Novell, Inc.
NetWare Loadable Module, NLM, Novell DOS, Transaction Tracking System, and TTS are trademarks of Novell, Inc.
All other company and product names are the trademarks or registered trademarks of their respective companies.

© Copyright 2001 Pervasive Software Inc. All rights reserved. Reproduction, photocopying, or transmittal of this publication, or portions of this publication, is prohibited without the express prior written consent of the publisher.
This product includes software developed by Powerdog Industries.
© Copyright 1994 Powerdog Industries. All rights reserved.
The ODBC Driver Manager for NetWare (ODBC.NLM) included in this product is based on the GNU iODBC software © Copyright 1995 by Ke Jin <kejin@empress.com> and was modified by Simba Technologies Inc. in June 1999.
This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
This library is distributed 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. See the GNU Lesser General Public License for more details.
A copy of the GNU Lesser General Public License is included in your installation of Pervasive.SQL 2000 at \pvsw\doc\lesser.htm. If you cannot find this license, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA. You may contact Pervasive Software Inc. using the contact information on the back cover of this manual.

SQL Engine Reference
March 2001
100-003673-004
## Contents

**About This Manual** ............................................................ xi
Who Should Read this Manual ................................................. xii
Manual Organization .............................................................. xiii
Conventions .............................................................................. xiv
For More Information ............................................................. xv

### 1 SQL Overview ................................................................. 1-1

*An Overview of the Structured Query Language (SQL)*

- Data Definition Statements .................................................. 1-3
  - Creating, Modifying, and Deleting Tables ............................... 1-3
  - Creating and Deleting Indexes .............................................. 1-4
  - Creating and Deleting Triggers ............................................ 1-4
  - Creating and Deleting Stored Procedures ............................... 1-5
- Data Manipulation Statements .................................................. 1-6
  - Retrieving Data .................................................................. 1-6
  - Modifying Data .................................................................... 1-7
  - Defining Transactions .......................................................... 1-7
  - Creating and Deleting Views ................................................ 1-8
  - Executing Stored Procedures ................................................. 1-8
  - Executing Triggers ............................................................... 1-8
- Data Control Statements .......................................................... 1-9
  - Enabling and Disabling Security ......................................... 1-9
  - Creating and Deleting Users and Groups ............................... 1-10
  - Granting and Revoking Rights .............................................. 1-10
- Database Names ................................................................... 1-11

### 2 ODBC Engine Reference .................................................. 2-1

*A Reference to Pervasive.SQL Supported Syntax*

- Pervasive ODBC Engine Interface Limits .................................. 2-2
- Data Source Name Connection String Keywords ....................... 2-3
- ODBC API Conformance ......................................................... 2-5
  - Exceptions to ODBC API Conformance ................................. 2-7
- SQL Grammar Conformance ..................................................... 2-11
  - Delimited Identifiers in SQL Statements ............................... 2-13
  - Global Variables .................................................................. 2-13
- SQL Grammar Elements ......................................................... 2-16
  - ADD ................................................................................. 2-17
  - ALL .................................................................................. 2-18
  - ALTER TABLE ................................................................. 2-19
  - ANY ................................................................................. 2-30
<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS</td>
<td>2-31</td>
</tr>
<tr>
<td>BEGIN [ATOMIC]</td>
<td>2-32</td>
</tr>
<tr>
<td>CALL</td>
<td>2-33</td>
</tr>
<tr>
<td>CASCADE</td>
<td>2-34</td>
</tr>
<tr>
<td>CASE</td>
<td>2-35</td>
</tr>
<tr>
<td>CLOSE</td>
<td>2-36</td>
</tr>
<tr>
<td>COMMIT</td>
<td>2-37</td>
</tr>
<tr>
<td>CREATE GROUP</td>
<td>2-39</td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>2-40</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>2-42</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>2-50</td>
</tr>
<tr>
<td>CREATE TRIGGER</td>
<td>2-59</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>2-62</td>
</tr>
<tr>
<td>DECLARE</td>
<td>2-64</td>
</tr>
<tr>
<td>DECLARE CURSOR</td>
<td>2-65</td>
</tr>
<tr>
<td>DELETE (positioned)</td>
<td>2-66</td>
</tr>
<tr>
<td>DELETE</td>
<td>2-67</td>
</tr>
<tr>
<td>DISTINCT</td>
<td>2-68</td>
</tr>
<tr>
<td>DROP GROUP</td>
<td>2-69</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>2-70</td>
</tr>
<tr>
<td>DROP PROCEDURE</td>
<td>2-71</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>2-72</td>
</tr>
<tr>
<td>DROP TRIGGER</td>
<td>2-73</td>
</tr>
<tr>
<td>DROP VIEW</td>
<td>2-74</td>
</tr>
<tr>
<td>END</td>
<td>2-75</td>
</tr>
<tr>
<td>EXISTS</td>
<td>2-76</td>
</tr>
<tr>
<td>FETCH</td>
<td>2-77</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>2-78</td>
</tr>
<tr>
<td>GRANT</td>
<td>2-80</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>2-85</td>
</tr>
<tr>
<td>HAVING</td>
<td>2-87</td>
</tr>
<tr>
<td>IF</td>
<td>2-88</td>
</tr>
<tr>
<td>IN</td>
<td>2-89</td>
</tr>
<tr>
<td>INSERT</td>
<td>2-90</td>
</tr>
<tr>
<td>JOIN</td>
<td>2-94</td>
</tr>
<tr>
<td>LEAVE</td>
<td>2-101</td>
</tr>
<tr>
<td>LOOP</td>
<td>2-102</td>
</tr>
<tr>
<td>NOT</td>
<td>2-103</td>
</tr>
<tr>
<td>OPEN</td>
<td>2-104</td>
</tr>
<tr>
<td>PRIMARY KEY</td>
<td>2-105</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>2-107</td>
</tr>
<tr>
<td>PRINT</td>
<td>2-108</td>
</tr>
<tr>
<td>RELEASE SAVEPOINT</td>
<td>2-109</td>
</tr>
<tr>
<td>RESTRICT</td>
<td>2-111</td>
</tr>
<tr>
<td>REVOKE</td>
<td>2-112</td>
</tr>
</tbody>
</table>
Contents

<table>
<thead>
<tr>
<th>XSIndex</th>
<th>D-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSAttrib.</td>
<td>D-11</td>
</tr>
<tr>
<td>X$View</td>
<td>D-13</td>
</tr>
<tr>
<td>X$Proc</td>
<td>D-14</td>
</tr>
<tr>
<td>X$User</td>
<td>D-15</td>
</tr>
<tr>
<td>XSRights</td>
<td>D-17</td>
</tr>
<tr>
<td>X$Relate.</td>
<td>D-19</td>
</tr>
<tr>
<td>XS$Trigger</td>
<td>D-21</td>
</tr>
<tr>
<td>XS$Depend</td>
<td>D-23</td>
</tr>
</tbody>
</table>
### Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>SQL Statement Types and Related Tasks</td>
<td>1-2</td>
</tr>
<tr>
<td>1-2</td>
<td>Data Definition Statements - Tables</td>
<td>1-3</td>
</tr>
<tr>
<td>1-3</td>
<td>Data Definition Statements - Indexes</td>
<td>1-4</td>
</tr>
<tr>
<td>1-4</td>
<td>Data Definition Statements - Triggers</td>
<td>1-4</td>
</tr>
<tr>
<td>1-5</td>
<td>Data Definition Statements - Trigger Control</td>
<td>1-4</td>
</tr>
<tr>
<td>1-6</td>
<td>Data Definition Statements - Stored Procedure</td>
<td>1-5</td>
</tr>
<tr>
<td>1-7</td>
<td>Data Definition Statements - Stored Procedure Control</td>
<td>1-5</td>
</tr>
<tr>
<td>1-8</td>
<td>Data Manipulation Statements - Retrieving Data</td>
<td>1-6</td>
</tr>
<tr>
<td>1-9</td>
<td>Data Manipulation Statements - Retrieving Data Options</td>
<td>1-7</td>
</tr>
<tr>
<td>1-10</td>
<td>Data Manipulation Statements - Modifying Data</td>
<td>1-7</td>
</tr>
<tr>
<td>1-11</td>
<td>Data Manipulation Statements - Views</td>
<td>1-8</td>
</tr>
<tr>
<td>1-12</td>
<td>Data Manipulation Statements - Stored Procedures</td>
<td>1-8</td>
</tr>
<tr>
<td>1-13</td>
<td>Data Control Statements - Security</td>
<td>1-9</td>
</tr>
<tr>
<td>1-14</td>
<td>Data Control Statements - Groups and Users</td>
<td>1-10</td>
</tr>
<tr>
<td>1-15</td>
<td>Data Control Statements - Rights</td>
<td>1-10</td>
</tr>
<tr>
<td>2-1</td>
<td>Valid Connection Strings for Client DSNs</td>
<td>2-4</td>
</tr>
<tr>
<td>2-2</td>
<td>Valid Connection Strings for Engine DSNs</td>
<td>2-4</td>
</tr>
<tr>
<td>2-3</td>
<td>Interface-supported ODBC API Functions</td>
<td>2-5</td>
</tr>
<tr>
<td>2-4</td>
<td>Options for SQLSetStmtOption and SQLGetStmtOption</td>
<td>2-8</td>
</tr>
<tr>
<td>2-5</td>
<td>Options for SQLSetConnectOption and SQLGetConnectOption</td>
<td>2-9</td>
</tr>
<tr>
<td>2-6</td>
<td>SQL Grammar Conformance</td>
<td>2-11</td>
</tr>
<tr>
<td>2-7</td>
<td>Emp Table</td>
<td>2-96</td>
</tr>
<tr>
<td>2-8</td>
<td>Dept Table</td>
<td>2-96</td>
</tr>
<tr>
<td>2-9</td>
<td>Addr Table</td>
<td>2-96</td>
</tr>
<tr>
<td>2-10</td>
<td>Loc Table</td>
<td>2-96</td>
</tr>
<tr>
<td>2-11</td>
<td>Two-way Left Outer Join</td>
<td>2-97</td>
</tr>
<tr>
<td>2-12</td>
<td>Three-way Radiating Left Outer Join</td>
<td>2-98</td>
</tr>
<tr>
<td>2-13</td>
<td>Three-way Chaining Left Outer Join</td>
<td>2-99</td>
</tr>
<tr>
<td>2-14</td>
<td>Three-way Radiating Left Outer Join, Less Optimized</td>
<td>2-99</td>
</tr>
<tr>
<td>2-15</td>
<td>Addr Table</td>
<td>2-127</td>
</tr>
<tr>
<td>2-16</td>
<td>Loc Table</td>
<td>2-127</td>
</tr>
<tr>
<td>2-17</td>
<td>SELECT Statement with Cartesian JOIN</td>
<td>2-128</td>
</tr>
<tr>
<td>2-18</td>
<td>String Functions</td>
<td>2-156</td>
</tr>
<tr>
<td>2-19</td>
<td>Numeric Functions</td>
<td>2-159</td>
</tr>
<tr>
<td>2-20</td>
<td>Time and Date Functions</td>
<td>2-162</td>
</tr>
<tr>
<td>2-21</td>
<td>System Functions</td>
<td>2-165</td>
</tr>
<tr>
<td>2-22</td>
<td>Logical Functions</td>
<td>2-165</td>
</tr>
<tr>
<td>2-23</td>
<td>Conversion Function</td>
<td>2-167</td>
</tr>
</tbody>
</table>
### Tables

<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1</td>
<td>Pervasive.SQL Data Types</td>
</tr>
<tr>
<td>A-2</td>
<td>Fully Supported Data Types</td>
</tr>
<tr>
<td>A-3</td>
<td>Partially Supported Data Types</td>
</tr>
<tr>
<td>A-4</td>
<td>Legacy Data Types</td>
</tr>
<tr>
<td>A-5</td>
<td>Pervasive.SQL to ODBC Data Type Mapping</td>
</tr>
<tr>
<td>A-6</td>
<td>Data Type Valid Lengths and Value Ranges</td>
</tr>
<tr>
<td>A-7</td>
<td>Infinity Representation</td>
</tr>
<tr>
<td>A-8</td>
<td>Btrieve Extended Key Types and Codes</td>
</tr>
<tr>
<td>A-9</td>
<td>INTEGER Key Type</td>
</tr>
<tr>
<td>A-10</td>
<td>Rightmost Digit with Embedded Sign</td>
</tr>
<tr>
<td>A-11</td>
<td>TIMESTAMP Data Type</td>
</tr>
<tr>
<td>B-1</td>
<td>SQL Reserved Words and Symbols</td>
</tr>
<tr>
<td>C-1</td>
<td>Connection and Session Control APIs</td>
</tr>
<tr>
<td>C-2</td>
<td>Preparing and Executing SQL Request APIs</td>
</tr>
<tr>
<td>C-3</td>
<td>Data Retrieval APIs</td>
</tr>
<tr>
<td>C-4</td>
<td>Statement Termination APIs</td>
</tr>
<tr>
<td>C-5</td>
<td>Database and Driver Information APIs</td>
</tr>
<tr>
<td>C-6</td>
<td>Metadata Information APIs</td>
</tr>
<tr>
<td>C-7</td>
<td>Transaction APIs</td>
</tr>
<tr>
<td>C-8</td>
<td>Deprecated Scalable SQL APIs</td>
</tr>
<tr>
<td>D-1</td>
<td>System Tables</td>
</tr>
<tr>
<td>D-2</td>
<td>X$File System Table Structure</td>
</tr>
<tr>
<td>D-3</td>
<td>X$Field System Table Index Definitions</td>
</tr>
<tr>
<td>D-4</td>
<td>X$Index System Table Structure</td>
</tr>
<tr>
<td>D-5</td>
<td>X$Field System Table Index Definitions</td>
</tr>
<tr>
<td>D-6</td>
<td>X$Index System Table Index Definitions</td>
</tr>
<tr>
<td>D-7</td>
<td>X$Attrib System Table Structure</td>
</tr>
<tr>
<td>D-8</td>
<td>X$Attrib System Table Index Definitions</td>
</tr>
<tr>
<td>D-9</td>
<td>X$View System Table Structure</td>
</tr>
<tr>
<td>D-10</td>
<td>X$View System Table Index Definitions</td>
</tr>
<tr>
<td>D-11</td>
<td>X$Proc System Table Structure</td>
</tr>
<tr>
<td>D-12</td>
<td>X$Proc System Table Index Definitions</td>
</tr>
<tr>
<td>D-13</td>
<td>X$User System Table Structure</td>
</tr>
<tr>
<td>D-14</td>
<td>X$User System Table Index Definitions</td>
</tr>
<tr>
<td>D-15</td>
<td>Xu$Flags System Table Bit Position Definitions</td>
</tr>
<tr>
<td>D-16</td>
<td>Xu$Rights System Table Index Definitions</td>
</tr>
<tr>
<td>D-17</td>
<td>Xu$Rights System Table Index Definitions</td>
</tr>
<tr>
<td>D-18</td>
<td>Xr$Rights System Table Bit Position Definitions</td>
</tr>
<tr>
<td>D-19</td>
<td>X$Rights System Table Index Definitions</td>
</tr>
<tr>
<td>D-20</td>
<td>X$Relate System Table Structure</td>
</tr>
<tr>
<td>D-21</td>
<td>X$Relate System Table Index Definitions</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>D-23</td>
<td>X$Trigger System Table Structure</td>
</tr>
<tr>
<td>D-24</td>
<td>X$Trigger System Table Index Definitions</td>
</tr>
<tr>
<td>D-25</td>
<td>X$Depend System Table Structure</td>
</tr>
<tr>
<td>D-26</td>
<td>X$Depend System Table Index Definitions</td>
</tr>
</tbody>
</table>
About This Manual

This manual contains the reference material you need for understanding the language functionality and limitations of Pervasive.SQL 2000i.
Who Should Read this Manual

This manual provides information for using Pervasive.SQL 2000i.

This manual assumes you have a general understanding of ODBC architecture and ODBC driver components, and have access to the Microsoft ODBC Software Development Kit.

This document also assumes you have a working understanding of modern database principles and terminology, the C language, and your development environment (compiler and linker).

Pervasive Software would appreciate your comments and suggestions about this manual. As a user of our documentation, you are in a unique position to provide ideas that can have a direct impact on future releases of this and other manuals. If you have comments or suggestions for the product documentation, post your request at http://www.pervasive.com/devtalk or send e-mail to docs@pervasive.com.
Manual Organization

This reference includes the following chapters:

- **Chapter 1 — “SQL Overview”**
  This chapter describes the types of SQL statements you can create using Pervasive.SQL 2000.

- **Chapter 2 — “ODBC Engine Reference”**
  This chapter describes the Pervasive.SQL 2000 Engine’s capabilities, characteristics, and conformance to the SQL grammar and ODBC API standards.

- **Appendix A — “Data Types”**
  This appendix contains tables of supported Data Types.

- **Appendix B — “SQL Reserved Words”**
  This appendix contains the list of supported SQL Keyword.

- **Appendix C — “SQL API Mapping to ODBC”**
  This appendix contains tables of Scalable SQL and ODBC functions.

- **Appendix D — “System Tables”**
  This appendix describes the system tables that comprise the relational data dictionary.

The manual also contains an index.
About This Manual

Conventions

Unless otherwise noted, command syntax, code, and examples use the following conventions:

<table>
<thead>
<tr>
<th>Case</th>
<th>Commands and reserved words typically appear in uppercase letters. Unless the manual states otherwise, you can enter these items using uppercase, lowercase, or both. For example, you can type MYPROG, myprog, or MYprog.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Square brackets enclose optional information, as in [log_name]. If information is not enclosed in square brackets, it is required.</td>
</tr>
<tr>
<td>l</td>
<td>A vertical bar indicates a choice of information to enter, as in [file name</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Angle brackets enclose multiple choices for a required item, as in /D=5</td>
</tr>
<tr>
<td>variable</td>
<td>Words appearing in italics are variables that you must replace with appropriate values, as in file name.</td>
</tr>
<tr>
<td>...</td>
<td>An ellipsis following information indicates you can repeat the information more than one time, as in [parameter ...].</td>
</tr>
<tr>
<td>::=</td>
<td>The symbol ::= means one item is defined in terms of another. For example, a::=b means the item a is defined in terms of b.</td>
</tr>
<tr>
<td>*</td>
<td>An asterisk is used as a wildcard symbol to indicate a series of APIs with the same prefix.</td>
</tr>
</tbody>
</table>

Note Unless otherwise noted, all references in this book to the Pervasive.SQL product refer to the current version, Pervasive.SQL 2000i.
For More Information

For complete information on the ODBC 2.5 specification, see the Microsoft ODBC Programmer’s Reference.
Structured Query Language (SQL) is a database language consisting of English-like statements you can use to perform database operations. Both the American National Standards Institute (ANSI) and IBM have defined standards for SQL. (The IBM standard is the Systems Application Architecture [SAA].) The Pervasive.SQL product implements most of the features of both ANSI SQL and IBM SAA SQL and provides additional extensions that neither standard specifies.

Pervasive.SQL allows you to create different types of SQL statements. The following table lists the types of SQL statements you can create and the tasks you can accomplish using each type of statement:
Table 1-1  SQL Statement Types and Related Tasks

<table>
<thead>
<tr>
<th>SQL Statement Type</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Definition</strong></td>
<td>Create, modify, and delete tables.</td>
</tr>
<tr>
<td></td>
<td>Create and delete indexes.</td>
</tr>
<tr>
<td></td>
<td>Create and delete stored SQL procedures.</td>
</tr>
<tr>
<td></td>
<td>Create and delete triggers.</td>
</tr>
<tr>
<td><strong>Data Manipulation</strong></td>
<td>Retrieve, insert, update, and delete data in tables.</td>
</tr>
<tr>
<td></td>
<td>Define transactions.</td>
</tr>
<tr>
<td></td>
<td>Define and delete views.</td>
</tr>
<tr>
<td></td>
<td>Execute stored SQL procedures.</td>
</tr>
<tr>
<td></td>
<td>Execute triggers.</td>
</tr>
<tr>
<td><strong>Data Control</strong></td>
<td>Enable and disable security for a dictionary.</td>
</tr>
<tr>
<td></td>
<td>Create and delete users and groups.</td>
</tr>
<tr>
<td></td>
<td>Grant and revoke table access rights.</td>
</tr>
</tbody>
</table>

The rest of this chapter briefly describes the SQL statements used in each statement category. For detailed information about each statement, refer to “ODBC Engine Reference” on page 2-1.

The following are the statement category overview sections found in this chapter:

- “Data Definition Statements” on page 1-3
- “Data Manipulation Statements” on page 1-6
- “Data Control Statements” on page 1-9
- “Database Names” on page 1-11
Data Definition Statements

Data definition statements let you specify the characteristics of your database. When you execute data definition statements, Pervasive.SQL stores the description of your database in a data dictionary. You must define your database in the dictionary before you can store or retrieve information.

Pervasive.SQL allows you to construct data definition statements to do the following:

- Create, modify, and delete tables.
- Create and delete indexes.
- Create and delete triggers.
- Create and delete stored procedures.

The following sections briefly describe the SQL statements associated with each of these tasks. For general information about defining the characteristics of your database, refer to the *Pervasive.SQL Programmer’s Guide*, available in the Pervasive.SQL Software Developer Kit (SDK).

Creating, Modifying, and Deleting Tables

You can create, modify, and delete tables from a database by constructing statements using the following statements:

*Table 1-2 Data Definition Statements - Tables*

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE</td>
<td>Defines a table and optionally creates the corresponding data file.</td>
</tr>
<tr>
<td>ALTER TABLE</td>
<td>Changes a table definition. With an ALTER TABLE statement, you can perform</td>
</tr>
<tr>
<td></td>
<td>such actions as add a column to the table definition, remove a column from</td>
</tr>
<tr>
<td></td>
<td>the table definition, change a column’s data type or length (or other</td>
</tr>
<tr>
<td></td>
<td>characteristics), and add or remove a primary key or a foreign key and</td>
</tr>
<tr>
<td></td>
<td>associate the table definition with an different data file.</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>Deletes a table from the data dictionary and optionally deletes the</td>
</tr>
<tr>
<td></td>
<td>associated data file from the disk.</td>
</tr>
</tbody>
</table>
**SQL Overview**

**Creating and Deleting Indexes**

You can create and delete indexes from a database by constructing statements using the following statements:

*Table 1-3  Data Definition Statements - Indexes*

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE INDEX</td>
<td>Defines a new index (a named index) for an existing table.</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>Deletes a named index.</td>
</tr>
</tbody>
</table>

**Creating and Deleting Triggers**

You can create and delete triggers from a database by constructing statements using the following statements:

*Table 1-4  Data Definition Statements - Triggers*

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TRIGGER</td>
<td>Defines a trigger for an existing table.</td>
</tr>
<tr>
<td>DROP TRIGGER</td>
<td>Deletes a trigger.</td>
</tr>
</tbody>
</table>

Pervasive.SQL provides additional SQL control statements, which you can only use in the body of a trigger. You can use the following statements in triggers:

*Table 1-5  Data Definition Statements - Trigger Control*

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE</td>
<td>Defines the trigger execution before the INSERT, UPDATE, or DELETE operation.</td>
</tr>
<tr>
<td>AFTER</td>
<td>Defines the trigger execution after the INSERT, UPDATE, or DELETE operation.</td>
</tr>
</tbody>
</table>
A stored procedure consists of statements you can precompile and save in the dictionary. To create and delete stored procedures, construct statements using the following:

**Table 1-6  Data Definition Statements - Stored Procedure**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE PROCEDURE</td>
<td>Stores a new procedure in the data dictionary.</td>
</tr>
<tr>
<td>DROP PROCEDURE</td>
<td>Deletes a stored procedure from the data dictionary.</td>
</tr>
</tbody>
</table>

Pervasive.SQL provides additional SQL control statements, which you can only use in the body of a stored procedure. You can use the following statements in stored procedures:

**Table 1-7  Data Definition Statements - Stored Procedure Control**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF...THEN...ELSE</td>
<td>Provides conditional execution based on the truth value of a condition.</td>
</tr>
<tr>
<td>LEAVE</td>
<td>Continues execution by leaving a block or loop statement.</td>
</tr>
<tr>
<td>LOOP</td>
<td>Repeats the execution of a block of statements.</td>
</tr>
<tr>
<td>WHILE</td>
<td>Repeats the execution of a block of statements while a specified condition is true.</td>
</tr>
</tbody>
</table>
Data Manipulation Statements

Data manipulation statements let you access and modify the contents of your database. Pervasive.SQL allows you to construct data manipulation statements to do the following:

- Retrieve data from tables.
- Modify data in tables.
- Define transactions.
- Create and delete views.
- Execute stored procedures.
- Execute triggers.

The following sections briefly describe the SQL statements associated with each of these tasks.

Retrieving Data

All statements you use to retrieve information from a database are based on the SELECT statement.

Table 1-8 Data Manipulation Statements - Retrieving Data

| SELECT | Retrieves data from one or more tables in the database. |

When you create a SELECT statement, you can use various clauses to specify different options. (See the entry for the SELECT statement in “ODBC Engine Reference” on page 2-1 for detailed information about each type of clause.) The types of clauses you use in a SELECT statement are as follows:
Data Manipulation Statements

Table 1-9  Data Manipulation Statements - Retrieving Data Options

<table>
<thead>
<tr>
<th>FROM</th>
<th>Specifies the tables or views from which to retrieve data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>Defines search criteria that qualify the data a SELECT statement retrieves.</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>Combines sets of rows according to the criteria you specify and allows you to determine aggregate values for one or more columns in a group.</td>
</tr>
<tr>
<td>HAVING</td>
<td>Allows you to limit a view by specifying criteria that the aggregate values of a group must meet.</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>Determines the order in which Pervasive.SQL returns selected rows.</td>
</tr>
</tbody>
</table>

In addition, you can use the UNION keyword to obtain a single result table from multiple SELECT queries.

Modifying Data

You can add, change, or delete data from tables and views by issuing statements such as the following:

Table 1-10  Data Manipulation Statements - Modifying Data

| INSERT | Adds rows to one or more tables or a view. |
| UPDATE | Changes data in a table or a view. |
| DELETE | Deletes rows from a table or a view. |

When you create a DELETE or UPDATE statement, you can use a WHERE clause to define search criteria that restrict the data upon which the statement acts.

Defining Transactions

To update the data in a database, you can issue SQL statements individually or you can define transactions (logical units of related statements). By defining transactions, you can ensure that either all the statements in a unit of work are executed successfully or none are executed. You can use transactions to group statements to ensure the logical integrity of your database.

Pervasive.SQL supports the ODBC API SQLTransact. See the Microsoft ODBC Programmer’s Reference for more information.
SQL Overview

Creating and Deleting Views

You can create and delete views by constructing statements using the following statements:

Table 1-11 Data Manipulation Statements - Views

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE VIEW</td>
<td>Defines a database view and stores the definition in the dictionary.</td>
</tr>
<tr>
<td>DROP VIEW</td>
<td>Deletes a view from the data dictionary.</td>
</tr>
</tbody>
</table>

Executing Stored Procedures

A stored procedure consists of statements you can precompile and save in the dictionary. To execute stored procedures, construct statements using the following:

Table 1-12 Data Manipulation Statements- Stored Procedures

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Recalls a previously compiled procedure and executes it.</td>
</tr>
</tbody>
</table>

Executing Triggers

A trigger consists of statements you can precompile and save in the dictionary. Triggers are executed automatically by the engine when the specified conditions occur.
Data Control Statements

Data control statements let you define security for your database. When you create a dictionary, no security is defined for it until you explicitly enable security for that dictionary. Pervasive.SQL allows you to construct data control statements to do the following:

- Enable and disable security.
- Create and delete users and groups.
- Grant and revoke rights.

**Note** If your Btrieve data files are secured using Btrieve owner names, the Relational Engine will honor them when performing ODBC operations (for example, Read Only access will be permitted with a type 1 or 3 owner name, no access with a type 0 or 2 owner name) if your database is not secured using Relational security. If your database is secured using Relational security, the Relational Engine will enforce access to the database solely based on the defined database user access rights when performing ODBC operations. These rights must be granted with the owner name specified.

The following sections briefly describe the SQL statements associated with each of these tasks.

**Enabling and Disabling Security**

You can enable or disable security for a database by issuing statements using the following statement:

*Table 1-13 Data Control Statements - Security*

| SET SECURITY | Enables or disables security for the database and sets the Master password. |
You can create or delete users and user groups for the database by constructing statements using the following statements:

<table>
<thead>
<tr>
<th>Table 1-14 Data Control Statements - Groups and Users</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CREATE GROUP</strong></td>
</tr>
<tr>
<td><strong>DROP GROUP</strong></td>
</tr>
<tr>
<td><strong>GRANT LOGIN TO</strong></td>
</tr>
<tr>
<td><strong>REVOKE LOGIN FROM</strong></td>
</tr>
</tbody>
</table>

You can assign or remove rights from users or groups by issuing statements using the following:

<table>
<thead>
<tr>
<th>Table 1-15 Data Control Statements - Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GRANT (access rights)</strong></td>
</tr>
<tr>
<td><strong>GRANT CREATETAB TO</strong></td>
</tr>
<tr>
<td><strong>REVOKE (access rights)</strong></td>
</tr>
<tr>
<td><strong>REVOKE CREATETAB FROM</strong></td>
</tr>
</tbody>
</table>
Database Names

A database name is a name you associate with the location of a dictionary and its data files; it is also the table qualifier. Database names are stored in the database names configuration file (DBNAMES.CFG). If you add a primary key, foreign key, or trigger to a table, the database name is also written to the data file associated with the table. Bound named databases also force the database name to be written to the data file for every table in the database. (For more information about bound databases, refer to the *Pervasive.SQL Programmer’s Guide.*)

Database names must follow these conventions:

- Begin with a letter.
- Cannot contain blanks.
- Cannot be a reserved keyword.
- Must not exceed 20 characters.
- Database names are not case-sensitive.
SQL Overview
This chapter contains information regarding the limits and conformance of the Pervasive.SQL 2000i ODBC interface:

- “Pervasive ODBC Engine Interface Limits” on page 2-2
- “Data Source Name Connection String Keywords” on page 2-3
- “ODBC API Conformance” on page 2-5
- “SQL Grammar Conformance” on page 2-11
- “SQL Grammar Elements” on page 2-16
- “Grammar Element Definitions” on page 2-147
- “Scalar Functions” on page 2-156
- “Other Characteristics” on page 2-168

For detailed information on the ODBC API, SQL grammar, and scalar functions, refer to the Microsoft ODBC Programmer’s Reference.
Pervasive ODBC Engine Interface Limits

The following limits apply to the Pervasive ODBC Engine Interface:

- Number of rows: 2 billion
- Number of SELECT list columns in a query: 1600
- Number of columns in a table: 1536
- Number of columns in a database: the total number of columns, constraints, and indexes in a database must be less than or equal to 65,535. Because a column with the NOT NULL constraint requires an additional field identifier for the constraint itself, the maximum number of NOT NULL columns (assuming no other columns, indexes, or constraints exist) is 32,767.
- Maximum size of a column: 2 GB
- Number of connections: limited by memory
- SQL statement length: 64 KB
- Maximum size of a single term (quoted literal string) in an SQL statement: 14,997, excluding null terminator and quotations (15,000 total)
- Number of statements per connection: limited by memory
- Table name length: 20 characters
- Column name length: 20 characters
- Index name length: 20 characters
- User name length: 30 characters
- Number of columns allowed in a trigger: 300
- Number of arguments in a parameter list for a stored procedure: 300
- Number of joined tables per query: limited by memory
- Length of DBQ entry in the odbc.ini: 20 characters (name of database)
- Maximum of 300 ANDed predicates. For example, this statement uses two ANDed predicates: SELECT * FROM person WHERE First_Name = 'Janis' AND Last_Name = 'Nipart' AND Perm_Street = '1301 K Street NW.'
- A character in a character string literal may be any ANSI character between 1 and 255 decimal. A single quote (') must be represented as two consecutive single quotes (").
Data Source Name Connection String Keywords

A connection string used to connect to a DSN may include any number of driver-defined keywords. Using these keywords, the driver has enough information to connect to the data source. The driver (for example, the Pervasive ODBC Engine Interface or Pervasive ODBC Client Interface) defines which keywords are required to connect to the data source.

Connection strings serve the same purpose in Pervasive.SQL 2000i as they did in previous versions. They are used to identify which data source to connect to. The difference now lies in the driver-defined keywords listed in the connection string. Pervasive.SQL 7 used a different set of keywords to identify a data source.
Listed below are the keywords used in Pervasive.SQL 2000i connection strings.

**Table 2-1  Valid Connection Strings for Client DSNs**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN</td>
<td>Name of the data source as returned by SQLDataSources or the data sources dialog box of SQLDriverConnect.</td>
</tr>
<tr>
<td>DRIVER</td>
<td>The description of the driver as returned by SQLDrivers function. For a Client DSN, it is &quot;Pervasive ODBC Client Interface&quot;.</td>
</tr>
<tr>
<td>ServerName</td>
<td>The address of the server or host name and the port number where the data resides.</td>
</tr>
<tr>
<td>ServerDSN</td>
<td>The name of an Engine data source referenced by this DSN.</td>
</tr>
<tr>
<td>TransportHint</td>
<td>Specifies the transport protocols to check for the ServerName. The list of transport protocols is specified in the order in which they should be searched.</td>
</tr>
<tr>
<td>PWD</td>
<td>The password corresponding to the user ID.</td>
</tr>
<tr>
<td>ArrayFetchOn</td>
<td>Enable array fetching. Array fetching is used to improve performance of data fetching between the client and the server. The default setting is to enable array fetching.</td>
</tr>
<tr>
<td>ArrayBufferSize</td>
<td>Size of the array buffer. Values between 1 and 64KB are acceptable. The default setting is 8KB.</td>
</tr>
<tr>
<td>UID</td>
<td>A user login ID.</td>
</tr>
</tbody>
</table>

**Table 2-2  Valid Connection Strings for Engine DSNs**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN</td>
<td>Name of the data source as returned by SQLDataSources or the data sources dialog box of SQLDriverConnect.</td>
</tr>
<tr>
<td>DRIVER</td>
<td>The description of the driver as returned by SQLDrivers function. For an Engine DSN, it is &quot;Pervasive ODBC Engine Interface&quot;.</td>
</tr>
<tr>
<td>DBQ</td>
<td>A database name.</td>
</tr>
<tr>
<td>UID</td>
<td>A user login ID.</td>
</tr>
<tr>
<td>PWD</td>
<td>The password corresponding to the user ID.</td>
</tr>
</tbody>
</table>
ODBC API Conformance

The Pervasive ODBC Engine Interface fully conforms to the ODBC v2.5 specifications for core grammar API and Level 1 API, and supports most of the Level 2 function calls. The following table lists the ODBC API functions supported by the Pervasive ODBC Engine Interface and the ODBC Conformance level.

<table>
<thead>
<tr>
<th>ODBC Function</th>
<th>ODBC Conformance Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLAllocConnect</td>
<td>Core</td>
</tr>
<tr>
<td>SQLAllocEnv</td>
<td>Core</td>
</tr>
<tr>
<td>SQLAllocStmt</td>
<td>Core</td>
</tr>
<tr>
<td>SQLBindCol</td>
<td>Core</td>
</tr>
<tr>
<td>SQLBindParameter</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLBrowseConnect</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLCancel</td>
<td>Core</td>
</tr>
<tr>
<td>SQLColAttributes</td>
<td>Core</td>
</tr>
<tr>
<td>SQLColumns</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLColumnPrivileges</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLConnect</td>
<td>Core</td>
</tr>
<tr>
<td>SQLDataSources</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLDescribeCol</td>
<td>Core</td>
</tr>
<tr>
<td>SQLDescribeParam</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLDisconnect</td>
<td>Core</td>
</tr>
</tbody>
</table>

Note: The ODBC API functions that are supported in the Pervasive-Oracle Interoperability Environment are described in a separate white paper that is available on the Pervasive home page http://www.pervasive.com
## Interface-supported ODBC API Functions

<table>
<thead>
<tr>
<th>ODBC Function</th>
<th>ODBC Conformance Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLDriverConnect</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLDrivers</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLError</td>
<td>Core</td>
</tr>
<tr>
<td>SQLExecDirect</td>
<td>Core</td>
</tr>
<tr>
<td>SQLExecute</td>
<td>Core</td>
</tr>
<tr>
<td>SQLExtendedFetch</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLFetch</td>
<td>Core</td>
</tr>
<tr>
<td>SQLForeignKeys</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLFreeConnect</td>
<td>Core</td>
</tr>
<tr>
<td>SQLFreeEnv</td>
<td>Core</td>
</tr>
<tr>
<td>SQLFreeStmt</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetConnectOption</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLGetCursorName</td>
<td>Core</td>
</tr>
<tr>
<td>SQLGetData</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLGetFunctions</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLGetInfo</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLGetStmtOption</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLGetTypeInfo</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLMoreResults</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLNativeSql</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQINumResultCols</td>
<td>Core</td>
</tr>
<tr>
<td>SQINumParams</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLParamData</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLPrepare</td>
<td>Core</td>
</tr>
<tr>
<td>SQLPrimaryKeys</td>
<td>Level 2</td>
</tr>
</tbody>
</table>
Exceptions to ODBC API Conformance

The following section contains details on the exceptions to ODBC API conformance as specified in Table 2-3 on page 2-5.

**SQLMoreResults**

The Pervasive ODBC Engine Interface always returns `SQL_NO_DATA_FOUND` for this function. The Pervasive ODBC Engine Interface supports this function, with its return value, due to requirements of Microsoft Access.

---

Table 2-3  Interface-supported ODBC API Functions

<table>
<thead>
<tr>
<th>ODBC Function</th>
<th>ODBC Conformance Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLProcedures</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLProcedureColumns</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLPutData</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLRowCount</td>
<td>Core</td>
</tr>
<tr>
<td>SQLSetConnectOption</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLSetCursorName</td>
<td>Core</td>
</tr>
<tr>
<td>SQLSetPos</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLSetStmtOption</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLSpecialColumns</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLStatistics</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLTables</td>
<td>Level 1</td>
</tr>
<tr>
<td>SQLTablePrivileges</td>
<td>Level 2</td>
</tr>
<tr>
<td>SQLTransact</td>
<td>Core</td>
</tr>
</tbody>
</table>
The following table lists the options the Pervasive ODBC Engine Interface supports for SQLSetStmtOption and SQLGetStmtOption:

<table>
<thead>
<tr>
<th>fOption (numerical value)</th>
<th>vParam</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_MAX_ROWS(1)</td>
<td></td>
<td>Supported according to the Microsoft ODBC Programmer's Reference.</td>
</tr>
<tr>
<td>SQL_NOSCAN(2)</td>
<td></td>
<td>Supported according to the Microsoft ODBC Programmer's Reference.</td>
</tr>
<tr>
<td>SQL_MAX_LENGTH(3)</td>
<td></td>
<td>Supported according to the Microsoft ODBC Programmer's Reference.</td>
</tr>
<tr>
<td>SQL_ASYNC_ENABLE(4)</td>
<td></td>
<td>Supported according to the Microsoft ODBC Programmer's Reference.</td>
</tr>
<tr>
<td>SQL_CURSOR_TYPE(6)</td>
<td></td>
<td>Supported according to the Microsoft ODBC Programmer's Reference.</td>
</tr>
<tr>
<td>SQL_CONCURRENCY(7)</td>
<td></td>
<td>Supported according to the Microsoft ODBC Programmer's Reference.</td>
</tr>
<tr>
<td>SQL_ROWSET_SIZE(9)</td>
<td></td>
<td>Supported according to the Microsoft ODBC Programmer's Reference.</td>
</tr>
</tbody>
</table>
Table 2-4  Options for SQLSetConnectOption and SQLGetConnectOption

<table>
<thead>
<tr>
<th>fOption (numerical value)</th>
<th>vParam</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1151</td>
<td>in the format: Tablename,Password (no space between the Tablename and the Password)</td>
<td>A Pervasive ODBC Engine Interface extension: appends password for table to an internal list in Pervasive ODBC Engine Interface so that the user does not have to be prompted for the password.</td>
</tr>
<tr>
<td>1153</td>
<td>0 (default) turns off table locking; 1 turns on table locking.</td>
<td>A Pervasive ODBC Engine Interface extension: When vParam is set to 1, all tables used by the hStmt are exclusively locked when a select, update, insert, delete, or create index statement is executed on the hStmt. The tables remain locked until the hStmt is dropped (by calling SQLFreeStmt with the SQL_DROP option) or the option is set to DEFLOCK and the hStmt is re-executed. Locked tables can only be used by the locking hStmt; they cannot be used by any other hStmts.</td>
</tr>
</tbody>
</table>

SQLSetConnectOption and SQLGetConnectOption

The following table lists the options the Pervasive ODBC Engine Interface supports for SQLSetConnectOption and SQLGetConnectOption:

Table 2-5  Options for SQLSetConnectOption and SQLGetConnectOption

<table>
<thead>
<tr>
<th>fOption (numerical value)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_ACCESS_MODE(101)</td>
<td>Supported according to the Microsoft ODBC Programmer’s Reference.</td>
</tr>
<tr>
<td>SQL_AUTO(102)</td>
<td>Supported according to the Microsoft ODBC Programmer’s Reference.</td>
</tr>
</tbody>
</table>
**ODBC Engine Reference**

**SQLGetTypeInfo**

SQLGetTypeInfo generates a list of native data type names (type_name) specified by the Pervasive ODBC Engine Interface. For example, SQL_CHAR is mapped to CHARACTER. Use the names which are returned from this function for the data type names for columns in a CREATE TABLE or ALTER TABLE statement or for parameters for procedures or declared variables in procedures and triggers.

**SQLSpecialColumns**

The Pervasive ODBC Engine Interface uses unique indexes as the optimal set of columns that uniquely identifies a row in the table. When a new row is inserted, the Pervasive ODBC Engine Interface does not return the values for autoincrement columns.
SQL Grammar Conformance

The ODBC v2.5 specification provides three levels of SQL grammar conformance: Minimum, Core, and Extended. Each higher level provides more fully-implemented data definition and data manipulation language support. The Pervasive ODBC Engine Interface fully supports the minimum SQL grammar, as well as many core and extended grammar statements. The Pervasive ODBC Engine Interface support for SQL grammar is summarized in the following table.

<table>
<thead>
<tr>
<th>SQL Grammar Statement</th>
<th>Minimum</th>
<th>Core</th>
<th>Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER TABLE</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CREATE GROUP</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CREATE TRIGGER</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>DELETE (positional)</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE (searched)</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DROP GROUP</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>DROP INDEX</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>DROP PROCEDURE</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DROP TRIGGER</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>DROP VIEW</td>
<td></td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-6 SQL Grammar Conformance

<table>
<thead>
<tr>
<th>SQL Grammar Statement</th>
<th>Minimum</th>
<th>Core</th>
<th>Extended</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRANT</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOIN LEFT OUTER (Select)</td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>REVOKE</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECT (with into)</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- approximate-numeric-literal</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- between-predicate</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- correlation-name</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- date arithmetic</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- date-literal</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- exact-numeric-literal</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- extended predicates</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- in-predicate</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- set-function</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- time-literal</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- timestamp-literal</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subqueries</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNION</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET SECURITY</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATE (positional)</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATE (searched)</td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Delimited Identifiers in SQL Statements**

Column names and table names can occur as delimited identifiers if they contain non-ODBC standard characters. The delimiter character for delimited identifiers is a double-quote. For example:

```sql
SELECT "last-name" FROM "non-standard-tbl"
```

**Global Variables**

Pervasive.SQL 2000i supports the following global variables:

- `@@IDENTITY`
- `@@ROWCOUNT`

Either variable can be prefaced with two at signs (@@) or an at sign and a colon (@:). For example, `@@IDENTITY` and `@:IDENTITY` are equivalent.

`@@IDENTITY` and `@@ROWCOUNT` are global variables per connection. Each database connection has its own `@@IDENTITY` and `@@ROWCOUNT` values.

**@@IDENTITY**

This variable returns the value of the most recently inserted IDENTITY column value (IDENTITY or SMALLIDENTITY). The value is a signed integer value. The initial value is NULL.

This variable can only refer to a single column. If the target table includes more than one IDENTITY column, the value of this variable refers to the IDENTITY column that is the table’s primary key. If no such column exists, then the value of this variable refers to the first IDENTITY column in the table.

If the most recent insert was to a table without an IDENTITY column, then the value of `@@IDENTITY` is set to NULL.

**Examples**

```sql
SELECT @@IDENTITY
```

Returns NULL if no records have been inserted in the current connection, otherwise returns the IDENTITY column value of the most recently inserted row.

```sql
SELECT * FROM T1 WHERE @:IDENTITY = 12
```

Returns the most recently inserted row if it has an IDENTITY column value of 12. Otherwise, returns no rows.

```sql
INSERT INTO T1(C2) VALUES (@@IDENTITY)
```
Inserts the IDENTITY value of the last row inserted into column C2 of the new row.

\[
\text{UPDATE T1 SET T1.C1 = (SELECT @@IDENTITY) WHERE T1.C1 = @@IDENTITY + 10}
\]

Updates column C1 with the IDENTITY value of the last row inserted, if the value of C1 is 10 greater than the IDENTITY column value of the last row inserted.

\[
\text{UPDATE T1 SET T1.C1 = (SELECT NULL FROM T2 WHERE T2.C1 = @@IDENTITY)}
\]

Updates column C1 with the value NULL if the value of C1 equals the IDENTITY column value of the last row inserted.

The example below creates a stored procedure and calls it. The procedure sets variable V1 equal to the sum of the input value and the IDENTITY column value of the last row updated. The procedure then deletes rows from the table anywhere column C1 equals V1. The procedure then prints a message stating how many rows were deleted.

\[
\text{CREATE PROCEDURE TEST (IN :P1 INTEGER);}
\]

\[
\text{BEGIN}
\text{DECLARE :V1 INTERGER;}
\text{SET :V1 = :P1 + @@IDENTITY;}
\text{DELETE FROM T1 WHERE T1.C1 = :V1;}
\text{IF (@@ROWCOUNT = 0) THEN}
\text{PRINT 'No row deleted';}
\text{ELSE}
\text{PRINT CONVERT(@@ROWCOUNT, SQL_CHAR) + ' rows deleted';}
\text{END IF;}
\text{END;}
\text{CALL TEST (@@IDENTITY)}
\]

\text{@@ROWCOUNT}

This variable returns the number of rows that were affected by the most recent operation in the current connection. The value is an unsigned integer. The initial value is zero.
SQL Grammar Conformance

**Grammar**

Same as the grammar for @@IDENTITY.

**Examples**

```sql
SELECT @@ROWCOUNT
```

Returns zero if no records were affected by the previous operation in the current connection, otherwise returns the number of rows affected by the previous operation.

```sql
CREATE TABLE T1 (C1 INTEGER, C2 INTEGER)
    INSERT INTO T1 (C1, C2) VALUES (100,200)
    INSERT INTO T1(C2) VALUES (100, @@ROWCOUNT)
    SELECT * FROM T1
SELECT @@ROWCOUNT FROM T1
```

Results:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>

The first SELECT generates two rows and shows that the value of @@ROWCOUNT was 1 when it was used to insert a row. The second SELECT returns 2 as the value of @@ROWCOUNT, that is, after the first SELECT returned two rows.

Also see the example for @@IDENTITY.
SQL Grammar Elements

The following pages in this section describe the complete grammar for all supported SQL grammar elements.

*Note* You can use the SQL Data Manager provided with the Pervasive Control Center to test most of the SQL examples. Exceptions are noted in the discussion of the grammar elements. Type the SQL statements directly in the query pane. If you wish to enter more than one SQL statement, separate each statement using the pound sign (#). You can change this delimiter character to a semi-colon by selecting **Tools | Properties** from the menu.
ADD

Remarks  Use the ADD clause within the ALTER TABLE statement to specify one or more column definitions, column constraints, or table constraints to be added.

See Also  “ALTER TABLE” on page 2-19
Remarks
When you specify the ALL keyword before a subquery, Pervasive.SQL 2000i performs the subquery and uses the result to evaluate the condition in the outer query. If all the rows the subquery returns meet the outer query’s condition for a particular row, Pervasive.SQL 2000i includes that row in the final result table of the statement.

Generally, you can use the EXISTS or NOT EXISTS keyword instead of the ALL keyword.

Examples
The following SELECT statement compares the ID column from the Person table to the ID columns in the result table of the subquery:

```sql
SELECT p.ID, p.Last_Name
FROM Person p
WHERE p.ID <> ALL
    SELECT f.ID FROM Faculty f WHERE f.Dept_Name = 'Chemistry');
```

If the ID value from Person does not equal to any of the ID values in the subquery result table, Pervasive.SQL includes the row from Person in the final result table of the statement.

See Also
“GRANT” on page 2-80
“SELECT (with into)” on page 2-117
“SELECT” on page 2-118
“UNION” on page 2-136
**ALTER TABLE**

The ALTER TABLE statement modifies a table definition.

**Syntax**

```
ALTER TABLE  table-name  [  IN  DICTIONARY  ]
   [  USING  'path_name'  ]  [  WITH  REPLACE  ]  alter-option-list
```

table-name ::=user-defined-name

alter-option-list ::=alter-option
   | (alter-option  [,  alter-option] . . . )

alter-option ::= ADD  [  COLUMN  ]  column-definition
   | ADD  table-constraint-definition
   | DROP  [  COLUMN  ]  column-name
   | DROP  CONSTRAINT  constraint-name
   | DROP PRIMARY KEY
   | MODIFY  [  COLUMN  ]  column-definition
   | ALTER  [  COLUMN  ]  column-definition

column-definition ::= column-name  data-type  [  DEFAULT  default-value  ]  [  column-constraint  [  column-constraint  ] . . .  [  CASE  |  COLLATE  ]  collation-name  ]

column-name ::= user-defined-name

data-type ::= data-type-name  [  (precision  [  ,  scale  ] ) ]

precision ::= integer

scale ::= integer

default-value ::= literal

literal ::= 'string'
   | number
   | { d 'date-literal' }
   | { t 'time-literal' }
   | { ts 'timestamp-literal' }

column-constraint ::= [  CONSTRAINT  constraint-name  ]  col-constraint

col-constraint ::= [  [  constraint-name  ]  col-constraint

constraint-name ::= user-defined-name
col-constraint ::= NOT NULL
    | UNIQUE
    | PRIMARY KEY
    | REFERENCES table-name [ ( column-name ) ] [ referential-actions ]

referential-actions ::= referential-update-action [ referential-delete-action ]
    | referential-delete-action [ referential-update-action ]

referential-update-action ::= ON UPDATE RESTRICT

referential-delete-action ::= ON DELETE CASCADE
    | ON DELETE RESTRICT

collation-name ::= 'string' | user-defined-name

table-constraint-definition ::= [ CONSTRAINT constraint-name ] table-constraint

table-constraint ::= UNIQUE ( column-name [ , column-name ] ... )
    | PRIMARY KEY ( column-name [ , column-name ] ... )
    | FOREIGN KEY ( column-name [ , column-name ] )
    | REFERENCES table-name
    [ ( column-name [ , column-name ] ... ) ]
    [ referential-actions ]

Remarks
Refer to CREATE TABLE for information pertaining to primary and foreign keys and referential integrity.

IN DICTIONARY

The purpose of using this keyword is to notify the SQL Relational Database Engine (SRDE) that you wish to make modifications to the DDFs, while leaving the underlying physical data unchanged. IN DICTIONARY is a very powerful and advanced feature. It should only be used by system administrators or when absolutely necessary. Normally, the SRDE keeps DDFs and data files totally synchronized, but this feature allows users the flexibility to force table dictionary definitions to match an existing data file. This can be useful when you want to create a definition in the dictionary to match an existing data file, or when you want to use a USING clause to change the data file path name for a table.

You cannot use this keyword on a bound database.
IN DICTIONARY is allowed on CREATE and DROP TABLE, in addition to ALTER TABLE. IN DICTIONARY affects dictionary entries only, no matter what CREATE/ALTER options are specified. Since Pervasive.SQL 2000i allows multiple options (any combination of ADD, DROP, ADD CONSTRAINT, and so on), IN DICTIONARY is honored under all circumstances to guarantee only the DDFs are affected by the schema changes.

Tables that exist in the DDFs only (the data file does not exist) are called detached entries. These tables are inaccessible via queries or other operations that attempt to open the physical underlying file. For this reason, IN DICTIONARY was added to DROP TABLE, because it is now possible to create detached entries using CREATE TABLE. Note that errors such as "Table not found" are generated by attempts to access these detached entries. One can verify whether a table really exists by using SQLTables or directly querying the Xf$Name column of X$File:

```
SELECT * FROM X$File WHERE Xf$Name = 'table_name'
```

It is possible for a detached table to cause confusion, so the IN DICTIONARY feature must be used with extreme care. It is crucial that it should be used to force table definitions to match physical files, not to detach them. Consider the following examples, assuming that the file test123.btr does not exist. (USING is explained below, in the next subtopic.)

```
CREATE TABLE t1 USING 't1.btr' (c1 INT)
ALTER TABLE t1 IN DICTIONARY USING 'test123.btr'
```

Or, combining both statements:

```
CREATE TABLE t1 IN DICTIONARY USING 'test123.btr' (c1 INT)
```

If you then attempt to SELECT from t1, you receive an error that the table was not found. Confusion can arise, because you just created the table—how can it not be found? Likewise, if you attempt to DROP the table without specifying IN DICTIONARY, you receive the same error. These errors are generated because there is no data file associated with the table.
USING

The USING keyword allows you to associate a CREATE TABLE or ALTER TABLE action with a particular data file.

Because Pervasive.SQL requires a Named Database to connect, the path_name provided must always be a simple file name or relative path and file name. Paths are always relative to the first Data Path specified for the Named Database to which you are connected.

The path/file name passed is partially validated when SQLPrepare is called. The following rules must be followed when specifying the path name:

- The text must be enclosed in single quotes, as shown in the grammar definition.
- Text must be 1 to 64 characters in length, such that the entry as specified fits in Xf$Loc in X$File. The entry is stored in Xf$Loc exactly as typed (trailing spaces are truncated and ignored).
- The path must be a simple, relative path. Paths that reference a server or volume are not allowed. For NetWare, a volume-based path (such as SYS:/path/testfile.btr) is not considered a simple, relative path.
- Relative paths containing a period ('.' - current directory), double-period ('..' - parent directory), slash '\', or any combination of the three are allowed. The path must contain a file name representing the SQL table name (path_name cannot end in a slash '\' or a directory name). All file names, including those specified with relative paths, are relative to the first Data Path as defined in the Named Database configuration.
- Root-based relative paths are also allowed. For example, assuming that the first data path is D:\PVSW\DEMODATA, the SRDE interprets the path name in the following statement as D:\TEMP\TEST123.BTR.

```
CREATE TABLE t1 USING '"temp\test123.btr' (c1 int)
```

- Slash ('\') characters in relative paths may be specified either UNIX style ('/') or in the customary backslash notation ('\'), depending on your preference. You may use a mixture of the two types, if desired. This is a convenience feature since you may know the directory structure scheme, but not necessarily know (or care) what type of server you are connected to. The path is stored in X$File exactly as typed. The SRDE engine converts the slash characters to the appropriate platform type when utilizing
the path to open the file. Also, since data files share binary compatibility between all supported platforms, this means that as long as the directory structure is the same between platforms (and path-based file names are specified as relative paths), the database files and DDFs can be moved from one platform to another with no modifications. This makes for a much simpler cross-platform deployment with a standardized database schema.

- If specifying a relative path, the directory structure in the USING clause must first exist. The SRDE does not create directories to satisfy the path specified in the USING clause.

Include a USING clause to specify the physical location and name of an existing data file to associate with an existing table. A USING clause also allows you to create a new data file at a particular location using an existing dictionary definition. (The string supplied in the USING clause is stored in the Xf$Loc column of the dictionary file X$File.) The original data file must be available when you create the new file since some of the file information must be obtained from the original.

In the DEMODATA sample database, the Person table is associated with the file PERSON.MKD. If you create a new file named PERSON2.MKD, the statement in the following example changes the dictionary definition of the Person table so that the table is associated with the new file.

```sql
ALTER TABLE Person IN DICTIONARY USING 'person2.mkd'
```

You must use either a simple file name or a relative path in the USING clause. If you specify a relative path, Pervasive.SQL interprets it relative to the first data file path associated with the database name.

The USING clause can be specified in addition to any other standard ALTER TABLE option. This means columns can be manipulated in the same statement that specifies the USING path.

If you specify a data file name that differs from the data file name currently used to store the table data, the SRDE creates the new file and copies all of the data from the existing file into the new file. For example, suppose person.mkd is the current data file that holds the data for table Person. You then alter table Person using data file person2.mkd, as shown in the statement above. The contents of person.mkd are copied into person2.mkd. Person2.mkd then becomes the data file associated with table Person and database.
operations affect person2.mkd. Person.mkd is not deleted, but it is not associated with the database any more.

The reason for copying the data is because Pervasive.SQL allows all other ALTER TABLE options at the same time as USING. The new data file created needs to be fully populated with the existing table's data. The file structure is not simply copied, but instead the entire contents are moved over, similar to a Btrieve BUTIL -CREATE and BUTIL -COPY. This can be helpful for rebuilding an SQL table, or compressing a file that once contained a large number of records but now contains only a few.

Note ALTER TABLE USING copies the contents of the existing data file into the newly specified data file, leaving the old data file intact but unlinked.

WITH REPLACE

Whenever WITH REPLACE is specified with the USING keyword, Pervasive.SQL automatically overwrites any existing file name with the specified file name. The existing file is always overwritten as long as the operating system allows it.

WITH REPLACE affects only the data file, it never affects the DDFs.

The following rules apply when using WITH REPLACE:

- WITH REPLACE can only be used with USING.
- When used with IN DICTIONARY, WITH REPLACE is ignored because IN DICTIONARY specifies that only the DDFs are affected.

Note No data is lost or discarded if WITH REPLACE is used with ALTER TABLE. The newly created data file, even if overwriting an existing file, still contains all data from the previous file. You cannot lose data by issuing an ALTER TABLE command.

Include WITH REPLACE in a USING clause to instruct Pervasive.SQL to replace an existing file (the file must reside at the location you specified in the USING clause). If you include WITH REPLACE, Pervasive.SQL creates a new file and copies all the data from the existing file into it. If you do not include WITH REPLACE and a file exists at the specified location, Pervasive.SQL returns a
status code and does not create the new file. The status code is SRDE error -4940, Btrieve error 59.

MODIFY COLUMN and ALTER COLUMN

The ability to modify the nullability or data type of a column is subject to the following restrictions:

- The target column cannot have a PRIMARY/FOREIGN KEY constraint defined on it.
- If converting the old type to the new type causes an overflow (arithmetic or size), the ALTER TABLE operation is aborted.
- If a nullable column contains NULL values, the column cannot be changed to a non-nullable column.

If you must change the data type of a key column, you can do so by dropping the key, changing the data type, and re-adding the key. Keep in mind that you must ensure that all associated key columns remain synchronized. For example, if you have a primary key in table T1 that is referenced by foreign keys in tables T2 and T3, you must first drop the foreign keys. Then you can drop the primary key. Then you need to change all three columns to the same data type. Finally, you must re-add the primary key and then the foreign keys.

The ANSI standard includes the ALTER keyword. Pervasive.SQL allows both keywords (ALTER and MODIFY) in the ALTER TABLE statement:

```
ALTER TABLE T1 MODIFY C1 INTEGER
ALTER TABLE T1 ALTER C1 INTEGER
ALTER TABLE T1 MODIFY COLUMN C1 INTEGER
ALTER TABLE T1 ALTER COLUMN C1 INTEGER
```

Pervasive.SQL allows altering a column to a smaller length if the actual data does not overflow the new, smaller length of the column. This behavior is similar to that of Microsoft SQL Server.

You can add, drop, or modify multiple columns on a single ALTER TABLE statement. Although it simplifies operations, this behavior is not considered ANSI-compliant. The following is a sample multi-column ALTER statement.

```
ALTER TABLE T1 (ALTER C2 INT, ADD D1 CHAR(20), DROP C4, ALTER C5 LONGVARCHAR, ADD D2 LONGVARCHAR NOT NULL)
```

You can convert all legacy data types (Pervasive.SQL v7 or earlier) to data types that are natively supported by Pervasive.SQL 2000i. But
the new data types cannot be converted backwards to legacy data types.

To add a LONGVARCHAR/LONGVARBINARY column to a legacy table that contains a NOTE/IVAR column, the NOTE/IVAR column first has to be converted to a LONGVARCHAR or LONGVARBINARY column. After converting the NOTE/IVAR column to LONGVARCHAR/LONGVARBINARY, you can add more LONGVARCHAR/LONGVARBINARY columns to the table. Note that the legacy engine does not work with this legacy table because the legacy engine can work with only one variable length column per table.

**Examples**

The following statement adds the Emergency_Phone column to the Person table

```sql
ALTER TABLE person ADD Emergency_Phone NUMERIC(10,0)
```

The following statement adds two integer columns col1 and col2 to the Class table.

```sql
ALTER TABLE class (ADD col1 INT, ADD col2 INT)
```

To drop a column from a table definition, specify the name of the column in a **DROP** clause. The following statement drops the emergency phone column from the Person table.

```sql
ALTER TABLE person DROP Emergency_Phone
```

The following statement drops col1 and col2 from the Class table.

```sql
ALTER TABLE class (DROP col1, DROP col2)
```

The following statement drops the constraint c1 in the Faculty table.

```sql
ALTER TABLE Faculty (DROP CONSTRAINT c1)
```

This example adds an integer column col3 and drops column col2 to the Class table

```sql
ALTER TABLE class (ADD col3 INT, DROP col2)
```

The following example creates a primary key named c1 on the ID field in the Faculty table. Note that you cannot create a primary key
on a Nullable column. Doing so generates the error, "Nullable columns are not allowed in primary keys".

```
ALTER TABLE Faculty (ADD CONSTRAINT c1 PRIMARY KEY (ID))
```

The following example creates a primary key PK_ID in the Faculty table.

```
ALTER TABLE Faculty (ADD PRIMARY KEY (ID))
```

The following example adds the constraint UNIQUE to the columns coll and col2.

```
ALTER TABLE Class (ADD UNIQUE (coll, col2))
```

The following example drops the primary key in the Faculty table. Because a table can have only one primary key, you cannot add a primary key to a table that already has a primary key defined. To change the primary key of a table, delete the existing key then add the new primary key.

```
ALTER TABLE Faculty (DROP PRIMARY KEY)
```

Before you can drop a primary key from a parent table, you must drop any corresponding foreign keys from dependent tables.

```
ALTER TABLE Class ADD CONSTRAINT Teacher FOREIGN KEY (Faculty_ID) REFERENCES Faculty (ID) ON DELETE RESTRICT
```

In this example, the restrict rule for deletions prevents someone from removing a faculty member from the database without first either changing or deleting all of that faculty’s classes.

If you add a foreign key to a table that already contains data, use the Referential Integrity (RI) test to find any data that does not conform to the new referential constraint. The RI test is run from the Check Database wizard in the Pervasive Control Center (PCC).

The following statement shows how to drop the foreign key added in this example. Pervasive.SQL drops the foreign key from the dependent table and eliminates the referential constraints between the dependent table and the parent table.
ALTER TABLE Class DROP CONSTRAINT Teacher

---

The following example adds a foreign key to the Class table without using the CONSTRAINT clause.

ALTER TABLE Class ADD FOREIGN KEY (Faculty_ID)
REFERENCES Faculty (ID) ON DELETE RESTRICT

This creates foreign key FK_Faculty_ID. To drop the foreign key, specify the CONSTRAINT keyword:

ALTER TABLE Class DROP CONSTRAINT FK_Faculty_ID

---

The following example illustrates multiple adding and dropping of constraints and columns in a table. This statement drops column salary, adds a column col1 of type integer, and drops constraint c1 in the Faculty table.

ALTER TABLE Faculty DROP salary, ADD col1 INT, DROP CONSTRAINT c1

---

The following examples illustrate altering the data type of multiple columns.

ALTER TABLE T1 (ALTER C2 INT, ADD D1 CHAR(20), DROP C4, ALTER C5 LONGVARCHAR, ADD D2 LONGVARCHAR NOT NULL)

ALTER TABLE T2 (ALTER C1 CHAR(50), DROP CONSTRAINT MY_KEY, DROP PRIMARY KEY, ADD MYCOLUMN INT)

The following examples illustrate how the column default and alternate collating sequence can be set or dropped with the ALTER or MODIFY column options.

CREATE TABLE T1 (c1 INT DEFAULT 10, c2 CHAR(10))

ALTER TABLE T1 ALTER c1 INT DEFAULT 20
— resets column c1 default to 20

ALTER TABLE T1 ALTER c1 INT
— drops column c1 default

ALTER TABLE T1 ALTER c2 CHAR(10)
COLLATE 'c:\psw\samples\upper.alt'
— sets alternate collating sequence on column c2

ALTER TABLE T1 ALTER c2 CHAR(10)
— drops alternate collating sequence on column c2
Upper.alt treats upper and lower case letters the same for sorting. For example, if a database has values abc, ABC, DEF, and Def, inserted in that order, the sorting with upper.alt returns as abc, ABC, DEF, and Def. (The values abc and ABC, and the values DEF and Def are considered duplicates and are returned in the order in which they were inserted.) Normal ASCII sorting sequences upper case letters before lower case, such that the sorting would return as ABC, DEF, Def, abc.
ANY

Remarks
The ANY keyword works similarly to the ALL keyword except that Pervasive.SQL 2000i includes the compared row in the result table if the condition is true for any row in the subquery result table.

Examples
The following statement compares the ID column from Person to the ID columns in the result table of the subquery. If the ID value from Person is equal to any of the ID values in the subquery result table, Pervasive.SQL includes the row from Person in the result table of the SELECT statement.

```sql
SELECT p.ID, p.Last_Name
FROM Person p
WHERE p.ID = ANY
(SELECT f.ID FROM Faculty f WHERE f.Dept_Name = 'Chemistry');
```

See Also
“SELECT” on page 2-118
AS

Remarks
Include an AS clause to assign a name to a select term or to a table name. You can use this name elsewhere in the statement to reference the select term. When you use the AS clause on a non-aggregate column, you can reference the name in WHERE, ORDER BY, GROUP BY, and HAVING clauses. When you use the AS clause on an aggregate column, you can reference the name only in an ORDER BY clause.

The name you define must be unique in the SELECT list.

Examples
The AS clause in the following statement instructs Pervasive.SQL to assign the name Total to the select term SUM (Amount_Paid) and order the results by the total for each student:

```sql
SELECT Student_ID, SUM(Amount_Paid) AS Total
FROM Billing
GROUP BY Student_ID
ORDER BY Total
```

When you use the AS clause on a table name in a FROM clause, you can reference the name in a WHERE, ORDER BY, GROUP BY, and HAVING clause.

```sql
SELECT DISTINCT c.Name, p.First_Name, c.Faculty_Id
FROM Person AS p, class AS c
WHERE p.Id = c.Faculty_Id
ORDER BY c.Faculty_Id
```

You can rewrite this query without using the AS clause in the FROM clause as follows.

```sql
SELECT DISTINCT c.Name, p.First_Name, c.Faculty_Id
FROM Person p, class c
WHERE p.Id = c.Faculty_Id
ORDER BY c.Faculty_Id
```

See Also
“SELECT” on page 2-118
BEGIN [ATOMIC]

Remarks
It is often convenient to group individual statements so that they can be treated as a single unit. The BEGIN and END statements are used in compound statements to group the statements into a unit. You can use a compound statement only in the body of a stored procedure or in a trigger declaration.

ATOMIC specifies that the set of statements within the unit either all succeed or all fail. If one condition within the BEGIN ATOMIC...END unit is not met, no records are affected. If the condition should affect more than one row, all rows (or none) are affected. For any record to be affected, all the conditions within a BEGIN ATOMIC...END unit must return true.

Examples
In the following example, two UPDATEs are grouped as an ATOMIC unit. The Perm_State column in the Person table is updated only if all of the other conditions are true. That is, a record for Bill Andrew exists with 'TX' as the Perm_State, and a record for Yvette Lopez exists with 'OR' as the Perm_State. If any of these conditions are not true, neither record is updated. Assume the BEGIN...END unit is within a procedure.

BEGIN ATOMIC
    UPDATE Person SET Perm_State = 'MA' WHERE Perm_State = 'TX' AND First_Name = 'Bill' AND Last_Name = 'Andrew';

    UPDATE Person SET Perm_State = 'WA' WHERE Perm_State = 'OR' AND First_Name = 'Yvette' AND Last_Name = 'Lopez';

END;

See Also
“CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
CALL

Remarks  Use the CALL statement to invoke a previously created stored procedure.

Examples  The following example calls a procedure without parameters:
          CALL NoParms() or CALL NoParms

          The following examples call a procedure with parameters:
          CALLParms(vParm1, vParm2)
          CALL CheckMax(N.Class_ID)

See Also  “CREATE PROCEDURE” on page 2-42
          “CREATE TRIGGER” on page 2-59
CASCAD

Remarks
If you specify CASCADE when creating a Foreign Key, Pervasive.SQL uses the DELETE CASCADE rule. When a user deletes a row in the parent table, Pervasive.SQL 2000i deletes the corresponding rows in the dependent table.

See Also
“ALTER TABLE” on page 2-19
“CREATE TABLE” on page 2-50
**CASE**

**Remarks**

The CASE keyword causes Pervasive.SQL to ignore case when evaluating restriction clauses involving a string column.

For example, suppose if you have a column called Name that is defined with the CASE attribute. If you insert two rows with Name = ‘Smith’ and Name = ‘SMITH,’ a query with a restriction specifying Name = ‘smith’ correctly returns both rows.

**Examples**

The following example shows how you add a column to the Student table with the CASE keyword.

```
ALTER TABLE Student ADD Name char(64) CASE
```

The following example shows how to modify a column with the CASE keyword.

```
ALTER TABLE Student MODIFY Name char(64) CASE
```

**See Also**

“ALTER TABLE” on page 2-19

“SELECT” on page 2-118
CLOSE

Remarks
The CLOSE statement closes an open SQL cursor.
The cursor that the cursor name specifies must be open.
This statement is allowed only inside of a stored procedure or a trigger, since cursors and variables are only allowed inside of stored procedures and triggers.

Examples
The following example closes the cursor BTUCursor.

CLOSE BTUCursor;

See Also
“OPEN” on page 2-104
“CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
The COMMIT statement signals the end of a logical transaction and converts temporary data into permanent data. The logical transaction begins with START TRANSACTION. COMMIT must always be paired with START TRANSACTION. START TRANSACTION must always be paired with a COMMIT or a ROLLBACK.

**Syntax**

```
COMMIT [WORK]
```

**Remarks**

COMMIT (and START TRANSACTION) is supported only within stored procedures. You cannot use COMMIT or START TRANSACTION within the SQL Data Manager. (The SQL Data Manager sets AUTOCOMMIT to “on.”)

Any committed statements within a stored procedure are controlled by the outermost transaction of the calling ODBC application. This means that, depending on the AUTOCOMMIT mode specified on SQLSetConnectOption, calling the stored procedure externally from an ODBC application performs one of two actions. It either commits automatically (AUTOCOMMIT on, the default) or waits for you to call SQLTransact with SQL_COMMIT or SQL_ROLLBACK (when AUTOCOMMIT is set to off).

You may call multiple START TRANSACTION statements to start the nested transactions, but the outermost COMMIT controls whether any nested committed blocks are committed or rolled back. For example, if transactions are nested five levels, then five COMMIT statements are needed to commit all of the transactions. COMMIT does not release any lock until the outermost transaction is committed.

COMMIT and COMMIT WORK perform the same functionality.

**Examples**

The following example, within a stored procedure, begins a transaction which updates the Amount_Owed column in the Billing table. This work is committed; another transaction updates the Amount_Paid column and sets it to zero. The final COMMIT WORK statement ends the second transaction.
Statements are delimited with a semi-colon inside stored procedures and triggers.

START TRANSACTION;
UPDATE Billing B
    SET Amount_Owed = Amount_Owed - Amount_Paid
    WHERE Student_ID IN
        (SELECT DISTINCT E.Student_ID
         FROM Enrolls E, Billing B
         WHERE E.Student_ID = B.Student_ID);

COMMIT WORK;

START TRANSACTION;
UPDATE Billing B
    SET Amount_Paid = 0
    WHERE Student_ID IN
        (SELECT DISTINCT E.Student_ID
         FROM Enrolls E, Billing B
         WHERE E.Student_ID = B.Student_ID);

COMMIT WORK;

See Also
“CREATE PROCEDURE” on page 2-42
“ROLLBACK” on page 2-114
“START TRANSACTION” on page 2-134
CREATE GROUP

The CREATE GROUP statement creates one or more security groups.

Syntax

```
CREATE GROUP group-name [ , group-name ] ...
```


Examples

The following example creates a group named pervasive.

```
CREATE GROUP pervasive
```

The next example uses a list to create several groups at once.

```
CREATE GROUP pervasive_dev, pervasive_marketing
```

See Also

“DROP GROUP” on page 2-69

“GRANT” on page 2-80

“REVOKE” on page 2-112

“SET SECURITY” on page 2-129
CREATE INDEX

Use the CREATE INDEX statement to create a named index for a specified table.

Syntax

```
CREATE [ UNIQUE | NOT MODIFIABLE ] INDEX index-name
ON table-name [ index-definition ]

index-definition ::= ( index-segment-definition [ , index-segment-definition ] ... )

index-segment-definition ::= column-name [ ASC | DESC ]

index-name ::= user-defined-name
```

Remarks

The maximum column size for varchar columns is 254 bytes if the column does not allow Null values and 253 bytes if the column is nullable.

The maximum column size for char columns is 255 bytes if the column does not allow Null values and 254 bytes if the column is nullable.

The maximum Btrieve key size is 255. When a column is nullable and indexed a segmented key is created with 1 byte for the null indicator and a maximum 254 bytes from the column indexed. Varchar columns differ from char columns in that either the length byte (Btrieve lstring) or a zero terminating byte (Btrieve zstring) are reserved, reducing the effective storage by 1 byte.

**Pervasive.SQL 2000i nullable columns:** For data files with 4096 byte page size, you are limited to 119 index segments per file. Because each indexed nullable column with true null support requires an index consisting of 2 segments, you cannot have more than 59 indexed nullable columns in a table (or indexed nullable true null fields in a Btrieve file). This limit is smaller for smaller page sizes. Any file created with Pervasive.SQL 2000i, with file create mode set to 7.x, and TRUENULLCREATE set to the default value of On, has true null support. Files created using an earlier file format, or with Pervasive.SQL 7, or with TRUENULLCREATE set to Off, do not have true null support and do not have this limitation.

A UNIQUE segment key guarantees that the combination of the segments for a particular row are unique in the file. It does not guarantee or require that each individual segment is unique.
Note All data types can be indexed except for the following:
BIT
LONGVARBINARY
LONGVARCHAR
BLOB
CLOB

Examples
The following example creates an index named Dept based on
Dept_name in the Faculty table.
 CREATE INDEX Dept on Faculty(Dept_Name)

The following example creates a non-modifiable segmented index in
the Person table.
 CREATE NOT MODIFIABLE INDEX X_Person on Person(ID,
Last_Name)

See Also
“DROP INDEX” on page 2-70
CREATE PROCEDURE

The CREATE PROCEDURE statement creates a new stored procedure. Stored procedures are SQL statements that are pre-defined and saved in the database dictionary.

Syntax

```
CREATE PROCEDURE procedure-name
    ( [ parameter [ , parameter ] ... ] )
    RETURNS ( result [ , result ] ... )
    [ WITH DEFAULT HANDLER ]
    as-or-semicolon
    proc-stmt
```

```
procedure-name ::= user-defined-name
```

```
parameter ::= parameter-type-name data-type [ DEFAULT proc-expr ] = proc-expr

| SQLSTATE

parameter-type-name ::= parameter-name

| parameter-name parameter-type-name

| parameter-name parameter-type

```

```
parameter-type ::= IN | OUT | INOUT | IN_OUT
```

```
parameter-name ::= [ : ] user-defined-name
```

```
proc-expr ::= same as normal expression but does not allow IF expression, or ODBC-style scalar functions
```

```
result ::= user-defined-name data-type
```

```
as-or-semicolon ::= AS | ;
```

```
proc-stmt ::= [ label-name : ] BEGIN [ATOMIC] [ proc-stmt [ ; proc-stmt ] ... ] END [ label-name ]

| CALL procedure-name ( proc-expr [ , proc-expr ] ... )

| CLOSE cursor-name

| DECLARE cursor-name CURSOR FOR select-statement [ FOR UPDATE | FOR READ ONLY ]

| DECLARE variable-name data-type [ DEFAULT proc-expr ] = proc-expr

| DELETE WHERE CURRENT OF cursor-name

| delete-statement
```
SQL Grammar Elements

| `FETCH` [ fetch-orientation [ FROM ] ] cursor-name [ INTO variable-name [ , variable-name ] ] |
| `IF` proc-search-condition `THEN` proc-stmt [ ; proc-stmt ]... [ `ELSE` proc-stmt [ ; proc-stmt ]... ] END IF |
| insert-statement |
| `LEAVE` label-name |
| [ label-name : ] LOOP proc-stmt [ ; proc-stmt ]... END LOOP [ label-name ] |
| `OPEN` cursor-name |
| `PRINT` proc-expr [ , 'string' ] |
| — applies only to Windows-based platforms |
| `RETURN` [ proc-expr ] |
| transaction-statement |
| `SET` variable-name = proc-expr |
| `SIGNAL` [ `ABORT` ] sqlstate-value |
| `START TRANSACTION` |
| update-statement |
| `UPDATE SET` column-name = proc-expr [ , column-name = proc-expr ]... WHERE CURRENT OF cursor-name |
| [ label-name : ] `WHILE` proc-search-condition `DO` [ proc-stmt [ ; proc-stmt ] ]... END `WHILE` [ label-name ] |

transaction-statement ::= commit-statement

commit-statement ::= see COMMIT statement

rollback-statement ::= see ROLLBACK statement

release-statement ::= see RELEASE SAVEPOINT statement

label-name ::= user-defined-name

cursor-name ::= user-defined-name

variable-name ::= user-defined-name
proc-search-condition ::= same as normal search-condition, but does not allow any expression that includes a subquery.

fetch-orientation ::= | NEXT

sqlstate-value ::= 'string'

Remarks

To execute stored procedures, use the CALL statement.

Note that, in a procedure, the name of a variable and the name of a parameter must begin with a colon (:), both in the definition and use of the variable or parameter.

Statements are delimited with a semi-colon inside stored procedures and triggers.

The WITH DEFAULT HANDLER clause, when present, causes the procedure to continue execution when an error occurs. The default behavior (without this clause) is to abort the procedure with SQLSTATE set to the error state generated by the statement.

The use of a StmtLabel at the beginning (and optionally at the end) of an IF statement is an extension to ANSI SQL 3.

The PRINT statement applies only to Windows-based platforms. It is ignored on other operating system platforms.

Examples

The following example creates stored procedure Enrollstudent, which inserts a record into the Enrolls table, given the Student ID and the Class ID.

```
CREATE PROCEDURE Enrollstudent(IN :Stud_id INTEGER, IN :Class_Id INTEGER, IN :GPA REAL);
BEGIN
    INSERT INTO Enrolls VALUES(:Stud_id, :Class_id, :GPA);
END;
```

```
CALL Enrollstudent(1023456781, 146, 3.2)
SELECT * FROM Enrolls WHERE Student_id = 1023456781
```

The CALL and SELECT statements, respectively, call the procedure by passing arguments, then display the row that was added.
The following procedure reads the Class table, using the classId parameter passed in by the caller and validates that the course enrollment is not already at its limit before updating the Enrolls table.

```sql
CREATE PROCEDURE Checkmax(in :classid integer);
BEGIN
    DECLARE :numenrolled integer;
    DECLARE :maxenrolled integer;
    SELECT COUNT(*) INTO :numenrolled FROM Enrolls WHERE class_ID = :classid;
    SELECT Max_size INTO :maxenrolled FROM Class WHERE id = :classid;
    IF (:numenrolled > :maxenrolled) THEN
        PRINT 'Enrollment Failed. Number of students enrolled reached maximum allowed for this class';
    ELSE
        PRINT 'Enrollment Successful.';
    END IF;
END;
CALL Checkmax(101)
```

Note that COUNT(expression) counts all non-NULL values for an expression across a predicate. COUNT(*) counts all values, including NULL values.

The following is an example of using the OUT parameter when creating stored procedures. Calling this procedure returns the number of students into the variable :outval that satisfies the WHERE clause.

```sql
CREATE PROCEDURE PROCOUT (out :outval INTEGER) AS BEGIN
    SELECT COUNT(*) INTO :outval FROM Enrolls WHERE Class_Id = 101;
END;
```
The following is an example of using the INOUT parameter when creating stored procedures. Calling this procedure requires an INPUT parameter :IOVAL and returns the value of the output in the variable :IOVAL. The procedure sets the value of this variable based on the input and the IF condition.

```sql
CREATE PROCEDURE ProcIODate (INOUT :ioval DATE)
AS BEGIN
    IF :ioval = '1982-03-03'
    THEN
        SET :ioval = '1982-05-05';
    ELSE
        SET :ioval = '1982-03-03';
    END IF;
END;
```

The following example illustrates using the RETURNS clause in a procedure. This sample returns all of the data from the Class table where the Start Date is equal to the date passed in on the CALL statement.

```sql
CREATE PROCEDURE DateReturnProc(IN :pdate DATE)
RETURNS(
    ID INTEGER,
    Name CHAR(7),
    Section CHAR(3),
    Max_Size USMALLINT,
    Start_Date DATE,
    Start_Time TIME,
    Finish_Time TIME,
    Building_Name CHAR(25),
    Room_Number UINTEGER,
    Faculty_ID UBIGINT
);
BEGIN
    SELECT * FROM class WHERE Start_Date = :pdate;
END;
CALL DateReturnProc('2001-06-05')
The following example shows the use of the WHERE CURRENT OF clause, which applies to positioned deletes.

```sql
CREATE PROCEDURE MyProc(IN :CourseName CHAR(7)) AS
BEGIN
    DECLARE c1 CURSOR FOR SELECT name FROM course
        WHERE name = :CourseName FOR UPDATE;
    OPEN c1;
    FETCH NEXT FROM c1 INTO :CourseName;
    DELETE WHERE CURRENT OF c1;
    CLOSE c1;
END;
CALL MyProc('HIS 305')
```

(Note that if you use a SELECT inside of a WHERE clause of a DELETE, it is a searched DELETE not a positioned DELETE.)

The following is an example of using ATOMIC, which groups a set of statements so that either all succeed or all fail. ATOMIC can be used only within the body of a stored procedure or trigger.

The first procedure does not specify ATOMIC, the second does.

```sql
CREATE TABLE t1 (c1 INTEGER)
CREATE UNIQUE INDEX t1i1 ON t1 (c1)
CREATE PROCEDURE p1 ();
BEGIN
    INSERT INTO t1 VALUES (1);
    INSERT INTO t1 VALUES (1);
END;
CREATE PROCEDURE p2 ();
BEGIN ATOMIC
    INSERT INTO t1 VALUES (2);
    INSERT INTO t1 VALUES (2);
END;
CALL p1()
CALL p2()
SELECT * FROM t1
```
Both procedures return an error because they attempt to insert duplicate values into a unique index.

The result is that t1 contains only one record because the first INSERT statement in procedure p1 succeeds even though the second fails. Likewise, the first INSERT statement in procedure p2 succeeds but the second fails. However, since ATOMIC is in procedure p2, all of the work done inside procedure p2 is rolled back when the error is encountered.

Using Stored Procedures

As an example, CALL foo(a, b, c) executes the stored procedure “foo” with parameters a, b, and c. Any of the parameters may be a dynamic parameter (such as ‘?’), which is necessary for retrieving the values of output and inout parameters. For example: CALL foo {(?, ?, ‘TX’)}. The curly braces are optional in your source code.

This is how stored procedures work in the current version of Pervasive.SQL.

- Triggers (CREATE TRIGGER, DROP TRIGGER) are supported. This support includes tracking dependencies that the trigger has on tables, and procedures, in the database.
- CONTAINS, NOT CONTAINS, BEGINS WITH are not supported.
- There is no support for dynamic SQL statement construction.
- LOOP: post conditional loops are not supported (REPEAT...UNTIL).
- ELSEIF: The conditional format uses IF ... THEN ... ELSE. There is no ELSEIF support.
- CASE: There is no support for CASE in stored procedures.

General Stored Procedure Engine Limitations

You should be aware of the following limitations before using stored procedures.

- There is no qualifier support in CREATE PROCEDURE or CREATE TRIGGER.
- Maximum length of a stored procedure variable name is 128 characters.
- Maximum length of a stored procedure name is 30 characters.
- Only partial syntactical validation occurs at CREATE PROCEDURE or CREATE TRIGGER time. Column names are not validated until run time.
There is currently no support for using subqueries everywhere expressions are used. For example, `set :arg = SELECT MIN(sal) FROM emp` is not supported. However, you could rewrite this query as `SELECT min(sal) INTO :arg FROM emp`.

- Only default handler error support.

### Limits to SQL Variables and Parameters
- Variable names must be preceded with a colon (:). This allows Pervasive.SQL 2000i Stored Procedure parser to differentiate between variables and column names.
- Variable names are case insensitive.
- No session variables are supported. Variables are local to the procedure.

### Limits to Cursors
- Positioned UPDATE does not accept tablename.

### Limits when using Long Data
- When you pass long data as arguments to an imbedded procedure, (that is, a procedure calling another procedure), the data is truncated to 65500 bytes.
- Long data arguments to and from procedures are limited to a total of 2 MB. (See MAXLEN_LONGPROCDATA define in spm.c)

Internally long data may be copied between cursors with no limit on data length. If a long data column is fetched from one statement and inserted into another, no limit is imposed. If, however, more than one destination is required for a single long data variable, only the first destination table receives multiple calls to PutData. The remaining columns are truncated to the first 65500 bytes. This is a limitation of the ODBC GetData mechanism.

### See Also
“DROP PROCEDURE” on page 2-71
CREATE TABLE

The CREATE TABLE statement creates a new table in a database.

CREATE TABLE contains functionality that goes beyond minimal or core SQL conformance. CREATE TABLE supports Referential Integrity features. Pervasive.SQL conforms closely to SQL 92 with the exception of CollIDList support.

Syntax

```sql
CREATE TABLE table-name [ IN DICTIONARY ]
    [ USING 'path_name' ] [ WITH REPLACE ]
    ( table-element [ , table-element ] ... )

table-name ::= user-defined-name

table-element ::= column-definition | table-constraint-definition

column-definition ::= column-name data-type [ DEFAULT default-value ] [ column-constraint [ column-constraint ] ... [ CASE | COLLATE collation-name ] ]

column-name ::= user-defined-name

data-type ::= data-type-name [ ( precision [ , scale ] ) ]

precision ::= integer

scale ::= integer

default-value ::= literal

literal ::= 'string'
    | number
    | { d 'date-literal' }
    | { t 'time-literal' }
    | { ts 'timestamp-literal' }

column-constraint ::= [ CONSTRAINT constraint-name ] col-constraint

constraint-name ::= user-defined-name

col-constraint ::= NOT NULL
    | UNIQUE
    | PRIMARY KEY
    | REFERENCES table-name [ ( column-name ) ] [ referential-actions ]
```
table-constraint-definition ::= [ CONSTRAINT constraint-name ] table-constraint
table-constraint ::= UNIQUE ( column-name [ , column-name ]... )
               | PRIMARY KEY ( column-name [ , column-name ]... )
               | FOREIGN KEY ( column-name [ , column-name ] ) REFERENCES table-name
               [ ( column-name [ , column-name ]... ) ] [ referential-actions ]
column-constraint ::= [ CONSTRAINT constraint-name ] col-constraint
constraint-name ::= user-defined-name
col-constraint ::= NOT NULL
               | UNIQUE
               | PRIMARY KEY
               | REFERENCES table-name [ ( column-name ) ] [ referential-actions ]
referential-actions ::= referential-update-action [ referential-delete-action ]
referential-update-action ::= ON UPDATE RESTRICT
referential-delete-action ::= ON DELETE CASCADE
               | ON DELETE RESTRICT

Remarks

Indexes must be created with the CREATE INDEX statement.

Foreign key constraint names must be unique in the dictionary. All other constraint names must be unique within the table in which they reside and must not have the same name as a column.

If the primary key name is omitted, the name of the first column in the key, prefixed by "PK_" is used as the name of the constraint.

If a reference column is not listed, the reference becomes, by default, the primary key of the table referenced. If a PK is unavailable, a "Key not found" error returns. You can avoid this situation by enumerating the target column.

If the foreign key name is omitted, the name of the first column in the key, prefixed by "FK_" is used as the name of the constraint. This is different behavior from previous versions of Pervasive.SQL.
If the UNIQUE key constraint name is omitted, the name of the first column in the constraint, prefixed by "UK_" is used as the name of the constraint.

If the NOT NULL key name is omitted, the name of the first column in the key, prefixed by "NN_" is used as the name of the constraint.

The maximum length of a constraint name is 20 characters. Pervasive.SQL use the left most 20 characters of the name after the prefix, if any, has been prepended.

A foreign key may reference the primary key of the same table (known as a self-referencing key).

If CREATE TABLE succeeds, the data file name for the created table is xxx.mkd, where xxx is the specified table name. If the table already exists, it is not replaced, and error -1303, “Table already exists” is signalled. The user must drop the table before replacing it.

**Delete Rule**

You can include an ON DELETE clause to define the delete rule Pervasive.SQL enforces for an attempt to delete the parent row to which a foreign key value refers. The delete rules you can choose are as follows:

- If you specify CASCADE, Pervasive.SQL uses the delete cascade rule. When a user deletes a row in the parent table, Scalable SQL deletes the corresponding row in the dependent table.
- If you specify RESTRICT, Pervasive.SQL enforces the delete restrict rule. A user cannot delete a row in the parent table if a foreign key value refers to it.

If you do not specify a delete rule, Pervasive.SQL applies the restrict rule by default.

**Update Rule**

Pervasive.SQL enforces the update restrict rule. This rule prevents the addition of a row containing a foreign key value if the parent table does not contain the corresponding primary key value. This rule is enforced whether or not you use the optional ON UPDATE clause, which allows you to specify the update rule explicitly.
IN DICTIONARY

See the discussion of IN DICTIONARY for “ALTER TABLE” on page 2-19.

USING

The USING keyword allows you to associate a CREATE TABLE or ALTER TABLE action with a particular data file.

Because Pervasive.SQL requires a Named Database to connect, the path_name provided must always be a simple file name or relative path and file name. Paths are always relative to the first Data Path specified for the Named Database to which you are connected.

The path/file name passed is partially validated when SQLPrepare is called. The following rules must be followed when specifying the path name:

- The text must be enclosed in single quotes, as shown in the grammar definition.
- Text must be 1 to 64 characters in length, such that the entry as specified fits in Xf$Loc in X$File. The entry is stored in Xf$Loc exactly as typed (trailing spaces are truncated and ignored).
- The path must be a simple, relative path. Paths that reference a server or volume are not allowed. For NetWare, a volume-based path (such as SYS:/path/testfile.btr) is not considered a simple, relative path.
- Relative paths containing a period (’.’ - current directory), double-period (’.’ - parent directory), slash ‘\’, or any combination of the three are allowed. The path must contain a file name representing the SQL table name (path_name cannot end in a slash ‘\’ or a directory name). All file names, including those specified with relative paths, are relative to the first Data Path as defined in the Named Database configuration.
- Root-based relative paths are also allowed. For example, assuming that the first data path is D:\PVS\DEMODATA, the SRDE interprets the path name in the following statement as D:\TEMP\TEST123.BTR.

CREATE TABLE t1 USING ‘\temp\test123.btr’ (c1 int)

- Slash (‘\’) characters in relative paths may be specified either UNIX style (‘/’) or in the customary backslash notation (‘\’), depending on your preference. You may use a mixture of the two types, if desired. This is a convenience feature since you may
know the directory structure scheme, but not necessarily know (or care) what type of server you are connected to. The path is stored in X$File exactly as typed. The SRDE engine converts the slash characters to the appropriate platform type when utilizing the path to open the file. Also, since data files share binary compatibility between all supported platforms, this means that as long as the directory structure is the same between platforms (and path-based file names are specified as relative paths), the database files and DDFs can be moved from one platform to another with no modifications. This makes for a much simpler cross-platform deployment with a standardized database schema.

- If specifying a relative path, the directory structure in the USING clause must first exist. The SRDE does not create directories to satisfy the path specified in the USING clause.

Include a USING clause to specify the physical location of the data file associated with the table. This is necessary when you are creating a table definition for an existing data file, or when you want to specify explicitly the name or physical location of a new data file.

If you do not include a USING clause, Pervasive.SQL generates a unique file name (based on the table name with the extension .MKD) and creates the data file in the first directory specified in the data file path associated with the database name.

If the USING clause points to an existing data file, the SRDE creates the table in the DDFs and returns SQL_SUCCESS_WITH_INFO. The informational message returned indicates that the dictionary entry now points to an existing data file. If you want CREATE TABLE to return only SQL_SUCCESS, specify IN DICTIONARY on the CREATE statement. If WITH REPLACE is specified (see below), then any existing data file with the same name is destroyed and overwritten with a newly created file.

**Note** Pervasive.SQL returns a successful status code if you specify an existing data file.

**WITH REPLACE**

Whenever WITH REPLACE is specified with the USING keyword, Pervasive.SQL automatically overwrites any existing file name with
the specified file name. The existing file is always overwritten as long as the operating system allows it.

WITH REPLACE affects only the data file, it never affects the DDFs. The following rules apply when using WITH REPLACE:

- WITH REPLACE can only be used with USING.
- When used with IN DICTIONARY, WITH REPLACE is ignored because IN DICTIONARY specifies that only the DDFs are affected.

Note: No data is lost or discarded if WITH REPLACE is used with ALTER TABLE. The newly created data file, even if overwriting an existing file, still contains all data from the previous file. You cannot lose data by issuing an ALTER TABLE command.

If you include WITH REPLACE in your CREATE TABLE statement, Pervasive.SQL creates a new data file to replace the existing file (if the file exists at the location you specified in the USING clause). Pervasive.SQL discards any data stored in the original file with the same name. If you do not include WITH REPLACE and a file exists at the specified location, Pervasive.SQL returns a status code and does not create a new file.

WITH REPLACE affects only the data file; it does not affect the table definition in the dictionary.

Examples

The following example creates a table named Billing with columns Student_ID, Transaction_Number, Log, Amount_Owed, Amount_Paid, Registrar_ID and Comments, using the specified data types.

```
CREATE TABLE Billing
    (Student_ID UBIGINT,
     Transaction_Number USMALLINT,
     Log TIMESTAMP,
     Amount_Owed DECIMAL(6,2),
     Amount_Paid DECIMAL(6,2),
     Registrar_ID DECIMAL(10,0),
     Comments LONGVARCHAR)
```
The following example creates a table named Faculty in the database with columns ID, Dept_Name, Designation, Salary, Building_Name, Room_Number, Rsch_Grant_Amount, and a primary key based on column ID.

```sql
CREATE TABLE Faculty
    (ID UBIGINT,
     Dept_Name CHAR(20) CASE,
     Designation CHAR(10) CASE,
     Salary CURRENCY,
     Building_Name CHAR(25) CASE,
     Room_Number UINTeger,
     Rsch_Grant_Amount DOUBLE,
     PRIMARY KEY (ID))
```

The following example is similar to the one just discussed, except the ID column, which is the primary key, is designated as UNIQUE.

```sql
CREATE TABLE organizations
    (ID UBIGINT UNIQUE,
     Name LONGVARCHAR,
     Advisor CHAR(30),
     Number_of_people INTEGER,
     Date_started DATE,
     Time_started TIME,
     Date_modified TIMESTAMP,
     Total_funds DOUBLE,
     Budget DECIMAL(2,2),
     Avg_funds REAL,
     President VARCHAR(20) CASE,
     Number_of_executives SMALLINT,
     Number_of_meetings TINYINT,
     Office UTINYINT,
     Active BIT,
     PRIMARY KEY(ID))
```

In the next example, assume that you need a table called StudentAddress to contain students’ addresses. You need to alter the Student table’s id column to be a primary key and then create a
StudentAddress table. (The Student table is part of the DEMODATA sample database.) Four ways are shown how to create the StudentAddress table.

First, make the *id* column of table Student a primary key.

```
ALTER TABLE Student ADD PRIMARY KEY (id)
```

This next statement creates a StudentAddress table to have a foreign key referencing the *id* column of table Student with the DELETE CASCADE rule. This means that whenever a row is deleted from the Student table (Student is the parent table in this case), all rows in the StudentAddress table with that same id are also deleted.

```
CREATE TABLE StudentAddress (id UBIGINT REFERENCES Student (id) ON DELETE CASCADE, addr CHAR(128))
```

This next statement creates a StudentAddress table to have a foreign key referencing the *id* column of table Student with the DELETE RESTRICT rule. This means that whenever a row is deleted from the Student table and there are rows in the StudentAddress table with that same id, an error occurs. You need to explicitly delete all the rows in StudentAddress with that *id* before the row in the Student table, the parent table, can be deleted.

```
CREATE TABLE StudentAddress (id UBIGINT REFERENCES Student (id) ON DELETE RESTRICT, addr CHAR(128))
```

This next statement creates a StudentAddress table to have a foreign key referencing the *id* column of table Student with the UPDATE RESTRICT rule. This means that if a row is added to the StudentAddress table that has an id that does not occur in the Student table, an error occurs. In other words, you must have a parent row before you can have foreign keys referring to that row. This is the default behavior of Pervasive.SQL. Moreover, Pervasive.SQL does not support any other UPDATE rules. Thus, whether this rule is stated explicitly or not makes no difference.

```
CREATE TABLE StudentAddress (id UBIGINT REFERENCES Student (id) ON UPDATE RESTRICT, addr CHAR(128))
```

This next statement creates a StudentAddress table to have a foreign key referencing the *id* column of table Student with the DELETE RESTRICT and UPDATE RESTRICT rules. The Pervasive.SQL parser accepts this syntax with RI rules. However, as stated above, the UPDATE RESTRICT rule is redundant since Pervasive.SQL does not behave any other way with respect to UPDATE rules.
CREATE TABLE StudentAddress (id UBIGINT REFERENCES Student (id) ON DELETE RESTRICT, addr CHAR(128))

This next example shows how to use an alternate collating sequence (ACS) when you create a table. The ACS file used is the sample one provided with Pervasive.SQL.

CREATE TABLE t5 (c1 CHAR(20) COLLATE 'c:\pvsw\samples\upper.alt')

Upper.alt treats upper and lower case letters the same for sorting. For example, if a database has values abc, ABC, DEF, and Def, inserted in that ordered, the sorting with upper.alt returns as abc, ABC, DEF, and Def. (The values abc and ABC, and the values DEF and Def are considered duplicates and are returned in the order in which they were inserted.) Normal ASCII sorting sequences upper case letters before lower case, such that the sorting would return as ABC, DEF, Def, abc.

See Also

“DROP TABLE” on page 2-72
CREATE TRIGGER

The CREATE TRIGGER statement creates a new trigger in a database. Triggers are a type of stored procedure that are automatically executed when data in a table is modified with an INSERT, UPDATE, or DELETE.

Unlike a regular stored procedure, a trigger cannot be executed directly nor can it have parameters. Triggers do not return a result set nor can they be defined on views.

Syntax

CREATE TRIGGER  trigger-name before-or-after ins-upd-del  ON  table-name
  [ ORDER  number  ]
  [ REFERENCING  referencing-alias  ] FOR EACH ROW
  [ WHEN  proc-search-condition  ]  proc-stmt

trigger-name ::= user-defined-name

before-or-after ::= BEFORE | AFTER

ins-upd-del ::= INSERT | UPDATE | DELETE

referencing-alias ::= OLD [ AS ]  correlation-name  [ NEW [ AS ]  correlation-name ]

  | NEW  [ AS ]  correlation-name  [ OLD  [ AS ]  correlation-name ]

 correlation-name ::= user-defined-name

Remarks

This function is an extension to SQL grammar as documented in the Microsoft ODBC Programmer’s Reference and implements a subset of the SQL 3/PSM (Persistent Stored Modules) specification.

Note In a trigger, the name of a variable must begin with a colon (:).

OLD (OLD correlation-name) and NEW (NEW correlation-name) can be used inside triggers, not in a regular stored procedure.

In a DELETE or UPDATE trigger, "OLD" or a OLD correlation-name must be prepended to a column name to reference a column in the row of data prior to the update or delete operation.
In an INSERT or UPDATE trigger, "NEW" or a NEW \textit{correlation-name} must be prepended to a column name to reference a column in the row about to be inserted or updated.

Trigger names must be unique in the dictionary.

Triggers are executed either before or after an UPDATE, INSERT, or DELETE statement is executed, depending on the type of trigger.

\section*{Examples}

The following example creates a trigger that records any new values inserted into the Tuition table into TuitionIDTable.

\begin{verbatim}
CREATE TABLE Tuitionidtable (PRIMARY KEY(id), id UBIGINT)
CREATE TRIGGER InsTrig
    BEFORE INSERT ON Tuition
    REFERENCING NEW AS Indata
    FOR EACH ROW
    INSERT INTO Tuitionidtable VALUES(Indata.ID);

An UPDATE on Tuition calls the trigger.

The following example shows how to keep two tables, A and B, synchronized with triggers. Both tables have the same structure.

\end{verbatim}

\begin{verbatim}
CREATE TABLE A (col1 INTEGER, col2 CHAR(10))
CREATE TABLE B (col1 INTEGER, col2 CHAR(10))
CREATE TRIGGER MyInsert
    AFTER INSERT ON A FOR EACH ROW
    INSERT INTO B VALUES (NEW.col1, NEW.col2);
CREATE TRIGGER MyDelete
    AFTER DELETE ON A FOR EACH ROW
    DELETE FROM B WHERE B.col1 = OLD.col1 AND B.col2 = OLD.col2;
CREATE TRIGGER MyUpdate
    AFTER UPDATE ON A FOR EACH ROW
    UPDATE B SET col1 = NEW.col1, col2 = NEW.col2 WHERE
        B.col1 = OLD.col1 AND B.col2 = OLD.col2;
\end{verbatim}
SQL Grammar Elements

Note that OLD and NEW in the example keep the tables synchronized only if table A is altered with non-positional SQL statements. If the ODBC SQLSetPos API or a positioned update or delete is used, then the tables stay synchronized only if table A does not contain any duplicate records. An SQL statement cannot be constructed to alter one record but leave another duplicate record unaltered.

See Also

“DROP TRIGGER” on page 2-73
CREATE VIEW

Use the CREATE VIEW statement to define a stored view on the database.

Syntax

CREATE VIEW view-name [ ( column-name [ , column-name ]... ) ]
AS query-specification

view-name ::= user-defined-name

Remarks

A view is a database object that stores a query and behaves like a table. A view contains a set of columns and rows. Data accessed through a view is stored in one or more tables; the tables are referenced by SELECT statements. Data returned by a view is produced dynamically every time the view is referenced.

The maximum length of a view name is 20 characters. The maximum number of columns in a view is 256. There is a 64KB limit on view definitions.

A grouped view is one that contains the GROUP BY clause and/or an aggregate function in the SELECT list. Grouped views are not allowed in the FROM clause of a SELECT statement with a join (that is, with multiple tables). Grouped views are not allowed in the FROM clause of a SELECT statement with a GROUP BY.

Grouped views may not be used in a subquery.

The WHERE clause against a grouped view is a HAVING clause, and appended to the HAVING clauses of the grouped view.

View definitions cannot contain UNION operators. The operator UNION cannot be applied to any SQL statement that references one or more views.

View definitions cannot contain procedures, nor can they contain an ORDER BY.

Examples

The following statement creates a view named vw_Person, which creates a phone list of all the people enrolled in a university. This view lists the last names, first names and telephone numbers with a heading for each column. The Person table is part of the DEMODATA sample database.

CREATE VIEW vw_Person (lastn,firstn,phone) AS SELECT Last_Name, First_Name,Phone FROM Person
In a subsequent query on the view, you may use the column headings in your `SELECT` statement, as shown in the next example.

```sql
SELECT lastn, firstn FROM vw_Person
```

See Also

“DROP VIEW” on page 2-74
DECLARE

Remarks  Use the DECLARE statement to define an SQL variable.
In Pervasive.SQL 2000i, this statement is allowed only inside of a
stored procedure or a trigger, since cursors and variables are allowed
only inside of stored procedures and triggers.
The name of a variable must begin with a colon (:), both in the
definition and use of the variable or parameter.
A variable must be declared before it can set to a value.

Examples  The following examples declare the variables :Counter and
:CurrentCapacity.
DECLARE :counter INTEGER = 0;
DECLARE :CurrentCapacity INTEGER = 0;

See Also  “CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
“SET VARIABLE” on page 2-131
DECLARE CURSOR

The DECLARE CURSOR statement defines an SQL cursor.

Syntax

```
DECLARE cursor-name CURSOR FOR select-statement
    [ FOR UPDATE | FOR READ ONLY ]
```

Remarks

In Pervasive.SQL 2000i, this statement is only allowed inside of a stored procedure or a trigger, since cursors and variables are only allowed inside of stored procedures and triggers.

The default behavior for cursors is read-only. Therefore, you must use FOR UPDATE to explicitly designate an update (write or delete).

Examples

The following example creates a cursor that selects values from the Degree, Residency, and Cost_Per_Credit columns in the Tuition table and orders them by ID number.

```
DECLARE BTUCursor CURSOR
    FOR SELECT Degree, Residency, Cost_Per_Credit
    FROM Tuition
    ORDER BY ID;
```

The following example uses FOR UPDATE to ensure a delete.

```
CREATE PROCEDURE MyProc(IN :CourseName CHAR(7)) AS
BEGIN
    DECLARE c1 CURSOR FOR SELECT name FROM course
        WHERE name = :CourseName FOR UPDATE;
    OPEN c1;
    FETCH NEXT FROM c1 INTO :CourseName;
    DELETE WHERE CURRENT OF c1;
    CLOSE c1;
END;

CALL MyProc('HIS 305')
```

See Also

“CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
DELETE (positioned)

Use the positioned DELETE statement to remove the current row of a view associated with an SQL cursor.

Syntax

```
DELETE WHERE CURRENT OF cursor-name
```

Remarks

This statement is allowed in stored procedures, triggers, and at the session level.

Note

Even though positioned DELETE is allowed at the session level, the DECLARE CURSOR statement is not. Use the SQLGetCursorName() API to obtain the cursor name of the active result set.

Examples

The following sequence of statements provide the setting for the positioned DELETE statement. The required statements for the positioned DELETE statement are DECLARE CURSOR, OPEN CURSOR, and FETCH FROM cursorname.

The Modern European History class has been dropped from the schedule, so this example deletes the row for Modern European History (HIS 305) from the Course table in the sample database:

```
CREATE PROCEDURE DropClass();
    DECLARE  :CourseName CHAR(7);
    DECLARE  c1 CURSOR
        FOR SELECT name
            FROM COURSE
            WHERE name = :CourseName;
    BEGIN
        SET  :CourseName  = 'HIS 305';
        OPEN c1;
        FETCH NEXT FROM c1 INTO :CourseName;
        DELETE WHERE CURRENT OF c1;
    END;
```

See Also

“CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
DELETE
This statement deletes specified rows from a database table.

**Syntax**
```
DELETE FROM table-name [ alias-name ]
[ WHERE search-condition ]
```

**Remarks**
INSERT, UPDATE, and DELETE statements behave in an atomic manner. That is, if an insert, update, or delete of more than one row fails, then all insertions, updates, or deletes of previous rows by the same statement are rolled back.

**Examples**
The following statements deletes the row for first name Ellen from the person table in the sample database.
```
DELETE FROM person WHERE First_Name = 'Ellen'
```

The following statement deletes the row for Modern European History (HIS 305) from the course table in the sample database:
```
DELETE FROM Course WHERE Name = 'HIS 305'
```
DISTINCT

Remarks
Include the DISTINCT keyword in your SELECT statement to direct Pervasive.SQL to remove duplicate values from the result. By using DISTINCT, you can retrieve all unique rows that match the SELECT statement’s specifications.

The following rules apply to using the DISTINCT keyword:

You can use DISTINCT in any statement that includes subqueries.

The DISTINCT keyword is ignored if the selection list contains an aggregate; the aggregate guarantees that no duplicate rows result.

The following usage of DISTINCT is not allowed:

```
SELECT DISTINCT column1, DISTINCT column2
```

Examples
The following statement retrieves all the unique courses taught by Professor Beir (who has a Faculty ID of 111191115):

```
SELECT DISTINCT c.Name
  FROM Course c, class cl
  WHERE c.name = cl.name AND cl.faculty_id = '111191115'
```

See Also
“SELECT” on page 2-118
**DROP GROUP**

This statement drops one or more groups in a secured database.

**Syntax**

```
DROP GROUP  group-name  [ ,  group-name ]...
```

**Remarks**

Separate multiple group names with a comma.

**Examples**

The following example drops the group pervasive.

```
DROP GROUP pervasive
```

The following example uses a list to drop groups.

```
DROP GROUP pervasive_dev, pervasive_marketing
```

**See Also**

“CREATE GROUP” on page 2-39
DROP INDEX

This statement drops a specific index from a designated table.

Syntax

```
DROP INDEX [ table-name . ] index-name
```

Examples

The following statement drops the named index from the Faculty table.

```
DROP INDEX Faculty.Dept
```

See Also

“CREATE INDEX” on page 2-40
DROP PROCEDURE
This statement removes one or more stored procedures from the current database.

Syntax
DROP PROCEDURE procedure-name

Examples
The following statement drops the stored procedure myproc from the dictionary:
DROP PROCEDURE myproc

See Also
“CREATE PROCEDURE” on page 2-42
**DROP TABLE**

This statement removes a table from a designated database.

**Syntax**

```
DROP TABLE  table-name  [  IN DICTIONARY  ]
```

**Remarks**

CASCADE and RESTRICT are not supported.

If any triggers depend on the table, the table is not dropped.

If a transaction is in progress and refers to the table, then an error is signalled and the table is not dropped.

The drop of table fails if other tables depend on the table to be dropped.

If a primary key exists, it is dropped. The user need not drop the primary key before dropping the table. If the primary key of the table is referenced by a constraint belonging to another table, then the table is not dropped and an error is signalled.

If the table has any foreign keys, then those foreign keys are dropped.

If the table has any other constraints (for example, NOT NULL, CHECK, UNIQUE, or NOT MODIFIABLE), then those constraints are dropped when the table is dropped.

**IN DICTIONARY**

See the discussion of IN DICTIONARY for “ALTER TABLE” on page 2-19.

**Examples**

The following statement drops the class table definition from the dictionary.

```
DROP TABLE Class
```

**See Also**

“ALTER TABLE” on page 2-19

“CREATE TABLE” on page 2-50
DROP TRIGGER

This statement removes a trigger from the current database.

Syntax

```sql
DROP TRIGGER trigger-name
```

Examples

The following example drops the trigger named InsTrig.

```sql
DROP TRIGGER InsTrig
```

See Also

“CREATE TRIGGER” on page 2-59
**DROP VIEW**

This statement removes a specified view from the database.

**Syntax**

```
DROP VIEW view-name
```

**Remarks**

[CASCADE | RESTRICT] is not supported.

**Examples**

The following statement drops the vw_person view definition from the dictionary.

```
DROP VIEW vw_person
```

**See Also**

“CREATE VIEW” on page 2-62
END

Remarks

See the discussion for BEGIN [ATOMIC] on page 2-32.
**EXISTS**

**Remarks**
Use the EXISTS keyword to test whether rows exist in the result of the subquery. For every row the outer query evaluates, Pervasive.SQL 2000i tests for the existence of a related row from the subquery. Pervasive.SQL 2000i includes in the statement’s result table each row from the outer query that corresponds to a related row from the subquery.

**Examples**
For example, the following statement returns a list containing only persons who have a 4.0 grade point average:

```sql
SELECT * FROM Person p WHERE EXISTS
    (SELECT * FROM Enrolls e WHERE e.Student_ID = p.id
     AND Grade = 4.0)
```

**See Also**
“SELECT” on page 2-118
FETCH

Remarks

A FETCH statement positions an SQL cursor on a specified row of a table and retrieves values from that row by placing them into the variables in the target list.

Examples

The FETCH statement in this example retrieves values from cursor c1 into the CourseName variable. The Positioned UPDATE statement in this example updates the row for Modern European History (HIS 305) in the Course table in the DEMODATA sample database:

```
CREATE PROCEDURE UpdateClass();
BEGIN
    DECLARE :CourseName CHAR(7);
    DECLARE :OldName CHAR(7);
    DECLARE  c1 CURSOR FOR SELECT name FROM course WHERE name = :CourseName;
    OPEN c1;
    SET :CourseName = 'HIS 305';
    FETCH NEXT FROM c1 INTO :OldName;
    UPDATE SET name = 'HIS 306' WHERE CURRENT OF c1;
END;
```

See Also

“CREATE PROCEDURE” on page 2-42
FOREIGN KEY

Remarks

Include the FOREIGN KEY keywords in the ADD clause to add a foreign key to a table definition.

If you add a foreign key to a table that already contains data, use the Pervasive Control Center utility to find any data that does not conform to the new referential constraint. See the Pervasive.SQL User’s Guide for information about this utility.

Note You must be logged in to the database using a database name before you can add a foreign key or conduct any other referential integrity (RI) operation. Also, when security is enabled, you must have the Reference right on the table to which the foreign key refers before you can add the key.

Include a FOREIGN KEY clause in your CREATE TABLE statement to define a foreign key on a dependent table. In addition to specifying a list of columns for the key, you can define a name for the key.

The columns in the foreign key column may be nullable; however, you should ensure that pseudo-null columns do not exist in a MicroKernel index that does not index pseudo-null values.

The foreign key name must be unique in the dictionary. If you omit the foreign key name, Pervasive.SQL uses the name of the first column in the key as the foreign key name. This can cause a duplicate foreign key name error if your dictionary already contains a foreign key with that name.

When you specify a foreign key, Pervasive.SQL creates an index on the columns that make up the key. This index has the same attributes as the index on the corresponding primary key except that it allows duplicate values. To assign other attributes to the index, create it explicitly using a CREATE INDEX statement. Then, define the foreign key with an ALTER TABLE statement. When you create the index, ensure that it does not allow null values and that its case and collating sequence attributes match those of the index on the corresponding primary key column.

The columns in a foreign key must be the same data types and lengths and in the same order as the referenced columns in the primary key. The only exception is that you can use an integer column in the foreign key to refer to an IDENTITY or
SMALLIDENTITY column in the primary key. In this case, the two columns must be the same length.

Pervasive.SQL checks for anomalies in the foreign keys before it creates the table. If it finds conditions that violate previously defined referential integrity (RI) constraints, it generates a status code and does not create the table.

When you define a foreign key, you must include a REFERENCES clause indicating the name of the table that contains the corresponding primary key. The primary key in the parent table must already be defined. In addition, if security is enabled on the database, you must have the Reference right on the table that contains the primary key.

You cannot create a self-referencing foreign key with the CREATE TABLE statement. Use an ALTER TABLE statement to create a foreign key that references the primary key in the same table.

Also, you cannot create a primary key and a foreign key on the same set of columns in a single statement. Therefore, if the primary key of the table you are creating is also a foreign key on another table, you must use an ALTER TABLE statement to create the foreign key.

Examples

The following statement adds a new foreign key to the Class table. (The Faculty column is defined as an index that does not include null values.)

```
ALTER TABLE Class ADD CONSTRAINT Teacher FOREIGN KEY (Faculty_ID) REFERENCES Faculty ON DELETE RESTRICT
```

In this example, the restrict rule for deletions prevents someone from removing a faculty member from the database without first either changing or deleting all of that faculty’s classes.

See Also

“ALTER TABLE” on page 2-19

“CREATE TABLE” on page 2-50
GRANT

This statement creates new user IDs and gives permissions to specific users in a secured database.

Syntax

```
GRANT CREATETAB TO public-or-user-or-group-name [ , user-or-group-name ] . . .

GRANT LOGIN TO user-password [ , user-password ] . . . [ IN GROUP 
    group-name ]

GRANT table-privilege ON [ TABLE ] table-name [ owner-name ]
    TO user-or-group-name [ , user-or-group-name ] . . .
```

```
table-privilege ::= ALL
    | SELECT [ ( column-name [ , column-name ] . . . ) ]
    | UPDATE [ ( column-name [ , column-name ] . . . ) ]
    | INSERT [ ( column-name [ , column-name ] . . . ) ]
    | DELETE
    | ALTER
    | REFERENCES
```

```
user-password :::= user-name [ : ] password
```

```
public-or-user-or-group-name :::= PUBLIC | user-or-group-name
```

```
user-or-group-name :::= user-name | group-name
```

```
user-name :::= user-defined-name
```

```
owner-name :::= user-defined-name
```

Remarks

CREATETAB and LOGIN arguments are extensions to the core SQL grammar.

Although an optional column list is in the syntax for the INSERT, ALTER, and REFERENCES privileges, the Pervasive.SQL Engine signals a "not supported" error if any GRANT INSERT, GRANT ALTER, or GRANT REFERENCES statement contains a column list.

Note ANSI SQL 3 permits column lists for INSERT, ALTER, REFERENCES, SELECT and UPDATE.
Users and Groups

Relational security is based on the existence of a default user named “Master” who has full access to the database when security is first turned on. Initially, no password is required for the Master user.

**Caution** If you turn on security, be sure to specify a password with a significant length, at least five characters. Do not leave the password field blank because doing so creates a major security risk for your database.

The Master user can create groups and other users and define sets of data access privileges for these groups and users.

If you want to grant the same level of access to all users and avoid having to set up individual groups and users, you can grant the desired level of access to PUBLIC. The default user PUBLIC represents any user connecting with or without a password.

**Note** If you wish to use groups, you must set up the groups before creating users. You cannot add a user to a group after you have already created the user.

You can use the Users node in PCC to perform these tasks. You can also use GRANT and REVOKE statements to perform these tasks.

User name and password must be enclosed in double quotes if they contain spaces or other non-alphanumeric characters.

See *Pervasive.SQL User’s Guide* for further information about users and groups.

Owner Name

An owner name is a password required to gain access to a Btrieve file. There is no relation between an owner name and any system user name or database user name. You should think of an owner name as a simple file password.

If you have a Btrieve owner name set on a file that is a table in a secure ODBC database, the Master user of the ODBC database must use the owner name in any GRANT statement to grant privileges on the given table to any user, including the Master user.
After the **GRANT** statement containing the owner name has been issued for a given user, that user can access the specified table by logging into the database, without specifying the owner name each time.

If a user tries to access a table through ODBC that has a Btrieve owner name, the access will not be allowed unless the Master user has granted privileges on the table to the user, with the correct owner name in the **GRANT** statement.

If a table has an owner name with the Read-Only attribute, the Master user automatically has **SELECT** rights on this table without specifically granting himself/herself the **SELECT** rights with the owner name.

**Examples**

A **GRANT ALL** statement grants the **INSERT**, **UPDATE**, **ALTER**, **SELECT**, **DELETE** and **REFERENCES** rights to the specified user or group. In addition, the user or group is granted the **CREATE TABLE** right for the dictionary.

The following statement grants all these privileges to dannyd for table Class.

```
GRANT ALL ON Class TO dannyd
```

This statement grants the **ALTER** privilege to user debieq.

```
GRANT ALTER ON Class TO debieq
```

The following statement gives **INSERT** privileges to keithv and miked on table Class.

```
GRANT INSERT ON Class TO keithv, miked
```

The following statement grants **INSERT** privileges on two columns, First_name and Last_name, in the person table to users keithv and brendanb

```
GRANT INSERT(First_name,last_name) ON Person to keithv,brendanb
```

The following example grants **CREATE TABLE** rights to users aideenw and punitas

```
GRANT CREATETAB TO aideenw, punitas
```
This next statement grants login rights to a user named ravi and specifies his password as “password.”

GRANT LOGIN TO ravi:password

The user name and password refer to Pervasive.SQL databases and are not related to user names and passwords set at level of the operating system. Pervasive.SQL user names, groups, and passwords are set through the Pervasive Control Center (PCC).

The following example grants login rights to users named dannyd and travisk and specifies their passwords as 'password' and 1234567 respectively.

GRANT LOGIN TO dannyd:password, travisk:1234567

If there are spaces in a name you may use double quotes as in the following example. This statement grants login rights to user named Jerry Gentry and Punita and specifies their password as sun and moon respectively.

GRANT LOGIN TO 'Jerry Gentry' :sun, Punita:moon

The following example grants the login rights to a user named Jerry Gentry with password 123456 and a user named travisk with password abcdef. It also adds them to the group pervasive_dev

GRANT LOGIN TO 'Jerry Gentry' :123456, travisk:abcdef IN GROUP pervasive_dev

To grant privileges on a table that has a Btrieve owner name, the Master user has to supply the correct owner name in the GRANT statement.

The following example grants the SELECT rights to the Master user on table T1 that has a Btrieve owner name of “abcd.”

GRANT SELECT ON T1 'abcd' TO Master

The Master user has all rights on a table that does not have an owner name. You can set an owner name on a table with the Maintenance utility. The Btrieve owner name is case sensitive.
See Also

“REVOKE” on page 2-112
“SET SECURITY” on page 2-129
“CREATE GROUP” on page 2-39
“DROP GROUP” on page 2-69
GROUP BY

Remarks

In addition to the GROUP BY syntax in a SELECT statement as specified in the Microsoft ODBC Programmer’s Reference, the Pervasive.SQL Engine supports an extended GROUP BY syntax that can include vendor strings.

A GROUP BY query returns a result set which contains one row of the select list for every group encountered. (See the Microsoft ODBC Programmer’s Reference for the syntax of a select list.)

Examples

The following example uses the course table to produce a list of unique departments:

```
SELECT Dept_Name FROM Course GROUP BY Dept_Name
```

In the next example, the result set contains a list of unique departments and the number of courses in each department:

```
SELECT Dept_Name, COUNT(*) FROM Course GROUP BY Dept_Name
```

Note that COUNT(expression) counts all non-NULL values for an expression across a predicate. COUNT(*) counts all values, including NULL values.

The rows operated on by the set function are those rows remaining after the WHERE search condition is applied. In this example, only those rows in the faculty table that have Salary > 80000 are counted:

```
SELECT COUNT(*) FROM Faculty WHERE Salary > 80000 GROUP BY Dept_Name
```

The following example shows an extended GROUP BY that includes vendor strings.

```
SELECT --(*vendor(Microsoft), product(ODBC) fn
left(at1.col2, 1) *)-- FROM at1 GROUP BY --
(*vendor(Microsoft), product(ODBC) fn left(at1.col2, 1) *)-- ORDER BY --(*vendor(Microsoft), product(ODBC) fn
left(at1.col2, 1) *)-- DESC
```
**See Also**

“SELECT” on page 2-118

“GRANT” on page 2-80

“REVOKE” on page 2-112
HAVING

Remarks
Use a HAVING clause in conjunction with a GROUP BY clause within SELECT statements to limit your view to groups whose aggregate values meet specific criteria.

The expressions in a HAVING clause may contain constants, set functions or an exact replica of one of the expressions in the GROUP BY expression list.

The Pervasive.SQL Engine does not support HAVING without GROUP BY.

Examples
The following example returns department names where the count of course names is greater than 5.

```
SELECT Dept_Name, COUNT(*) FROM Course GROUP BY Dept_Name HAVING COUNT(*) > 5
```

Note that COUNT(expression) counts all non-NULL values for an expression across a predicate. COUNT(*) counts all values, including NULL values.

The next example returns department name that matches Accounting and has a number of courses greater than 5.

```
SELECT Dept_Name, COUNT(*) FROM Course GROUP BY Dept_Name HAVING COUNT(*) > 5 AND Dept_Name = 'Accounting'
```

See Also
“SELECT” on page 2-118
**IF**

**Remarks**

IF statements provide conditional execution based on the value of a condition. The IF . . . THEN . . . [ELSE . . .] construct controls flow based on which of two statement blocks will be executed.

You may use IF statements in the body of both stored procedures and triggers.

**Examples**

The following example uses the IF statement to set the variable Negative to either 1 or 0, depending on whether the value of vInteger is positive or negative.

```sql
IF (:vInteger < 0) THEN
    SET :Negative = '1';
ELSE
    SET :Negative = '0';
END IF;
```

The following example uses the IF statement to test the loop for a defined condition (SQLSTATE = '02000'). If it meets this condition, then the WHILE loop is terminated.

```sql
FETCH_LOOP:
    WHILE (:counter < :NumRooms) DO
        FETCH NEXT FROM cRooms INTO :CurrentCapacity;
        IF (SQLSTATE = '02000') THEN
            LEAVE FETCH_LOOP;
        END IF;
        SET :counter = :counter + 1;
        SET :TotalCapacity = :TotalCapacity + :CurrentCapacity;
    END WHILE;
```

**See Also**

“CREATE PROCEDURE” on page 2-42

“CREATE TRIGGER” on page 2-59
REM Remarks
Use the IN operator to test whether the result of the outer query is included in the result of the subquery. The result table for the statement includes only rows the outer query returns that have a related row from the subquery.

EX Examples
The following example lists the names of all students who have taken Chemistry 408:

```
SELECT p.First_Name + ' ' + p.Last_Name
FROM Person p, Enrolls e
WHERE (p.id = e.student_id)
AND (e.class_id IN
 (SELECT c.ID FROM Class c WHERE c.Name = 'CHE 408'))
```

Pervasive.SQL first evaluates the subquery to retrieve the ID for Chemistry 408 from the Class table. It then performs the outer query, restricting the results to only those students who have an entry in the Enrolls table for that course.

Often, you can perform IN queries more efficiently using either the EXISTS keyword or a simple join condition with a restriction clause. Unless the purpose of the query is to determine the existence of a value in a subset of the database, it is more efficient to use the simple join condition because Pervasive.SQL optimizes joins more efficiently than it does subqueries.

SA See Also
“SELECT” on page 2-118
ODBC Engine Reference

**INSERT**

This statement inserts column values into one or more tables.

**Syntax**

```
INSERT INTO table-name [ alias-name ]
   [ ( column-name [ , column-name ]... ) ]
insert-values
```

```
insert-values ::= values-clause
   | query-specification
```

```
values-clause ::= VALUES ( expression [ , expression ]... )
```

**Remarks**

INSERT, UPDATE, and DELETE statements behave in an atomic manner. That is, if an insert, update, or delete of more than one row fails, then all insertions, updates, or deletes of previous rows by the same statement are rolled back.

All data types for data created prior to Pervasive.SQL 2000 (legacy data) report back as nullable. This means that you can INSERT NULL into any legacy column type without pseudo-NULL conversion. The following data types are treated as pseudo-NULL by default:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>Yes</td>
</tr>
<tr>
<td>Decimal</td>
<td>Yes</td>
</tr>
<tr>
<td>Money</td>
<td>Yes</td>
</tr>
<tr>
<td>Numeric</td>
<td>Yes</td>
</tr>
<tr>
<td>NumericSA</td>
<td>Yes</td>
</tr>
<tr>
<td>NumericSTS</td>
<td>Yes</td>
</tr>
<tr>
<td>Timestamp</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(Normally, when you convert a legacy column to pseudo-NULL, you lose one of the binary values, forfeiting it so that you can query the column for NULL. These data types, however, because of their design, have a different, unique internal value for NULL in addition to their normal data range. With these data types, no binary values are lost if they are converted to NULL so there is no harm considering them as pseudo-NULL by default.)

The rest of the data types are considered “legacy nullable,” meaning that NULL may be inserted into them. When values are queried, however, the non-NULL binary equivalent is returned. This same binary equivalent must be used in WHERE clauses to retrieve specific values.
The binary equivalents are:

- 0 for Binary types
- Empty string from string and BLOB types (legacy types LVAR and NOTE)

Pervasive.SQL 2000i allows you to use the variables CURTIME,
CURDATE and NOW in INSERT statements to insert the current
date, time and timestamp values.

**Examples**

The following statement adds data to the Course table by directly
specifying the values in three VALUES clauses:

```
INSERT INTO Course(Name, Description, Credit_Hours)
VALUES ('CHE 308', 'Organic Chemistry II', 4)
INSERT INTO Course(Name, Description, Credit_Hours)
VALUES ('ENG 409', 'Creative Writing II', 3)
INSERT INTO Course(Name, Description, Credit_Hours)
VALUES ('MAT 307', 'Probability II', 4)
```

The following INSERT statement uses a SELECT clause to retrieve
from the Student table the ID numbers of students who have taken
classes.

The statement then inserts the ID numbers into the Billing table.

```
INSERT INTO Billing (Student_ID)
SELECT ID
FROM Student
WHERE Cumulative_Hours > 0
```

The following example illustrates the use of the CURTIME,
CURDATE and NOW variables to insert the current date, time and
timestamp values inside an INSERT statement.

```
CREATE TABLE Timetbl (c1 TIME, c2 DATE, c3 TIMESTAMP)
INSERT INTO Timetbl(c1, c2, c3) VALUES(CURTIME, CURDATE, NOW)
```
The following example shows what occurs when you use `INSERT` for `IDENTITY` columns and columns with default values.

```sql
CREATE TABLE (id IDENTITY, c1 INTEGER DEFAULT 100)
INSERT INTO (id) VALUES (0)
INSERT INTO t VALUES (0,1)
INSERT INTO t VALUES (10,10)
INSERT INTO t VALUES (0,2)
INSERT INTO t (c1) VALUES (3)
SELECT * FROM t
```

The `SELECT` shows the table contains the following rows:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>

The first row illustrates that if “0” is specified in the VALUES clause for an `IDENTITY` column, then the value inserted is “1” if the table is empty.

The first row also illustrates that if no value is specified in the VALUES clause for a column with a default value, then the specified default value is inserted.

The second row illustrates that if “0” is specified in the VALUES clause for an `IDENTITY` column, then the value inserted is one greater than the largest value in the `IDENTITY` column.

The second row also illustrates that if a value is specified in the VALUES clause for a column with a default value, then the specified value overrides the default value.

The third row illustrates that if a value other than “0” is specified in the VALUES clause for an `IDENTITY` column, then that value is inserted. If a row already exists that contains the specified value for the `IDENTITY` column, then the message “The record has a key field containing a duplicate value(Btrieve Error 5)" is returned and the `INSERT` fails.
The fourth rows shows again that if “0” is specified in the VALUES clause for an IDENTITY column, then the value inserted is one greater than the largest value in the IDENTITY column. This is true even if “gaps” exist between the values (that is, the absence of one or more rows with IDENTITY column values less than the largest value).

The fifth row illustrates that if no value is specified in the VALUES clause for an IDENTITY column, then the value inserted is one greater than the largest value in the IDENTITY column.

See Also

“CREATE TABLE” on page 2-50

“SELECT” on page 2-118
JOIN

You can specify a single table or view, multiple tables, or a single view and multiple tables. When you specify more than one table, the tables are said to be joined.

Syntax

\[
\text{join-definition} ::= \text{table-reference} \ [\text{INNER}] \ JOIN \ \text{table-reference} \ \text{ON} \ \text{search-condition} \\
| \ \text{table-reference} \ \text{CROSS JOIN} \ \text{table-reference} \\
| \ \text{outer-join-definition}
\]

\[
\text{outer-join-definition} ::= \text{table-reference} \ \text{outer-join-type} \ JOIN \ \text{table-reference} \ \text{ON} \ \text{search-condition}
\]

\[
\text{outer-join-type} ::= \text{LEFT} [\text{OUTER}] | \text{RIGHT} [\text{OUTER}] | \text{FULL} [\text{OUTER}]
\]

The following example illustrates a two-table outer join:

```
SELECT * FROM Person LEFT OUTER JOIN Faculty ON Person.ID = Faculty.ID
```

The following example shows an outer join embedded in a vendor string. The “OJ” can be either upper or lower case.

```
SELECT t1.deptno, ename FROM {OJ emp t2 LEFT OUTER JOIN dept t1 ON t2.deptno=t1.deptno}
```

The Pervasive ODBC Engine Interface supports two-table outer joins as specified in the Microsoft ODBC Programmer’s Reference.

In addition to simple two-table outer joins, the Pervasive ODBC Engine Interface supports n-way nested outer joins.

The outer join may or may not be embedded in a vendor string. If a vendor string is used, Pervasive ODBC Engine Interface strips it off and parses the actual outer join text.
LEFT OUTER

The Pervasive.SQL Engine has implemented LEFT OUTER JOIN using SQL92 (SQL2) as a model. The syntax is a subset of the entire SQL92 syntax which includes cross joins, right outer joins, full outer joins, and inner joins. The TableRefList below occurs after the FROM keyword in a SELECT statement and before any subsequent WHERE, HAVING, and other clauses. Note the recursive nature of TableRef and LeftOuterJoin—a TableRef can be a left outer join that can include TableRefs which, in turn, can be left outer joins and so forth.

TableRefList :

  TableRef [ TableRefList ]
  | TableRef
  | OuterJoinVendorString [ TableRefList ]

TableRef :

  TableName [CorrelationName]
  | LeftOuterJoin
  | ( LeftOuterJoin )

LeftOuterJoin :

TableRef LEFT OUTER JOIN TableRef ON SearchCond

The search condition (SearchCond) contains join conditions which in their usual form are \( LT.ColumnName = RT.ColumnName \), where \( LT \) is left table, \( RT \) is right table, and \( ColumnName \) represents some column within a given domain. Each predicate in the search condition must contain some non-literal expression.

The implementation of left outer join goes beyond the syntax in the Microsoft ODBC Programmer’s Reference.

Vendor Strings

The syntax in the previous section includes but goes beyond the ODBC syntax in the Microsoft ODBC Programmer’s Reference. Furthermore, the vendor string escape sequence at the beginning and end of the left outer join does not change the core syntax of the outer join.

The Pervasive.SQL Engine accepts outer join syntax without the vendor strings. However, for applications that want to comply with ODBC across multiple databases, the vendor string construction should be used. Because ODBC vendor string outer joins do not
support more than two tables, it may be necessary to use the syntax shown following Table 2-10 on page 2-96.

Examples

The following four tables are used in the examples in this section.

**Table 2-7  Emp Table**

<table>
<thead>
<tr>
<th>FirstName</th>
<th>LastName</th>
<th>DeptID</th>
<th>EmpID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Franky</td>
<td>Avalon</td>
<td>D103</td>
<td>E1</td>
</tr>
<tr>
<td>Gordon</td>
<td>Lightfoot</td>
<td>D102</td>
<td>E2</td>
</tr>
<tr>
<td>Lawrence</td>
<td>Welk</td>
<td>D101</td>
<td>E3</td>
</tr>
<tr>
<td>Bruce</td>
<td>Cockburn</td>
<td>D102</td>
<td>E4</td>
</tr>
</tbody>
</table>

**Table 2-8  Dept Table**

<table>
<thead>
<tr>
<th>DeptID</th>
<th>LocID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>D101</td>
<td>L1</td>
<td>TV</td>
</tr>
<tr>
<td>D102</td>
<td>L2</td>
<td>Folk</td>
</tr>
</tbody>
</table>

**Table 2-9  Addr Table**

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Street</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>101 Mem Lane</td>
</tr>
<tr>
<td>E2</td>
<td>14 Young St.</td>
</tr>
</tbody>
</table>

**Table 2-10  Loc Table**

<table>
<thead>
<tr>
<th>LocID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>PlanetX</td>
</tr>
<tr>
<td>L2</td>
<td>PlanetY</td>
</tr>
</tbody>
</table>

The following example shows a simple two-way Left Outer Join:
SELECT * FROM Emp LEFT OUTER JOIN Dept ON Emp.DeptID = Dept.DeptID

This two-way outer join produces the following result set:

Table 2-11  Two-way Left Outer Join

<table>
<thead>
<tr>
<th>Emp</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FirstName</td>
<td>LastName</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>Franky</td>
<td>Avalon</td>
</tr>
<tr>
<td>Gordon</td>
<td>Lightfoot</td>
</tr>
<tr>
<td>Lawrence</td>
<td>Welk</td>
</tr>
<tr>
<td>Bruce</td>
<td>Cockburn</td>
</tr>
</tbody>
</table>

Notice the NULL entry for Franky Avalon in the table. That is because no DeptID of D103 was found in the Dept table. In a standard (INNER) join, Franky Avalon would have been dropped from the result set altogether.

Algorithm

The algorithm that the Pervasive.SQL 2000i Engine uses for the previous example is this:

Taking the left table, traverse the right table, and for every case where the ON condition is TRUE for the current right table row, return a result set row composed of the appropriate right table row appended to the current left-table row.

If there is no right table row where the ON condition is TRUE, (it is FALSE for all right table rows given the current left table row), create a row instance of the right table with all column values NULL.

That result set, combined with the current left-table row for each row, is indexed in the returned result set. The algorithm is repeated for every left table row to build the complete result set. In the simple two-way left outer join shown previously, Emp is the left table and Dept is the right table.
**Note** Although irrelevant to the algorithm, the appending of the left table to the right table assumes proper projection as specified in the select list of the query. This projection ranges from all columns (for example, `SELECT * FROM ...`) to only one column in the result set (for example, `SELECT FirstName FROM ...`.

With radiating left outer joins, all other tables are joined onto one central table. In the following example of a three-way radiating left outer join, `Emp` is the central table and all joins radiate from that table.

```
SELECT * FROM (Emp LEFT OUTER JOIN Dept ON Emp.DeptID = Dept.DeptID) LEFT OUTER JOIN Addr ON Emp.EmpID = Addr.EmpID
```

<table>
<thead>
<tr>
<th>Emp</th>
<th>Dept</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Last</td>
<td>Name</td>
</tr>
<tr>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Franky</td>
<td>Avalon</td>
<td>D103</td>
</tr>
<tr>
<td>Gordon</td>
<td>Lightfoot</td>
<td>D102</td>
</tr>
<tr>
<td>Lawrence</td>
<td>Welk</td>
<td>D101</td>
</tr>
<tr>
<td>Bruce</td>
<td>Cockburn</td>
<td>D102</td>
</tr>
</tbody>
</table>

In a chaining left outer join, one table is joined to another, and that table, in turn, is joined to another. The following example illustrates a three-way chaining left outer join:
SELECT * FROM (Emp LEFT OUTER JOIN Dept ON Emp.DeptID = Dept.DeptID) LEFT OUTER JOIN Loc ON Dept.LocID = Loc.LocID

Table 2-13 Three-way Chaining Left Outer Join

<table>
<thead>
<tr>
<th>Emp</th>
<th>Dept</th>
<th>Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td>Last Name</td>
<td>Dept ID</td>
</tr>
<tr>
<td>Franky</td>
<td>Avalon</td>
<td>D103</td>
</tr>
<tr>
<td>Gordon</td>
<td>Lightfoot</td>
<td>D102</td>
</tr>
<tr>
<td>Lawrence</td>
<td>Welk</td>
<td>D101</td>
</tr>
<tr>
<td>Bruce</td>
<td>Cockburn</td>
<td>D102</td>
</tr>
</tbody>
</table>

This join could also be expressed as:

SELECT * FROM Emp LEFT OUTER JOIN (Dept LEFT OUTER JOIN Loc ON Dept.LocID = Loc.LocID) ON Emp.DeptID = Dept.DeptID

We recommend the first syntax because it lends itself to both the radiating and chaining joins. This second syntax cannot be used for radiating joins because nested left outer join ON conditions cannot reference columns in tables outside their nesting. In other words, in the following query, the reference to Emp.EmpID is illegal:

SELECT * FROM Emp LEFT OUTER JOIN (Dept LEFT OUTER JOIN Addr ON Emp.EmpID = Addr.EmpID) ON Emp.DeptID = Dept.DeptID

The following example shows a three-way radiating left outer join, less optimized:

SELECT * FROM Emp E1 LEFT OUTER JOIN Dept ON E1.DeptID = Dept.DeptID, Emp E2 LEFT OUTER JOIN Addr ON E2.EmpID = Addr.EmpID WHERE E1.EmpID = E2.EmpID

Table 2-14 Three-way Radiating Left Outer Join, Less Optimized

<table>
<thead>
<tr>
<th>Emp</th>
<th>Dept</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name</td>
<td>Last Name</td>
<td>Dept ID</td>
</tr>
<tr>
<td>Franky</td>
<td>Avalon</td>
<td>D103</td>
</tr>
</tbody>
</table>
This query returns the same results as shown in Table 2-13, assuming there are no NULL values for EmpID in Emp and EmpID is a unique valued column. This query, however, is not optimized as well as the one show for Table 2-13 and can be much slower.

**See Also**

“SELECT” on page 2-118
LEAVE

Remarks
A LEAVE statement continues execution by leaving a block or loop statement. You can use LEAVE statements in the body of a stored procedure or a trigger.

Examples
The following example increments the variable vInteger by 1 until it reaches a value of 11, when the loop is ended with a LEAVE statement.

TestLoop:
LOOP
   IF (:vInteger > 10) THEN
      LEAVE TestLoop;
   END IF;
   SET :vInteger = :vInteger + 1;
END LOOP;

See Also
“IF” on page 2-88
“LOOP” on page 2-102
LOOP

Remarks
A LOOP statement repeats the execution of a block of statements. This statement is only allowed in stored procedures and triggers. Pervasive.SQL does not support post-conditional loops (REPEAT...UNTIL).

Examples
The following example increments the variable vInteger by 1 until it reaches a value of 11, when the loop is ended.

```
TestLoop:
LOOP
  IF (:vInteger > 10) THEN
    LEAVE TestLoop;
  END IF;
  SET :vInteger = :vInteger + 1;
END LOOP;
```

See Also
“CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
“IF” on page 2-88
NOT

Remarks
Using the NOT keyword with the EXISTS keyword allows you to test whether rows do not exist in the result of the subquery. For every row the outer query evaluates, Pervasive.SQL tests for the existence of a related row from the subquery. Pervasive.SQL excludes from the statement’s result table each row from the outer query that corresponds to a related row from the subquery.

Including the NOT keyword along with the IN operator allows you to test whether the result of the outer query is not included in the result of the subquery. The result table for the statement includes only rows the outer query returns that do not have a related row from the subquery.

Examples
The following statement returns a list of students who are not enrolled in any classes:

```
SELECT * FROM Person p WHERE NOT EXISTS
  (SELECT * FROM Student s WHERE s.id = p.id
   AND Cumulative_Hours = 0)
```

See Also
“SELECT” on page 2-118
OPEN

Remarks
The OPEN (cursor) statement opens a cursor. A cursor must be defined before it can be opened.
This statement is allowed only inside of a stored procedure or a trigger, since cursors and variables are only allowed inside of stored procedures and triggers.

Examples
The following example opens the declared cursor BTUCursor.

```
DECLARE BTUCursor CURSOR
    FOR SELECT Degree, Residency, Cost_Per_Credit
    FROM Tuition
    ORDER BY ID;
OPEN BTUCursor;
```

See Also
“CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
“DECLARE CURSOR” on page 2-65
PRIMARY KEY

Remarks

Include PRIMARY KEY in the ADD clause to add a primary key to a table definition.

Before adding the primary key, you must ensure that the columns in the primary key column list are defined as a unique index that does not include null values. If such an index does not exist, create one with the CREATE INDEX statement.

Because a table can have only one primary key, you cannot add a primary key to a table that already has a primary key defined. To change the primary key of a table, delete the existing key using a DROP clause in an ALTER TABLE statement and add the new primary key.

Note You must be logged in to the database using a database name before you can add a primary key or conduct any other referential integrity (RI) operation.

Include a PRIMARY KEY clause with the CREATE TABLE statement to define the key.

To define referential constraints on your database, you must include a PRIMARY KEY clause to specify the primary key on the parent table. The primary key can consist of one column or multiple columns but can only be defined on columns that are not null. The columns you specify must also appear in the column Definitions list of the CREATE TABLE statement.

You must define the columns that make up a primary key as a unique index that does not include null values. When you specify a primary key, Pervasive.SQL creates an index with the specified attributes on the defined group of columns.
Examples

The following statement defines a primary key on a table called Faculty. (The ID column is defined as a unique index that does not include null values.)

```
ALTER TABLE Faculty ADD PRIMARY KEY (ID)
```

See Also

“ALTER TABLE” on page 2-19

“CREATE TABLE” on page 2-50
PUBLIC

Remarks
You can include the PUBLIC keyword in the FROM clause to revoke the Create Table right from all the users to whom the right was not explicitly assigned.

Include a FROM clause to specify the group or user from whom you are revoking rights. You can specify a single name or a list of names, or you can include the PUBLIC keyword to revoke access rights from all users whose rights are not explicitly assigned.

Examples
To assign access rights to all users in the dictionary, include the PUBLIC keyword to grant the rights to the PUBLIC group, as in the following example:

```
GRANT SELECT ON Course TO PUBLIC
```

This statement assigns the Select right on the Course table to all users defined in the dictionary. If you later revoke the Select right from the PUBLIC group, only users who are granted the Select right explicitly can access the table.

The following statement includes the PUBLIC keyword to grant the Create Table right to all the users defined in the dictionary:

```
GRANT CREATETAB TO PUBLIC
```

See Also
“GRANT” on page 2-80

“REVOKE” on page 2-112
PRINT

Remarks
Use PRINT to print variable values or constants. The PRINT statement applies only to Windows-based platforms. It is ignored on other operating system platforms.

You can use PRINT only within stored procedures.

Examples
The following example prints the value of the variable :myvar.

```
PRINT(:myvar);
PRINT('MYVAR = ' + :myvar);
```

The following example prints a text string followed by a numeric value. You must convert a number value to a text string to print the value.

```
PRINT('Students enrolled in History 101: ' + convert(:int_val, SQL_CHAR));
```

See Also
“CREATE PROCEDURE” on page 2-42
RELEASE SAVEPOINT

Use the RELEASE SAVEPOINT statement to delete a savepoint.

Syntax

```
RELEASE SAVEPOINT savepoint-name
```

```
savepoint-name ::= user-defined-name
```

Remarks

RELEASE, ROLLBACK, and SAVEPOINT and are supported at the session level (outside of stored procedures) only if AUTOCOMMIT is off. Otherwise, RELEASE, ROLLBACK, and SAVEPOINT must be used within a stored procedure.

Any committed statements within a stored procedure are controlled by the outermost transaction of the calling ODBC application. This means that, depending on the AUTOCOMMIT mode specified on SQLSetConnectOption, calling the stored procedure externally from an ODBC application performs one of two actions. It either commits automatically (AUTOCOMMIT on, the default) or waits for you to call SQLTransact with SQL_COMMIT or SQL_ROLLBACK (when AUTOCOMMIT is set to off).

Examples

The following example sets a SAVEPOINT then checks a condition to determine whether to ROLLBACK or to RELEASE the SAVEPOINT.

```
CREATE PROCEDURE Enroll_student(IN :student ubigint, IN :classnum INTEGER);
BEGIN
    DECLARE :CurrentEnrollment INTEGER;
    DECLARE :MaxEnrollment INTEGER;
    SAVEPOINT SP1;
    INSERT INTO Enrolls VALUES (:student, :classnum, 0.0);
    SELECT COUNT(*) INTO :CurrentEnrollment FROM Enrolls WHERE class_id = :classnum;
    SELECT Max_size INTO :MaxEnrollment FROM Class WHERE ID = :classnum;
```
IF :CurrentEnrollment >= :MaxEnrollment
THEN
ROLLBACK TO SAVEPOINT SP1;
ELSE
RELEASE SAVEPOINT SP1;
END IF;
END;

Note that COUNT(expression) counts all non-NULL values for an expression across a predicate. COUNT(*) counts all values, including NULL values.

See Also
“CREATE PROCEDURE” on page 2-42
“ROLLBACK” on page 2-114
“SAVEPOINT” on page 2-115
RESTRICT

Remarks
If you specify RESTRICT, Pervasive.SQL enforces the DELETE RESTRICT rule. A user cannot delete a row in the parent table if a foreign key value refers to it.

If you do not specify a delete rule, Pervasive.SQL applies the RESTRICT rule by default.

See Also
“ALTER TABLE” on page 2-19
REVOKE

REVOKE deletes user IDs and removes permissions to specific users in a secured database.

**Syntax**

REVOKE CREATETAB FROM public-or-user-group-name [ , public-or-user-group-name ]...

REVOKE LOGIN FROM user-name [ , user-name ]...

REVOKE table-privilege ON table-name
    FROM user-or-group-name [ , user-or-group-name ]...

`table-privilege ::= ALL |
                   SELECT [ ( column-name [ , column-name ]... ) ] |
                   UPDATE [ ( column-name [ , column-name ]... ) ] |
                   INSERT [ ( column-name [ , column-name ]... ) ] |
                   DELETE |
                   ALTER |
                   REFERENCES`

`public-or-user-group-name ::= PUBLIC | user-group-name`

`user-group-name ::= user-name | group-name`

`group-name ::= user-defined-name`

`user-name ::= user-defined-name`

**Examples**

The following statement revokes all these privileges from dannyd for table Class.

REVOKE ALL ON Class FROM 'dannyd'

The following statement revokes all privileges from dannyd and travisk for table Class.

REVOKE ALL ON Class FROM 'dannyd', travisk

This statement revokes DELETE privileges from dannyd and travisk for table Class.

REVOKE DELETE ON Class FROM dannyd, travisk
SQL Grammar Elements

The following example revokes INSERT rights from keithv and miked for table Class.

REVOKE INSERT ON Class FROM keithv, miked

The following example revokes INSERT rights from keithv and brendanb for table Person and columns First_name and Last_name.

REVOKE INSERT(First_name, Last_name) ON Person FROM keithv, brendanb

The following statement revokes ALTER rights from dannyd from table Class.

REVOKE ALTER ON Class FROM dannyd

The following example revokes SELECT rights from dannyd and travisk on table Class.

REVOKE SELECT ON Class FROM dannyd, travisk

The following statement revokes SELECT rights from dannyd and travisk in table Person for columns First_name and Last_name.

REVOKE SELECT(First_name, Last_name) ON Person FROM dannyd, travisk

The following example revokes UPDATE rights from dannyd and travisk for table Person.

REVOKE UPDATE ON Person ON dannyd, travisk

See Also

“GRANT” on page 2-80
ROLLBACK

ROLLBACK returns the database to the state it was in before the current transaction began. This statement releases the locks acquired since the last SAVEPPOINT or START TRANSACTION.

Syntax

```
ROLLBACK [ WORK ] [ TO SAVEPPOINT savepoint-name ]
```

Remarks

ROLLBACK, SAVEPPOINT, and RELEASE are supported at the session level (outside of stored procedures) only if AUTOCOMMIT is off. Otherwise, ROLLBACK, SAVEPPOINT, and RELEASE must be used within a stored procedure.

Any committed statements within a stored procedure are controlled by the outermost transaction of the calling ODBC application. This means that, depending on the AUTOCOMMIT mode specified on SQLSetConnectOption, calling the stored procedure externally from an ODBC application performs one of two actions. It either commits automatically (AUTOCOMMIT on, the default) or waits for you to call SQLTransact with SQL_COMMIT or SQL_ROLLBACK (when AUTOCOMMIT is set to off).

In the case of nested transactions, ROLLBACK rolls back to the nearest START TRANSACTION. For example, if transactions are nested five levels, then five ROLLBACK statements are needed to undo all of the transactions. A transaction is either committed or rolled back, but not both. That is, you cannot roll back a committed transaction.

Examples

The following statement undoes the changes made to the database since the beginning of a transaction.

```
ROLLBACK WORK
```

The following statement undoes the changes made to the database since the last savepoint.

```
ROLLBACK TO SAVEPPOINT SP1
```

See Also

“COMMIT” on page 2-37

“RELEASE SAVEPPOINT” on page 2-109

“SAVEPPOINT” on page 2-115
SQL Grammar Elements

SAVEPOINT

SAVEPOINT defines a point in a transaction to which you can roll back.

Syntax

```
SAVEPOINT savepoint-name
```

savepoint-name ::= user-defined-name

Remarks

ROLLBACK, SAVEPOINT, and RELEASE are supported at the session level (outside of stored procedures) only if AUTOCOMMIT is off. Otherwise, ROLLBACK, SAVEPOINT, and RELEASE must be used within a stored procedure.

Any committed statements within a stored procedure are controlled by the outermost transaction of the calling ODBC application. This means that, depending on the AUTOCOMMIT mode specified on SQLSetConnectOption, calling the stored procedure externally from an ODBC application performs one of two actions. It either commits automatically (AUTOCOMMIT on, the default) or waits for you to call SQLTransact with SQL_COMMIT or SQL_ROLLBACK (when AUTOCOMMIT is set to off).

A SAVEPOINT applies only to the procedure in which it is defined. That is, you cannot reference a SAVEPOINT defined in another procedure.

Examples

The following example sets a SAVEPOINT then checks a condition to determine whether to ROLLBACK or to RELEASE the SAVEPOINT.

```
CREATE PROCEDURE Enroll_student( IN :student ubiquint, IN :classnum INTEGER);
BEGIN
   DECLARE :CurrentEnrollment INTEGER;
   DECLARE :MaxEnrollment INTEGER;
   SAVEPOINT SP1;
   INSERT INTO Enrolls VALUES (:student, :classnum, 0.0);
   SELECT COUNT(*) INTO :CurrentEnrollment FROM Enrolls WHERE class_id = :classnum;
   SELECT Max_size INTO :MaxEnrollment FROM Class WHERE ID = :classnum;
   IF :CurrentEnrollment >= :MaxEnrollment
```
THEN
ROLLBACK TO SAVEPOINT SP1;
ELSE
RELEASE SAVEPOINT SP1;
END IF;
END;

Note that COUNT(expression) counts all non-NULL values for an expression across a predicate. COUNT(*) counts all values, including NULL values.

See Also

“COMMIT” on page 2-37
“CREATE PROCEDURE” on page 2-42
“RELEASE SAVEPOINT” on page 2-109
“ROLLBACK” on page 2-114
SELECT (with into)

The SELECT (with INTO) statement allows you to select column values from a specified table to insert into variable names within stored procedures.

Syntax

```
SELECT [ ALL | DISTINCT ] select-list INTO variable-name [ ,
variable-name ]...
FROM table-reference [ , table-reference ]... [ WHERE search-condition ]
[ GROUP BY expression [ , expression ]...[ HAVING search-condition ] ]
```

Remarks

The SELECT with the INTO clause is only allowed within stored procedures.

Examples

The following example assigns into variables :x, :y the values of first_name an last_name in the Person table where first name is Bill.

```
SELECT first_name, last_name INTO :x, :y FROM person
WHERE first_name = 'Bill'
```

See Also

“CREATE PROCEDURE” on page 2-42
SELECT

Retrieves specified information from a database. A SELECT statement creates a temporary view.

**Syntax**

```
query-specification [ [ UNION [ ALL ] query-specification ]... 
[ ORDER BY order-by-expression [ , order-by-expression ]... ]

order-by-expression ::= expression [ CASE | COLLATE collation-name ] [ ASC | DESC ]

query-specification ::= ( query-specification )
  | SELECT [ ALL | DISTINCT ] select-list
    FROM table-reference [ , table-reference ]...
    [ WHERE search-condition ]
    [ GROUP BY expression [ , expression ]... 
    [ HAVING search-condition ] ]

select-list ::= * | select-item [ , select-item ]...

select-item ::= expression [ AS alias-name ] | table-name . *

table-reference ::= { OJ outer-join-definition }
  | table-name [ AS alias-name ]
  | join-definition
  | ( join-definition )

join-definition ::= table-reference [ INNER ] JOIN table-reference ON search-condition
  | table-reference CROSS JOIN table-reference
  | outer-join-definition

outer-join-definition ::= table-reference outer-join-type JOIN table-reference ON search-condition

outer-join-type ::= LEFT [ OUTER ] | RIGHT [ OUTER ] | FULL
  [ OUTER ]

search-condition ::= search-condition AND search-condition
  | search-condition OR search-condition
  | NOT search-condition
  | ( search-condition )
  | predicate
```
SQL Grammar Elements

**predicate** ::= expression [ NOT ] BETWEEN expression AND expression

| expression comparison-operator expression-or-subquery |
| expression [ NOT ] IN ( query-specification ) |
| expression [ NOT ] IN ( value [ , value ]... ) |
| expression [ NOT ] LIKE value |
| expression IS [ NOT ] NULL |
| expression comparison-operator ANY ( query-specification ) |
| expression comparison-operator ALL ( query-specification ) |
| expression comparison-operator SOME ( query-specification ) |
| EXISTS ( query-specification ) |

**comparison-operator** ::= < | > | <= | >= | = | <>

**expression-or-subquery** ::= expression | ( query-specification )

**value** ::= literal | USER | ?

**expression** ::= expression - expression

| expression + expression |
| expression * expression |
| expression / expression |
| ( expression ) |
| -expression |
| +expression |
| column-name |
| ? |
| literal |
| set-function |
| scalar-function |
| { fn scalar-function } |

**USER**

| IF ( search-condition , expression , expression ) |
| SQLSTATE |
| : user-defined-name |

@:IDENTITY
@:ROWCOUNT
@@IDENTITY
@@ROWCOUNT
set-function ::= COUNT (*)
   | COUNT ( [ DISTINCT | ALL ] expression )
   | SUM  ( [ DISTINCT | ALL ] expression )
   | AVG  ( [ DISTINCT | ALL ] expression )
   | MIN  ( [ DISTINCT | ALL ] expression )
   | MAX  ( [ DISTINCT | ALL ] expression )

scalar-function ::= see "Scalar Functions" on page 2-156

Remarks

In addition to supporting a GROUP BY on a column-list, as specified in the Microsoft ODBC Programmer’s Reference, Pervasive ODBC Engine Interface has extended the syntax to support a GROUP BY on an expression-list or on any expression in a GROUP BY expression-list. See “GROUP BY” on page 2-85 for more information on GROUP BY extensions. HAVING is not supported without GROUP BY.

Result sets and stored views generated by executing SELECT statements with any of the following characteristics are read-only (they cannot be updated). That is, a positioned UPDATE, a positioned DELETE and an SQLSetPos call to add, alter or delete data is no allowed on the result set or stored view:

- SQL_CONCUR_READ_ONLY was specified as the SQL_CONCURRENCY type via SQLSetStmtOption
- The selection-list contains an aggregate:
  SELECT SUM(c1) FROM t1
- The selection-list specifies DISTINCT:
  SELECT DISTINCT c1 FROM t1
- The view contains a GROUP BY clause:
  SELECT SUM(c1), c2 FROM t1 GROUP BY c2
- The view is a join (references multiple tables):
  SELECT * FROM t1, t2
- The view uses the UNION operator and UNION ALL is not specified or all SELECT statements do not reference the same table:
  SELECT c1 FROM t1 UNION SELECT c1 FROM t1
  SELECT c1 FROM t1 UNION ALL SELECT c1 FROM t2
  - Note that stored views do not allow the UNION operator.
The view contains a subquery that references a table other than the table in the outer query:

```
SELECT c1 FROM t1 WHERE c1 IN (SELECT c1 FROM t2)
```

### Examples

This simple SELECT statement retrieves all the data from the Faculty table.

```
SELECT * FROM Faculty
```

This statement retrieves the data from the person and the faculty table where the id column in the person table is the same as the id column in the faculty table.

```
SELECT Person.id, Faculty.salary FROM Person, Faculty
WHERE Person.id = Faculty.id
```

The following example retrieves student_id and sum of the amount_paid where it is greater than or equal to 100 from the billing table. It then groups the records by student_id.

```
SELECT Student_ID, SUM(Amount_Paid)
FROM Billing
GROUP BY Student_ID
HAVING SUM(Amount_Paid) >=100.00
```

If the expression is a positive integer literal, then that literal is interpreted as the number of the column in the result set and ordering is done on that column. No ordering is allowed on set functions or an expression that contains a set function.

### Subqueries

The following types of subqueries are supported: comparison, quantified, in, exists, and correlated. ORDER BY clauses are not allowed in a subquery clause.

Correlated subquery predicates in the HAVING clause which contain references to grouped columns are not supported.

### approximate-numeric-literal

#### Examples

```
SELECT * FROM results WHERE quotient =-4.5E-2
```

```
INSERT INTO results (quotient) VALUES (+5E7)
```
between-predicate

Remarks
The syntax expression1 BETWEEN expression2 and expression3 returns TRUE if expression1 >= expression2 and expression1 <= expression3. FALSE is returned if expression1 >= expression3, or is expression1 <= expression2.

Expression2 and expression3 may be dynamic parameters (for example, SELECT * FROM emp WHERE emp_id BETWEEN ? AND ?)

Examples
The next example retrieves the first names from the person table whose ID fall between 10000 and 20000.

SELECT First_name FROM Person WHERE ID BETWEEN 10000 AND 20000

correlation-name

Remarks
Both table and column correlation names are supported.

Examples
The following example selects data from both the person table and the faculty table using the aliases T1 and T2 to differentiate between the two tables.

SELECT * FROM Person T1, Faculty T2 WHERE T1.id = T2.id

The correlation name for a table name can also be specified in using the FROM clause, as seen in the following example.

SELECT a.Name, b.Capacity FROM Class a, Room b
WHERE a.Room_Number = b.Number

exact-numeric-literal

Examples
SELECT car_num, price FROM cars WHERE car_num = 49042 AND price=49999.99

in-predicate

Examples
This selects the records from table Person table where the first names are Bill and Roosevelt.

SELECT * FROM Person WHERE First_name IN ('Roosevelt', 'Bill')

2-122
set-function

**Examples**

The following example selects the minimum salary from the Faculty table.

```
SELECT MIN(salary) FROM Faculty
```

MIN(expression), MAX(expression), SUM(expression), AVG(expression), COUNT(*), and COUNT(expression) are supported.

```
COUNT(expression) counts all non-NULL values for an expression across a predicate. COUNT(*) counts all values, including NULL values.
```

The following example counts all the rows in q where a+b does not equal NULL.

```
SELECT COUNT(a+b) FROM q
```

date-literal

**Remarks**

Date constants may be expressed in SQL statements as a character string or embedded in a vendor string. SQL_CHAR and the vendor string representation are treated as a value of type SQL_DATE. This becomes important when conversions are attempted.

The Pervasive ODBC Engine Interface partially supports extended SQL grammar, as outlined in this function.

**Examples**

The next two statements return all the classes whose start date is after 1995-06-05.

```
SELECT * FROM Class WHERE Start_Date > '1995-06-05'

SELECT * FROM Class WHERE Start_Date > {d '1995-06-05'}
```

The Pervasive ODBC Engine Interface supports the following date literal format: 'YYYY-MM-DD'.

Dates may be in the range of year 0 to 9999.

time-literal

**Examples**

The following two statements retrieve records from the class table where the start time for the classes is 14:00:00.
SELECT * FROM Class WHERE Start_time = '14:00:00'

SELECT * FROM Class WHERE Start_time = {t '14:00:00'}

The Pervasive ODBC Engine Interface supports the following time literal format: 'HH:MM:SS'.

Time constants may be expressed in SQL statements as a character string or embedded in a vendor string. Character string representation is treated as a string of type SQL_CHAR and the vendor string representation as a value of type SQL_TIME.

The Pervasive ODBC Engine Interface partially supports extended SQL grammar, as outlined in this function.

**timestamp-literal**

**Remarks**

Timestamp constants may be expressed in SQL statements as a character string or embedded in a vendor string. The Pervasive ODBC Engine Interface treats the character string representation as a string of type SQL_CHAR and the vendor string representation as a value of type SQL_TIMESTAMP. The Pervasive ODBC Engine Interface partially supports extended SQL grammar, as outlined in this function.

**Examples**

The next two statements retrieve records from the Billing table where the start day and time for the log is 1996-03-28 at 17:40:49.

SELECT * FROM Billing WHERE log = '1996-03-28 17:40:49'

SELECT * FROM Billing WHERE log = {ts '1996-03-28 17:40:49'}

The Pervasive ODBC Engine Interface supports the following time literal format: 'YYYY-MM-DD HH:MM:SS'
date arithmetic

Examples

SELECT * FROM person P, Class C WHERE p.Date_Of_Birth < '1973-09-05' AND c.Start_date > (d '1995-05-08') + 30

The Pervasive ODBC Engine Interface supports adding or subtracting an integer from a date where the integer is the number of days to add or subtract, and the date is embedded in a vendor string. (This is equivalent to executing a convert on the date).

The Pervasive ODBC Engine Interface also supports subtracting one date from another to yield a number of days.

IF

Remarks

The IF system scalar function provides conditional execution based on the truth value of a condition

Examples

This expression prints the column header as “Prime1” and amount owed as 2000 where the value of the column amount_owed is 2000 or it prints a 0 if the value of the amount_owed column is not equal to 2000.

SELECT Student_ID, Amount_Owed,
       IF (Amount_Owed = 2000, Amount_Owed, Convert(0, SQL_DECIMAL)) "Prime1"
FROM Billing

From table Class, the following example prints the value in the Section column if the section is equal to 001, else it prints “xxx” under column header Prime1

Under column header Prime2, it prints the value in the Section column if the value of the section column is equal to 002, or else it prints “yyy.”

SELECT ID, Name, Section,
       IF (Section = '001', Section, 'xxx') "Prime1",
       IF (Section = '002', Section, 'yyy') "Prime2"
FROM Class

You can combine header Prime1 and header Prime2 by using nested IF functions. Under column header Prime, the following query prints the value of the Section column if the value of the Section column is equal to 001 or 002. Otherwise, it print “xxx.”
SELECT ID, Name, Section,
    IF (Section = '001', Section, IF(Section = '002',
      Section, 'xxx')) Prime
FROM Class

left outer join

Remarks
The following example shows how to access the “Person” and “Student” tables from the DEMODATA database to obtain the Last Name, First Initial of the First Name and GPA of students. With the LEFT OUTER JOIN, all rows in the “Person” table are fetched (the table to the left of LEFT OUTER JOIN). Since not all people have GPAs, some of the columns have NULL values for the results. This is how outer join works, returning non-matching rows from either table.

Examples
SELECT Last_Name,Left(First_Name,1) AS First_Initial,Cumulative_GPA AS GPA FROM "Person"
  LEFT OUTER JOIN "Student" ON Person.ID=Student.ID
ORDER BY Cumulative_GPA DESC, Last_Name

Assume that you want to know everyone with perfectly rounded GPAs and have them all ordered by the length of their last name. Using the MOD statement and the LENGTH scalar function, you can achieve this by adding the following to the query:

WHERE MOD(Cumulative_GPA,1)=0 ORDER BY LENGTH(Last_Name)

right outer join

Remarks
The difference between LEFT and RIGHT OUTER JOIN is that all non matching rows show up for the table defined to the right of RIGHT OUTER JOIN. Change the query for LEFT OUTER JOIN to include a RIGHT OUTER JOIN instead. The difference is that the all non-matching rows from the right table, in this case “Student,” show up even if no GPA is present. However, since all rows in the “Student” table have GPA’s, all rows are fetched.
SQL Grammar Elements

Examples

SELECT Last_Name, Left(First_Name, 1) AS First_Initial, Cumulative_GPA AS GPA FROM "Person"
  RIGHT OUTER JOIN "Student" ON Person.ID=Student.ID
  ORDER BY Cumulative_GPA DESC, Last_Name

Cartesian join

Remarks

A Cartesian join is the matrix of all possible combinations of the rows from each of the tables. The number of rows in the Cartesian product equals the number of rows in the first table times the number of rows in the second table.

Examples

Assume you have the following tables in your database:

Table 2-15 Addr Table

<table>
<thead>
<tr>
<th>EmplID</th>
<th>Street</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>101 Mem Lane</td>
</tr>
<tr>
<td>E2</td>
<td>14 Young St.</td>
</tr>
</tbody>
</table>

Table 2-16 Loc Table

<table>
<thead>
<tr>
<th>LocID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>PlanetX</td>
</tr>
<tr>
<td>L2</td>
<td>PlanetY</td>
</tr>
</tbody>
</table>

The following performs a Cartesian JOIN on these tables:

SELECT * FROM Addr, Loc
This results in the following:

### Table 2-17 SELECT Statement with Cartesian JOIN

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Street</th>
<th>LocID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>101 Mem Lane</td>
<td>L1</td>
<td>PlanetX</td>
</tr>
<tr>
<td>E1</td>
<td>101 Mem Lane</td>
<td>L2</td>
<td>PlanetY</td>
</tr>
<tr>
<td>E2</td>
<td>14 Young St</td>
<td>L1</td>
<td>PlanetX</td>
</tr>
<tr>
<td>E2</td>
<td>14 Young St</td>
<td>L2</td>
<td>PlanetY</td>
</tr>
</tbody>
</table>

### DISTINCT

#### Remarks
You can use **DISTINCT** with SUM, AVG, COUNT, MIN, and MAX (but it does not change results with MIN and MAX). **DISTINCT** eliminates duplicate values before calculating the sum, average or count.

#### Examples
Suppose you want to know the salaries for different departments including the minimum, maximum and salary, and you want to remove duplicate salaries. The following statement would do this, excluding the computer science department:

```
SELECT dept_name, MIN(salary), MAX(salary), AVG(DISTINCT salary) FROM faculty WHERE dept_name<>'computer_science' GROUP BY dept_name
```

If you wanted to include duplicate salaries, you would use:

```
SELECT dept_name, MIN(salary), MAX(salary), AVG(salary) FROM faculty WHERE dept_name<>'computer_science' GROUP BY dept_name
```

#### See Also
“Global Variables” on page 2-13
“JOIN” on page 2-94
SET SECURITY

The SET SECURITY statement allows you to enable and disable security for the database to which you are currently logged in.

Syntax

```
SET SECURITY = password
SET SECURITY = NULL
```

Examples

The following example sets the password as 'password'.

```
SET SECURITY = 'password'
```

The following example sets the password as 123456.

```
SET SECURITY = '123456'
```

The following example disables security.

```
SET SECURITY = NULL
```

Remarks

You must be logged in as Master to set security. You can then assign a password by using the SET SECURITY statement. There is no password required to log in as Master initially.

When using SET SECURITY, user name and password are case sensitive.

Only one Master user connection to the database is allowed to set security. You can set security from the Pervasive Control Center (PCC).

Note

The SET SECURITY statement cannot be executed within the SQL Data Manager. An error results if you try. For a database with no security, the SQL Data Manager locks the dictionary files, which prevents you from setting the password. For a secure database, the SQL Data Manager opens a second connection to the database files, which prevents you from disabling security.

See Also

“GRANT” on page 2-80

“REVOKE” on page 2-112
SET TRUENULLCREATE

The SET TRUENULLCREATE statement turns on or off true NULLs when you create new tables.

Syntax

\[
\text{SET TRUENULLCREATE} = \langle \text{ON} \mid \text{OFF} \rangle
\]

Remarks

Pervasive.SQL 2000i allows you to set the default format for creation of tables with regard to NULL support. Normally, the product creates new tables using the true NULL data record format, which adds a NULL indicator byte to the beginning of every field. By turning off this engine setting using an SQL statement, you can create new tables that use the legacy NULL data record format that was used in Pervasive.SQL 7.

The creation mode remains in effect until it is changed by issuing the statement again, or until the connection is disconnected. Because this setting is maintained on a per-connection basis, separate database connections can maintain different creation modes, even within the same application. Every connection starts with the setting in default mode, where new tables are created with true NULL support.

This feature does not affect existing tables or available column data types. All tables are created using Pervasive.SQL data types. For example, old data types such as NOTE or LV AR are not available for use regardless of which type of NULL support is selected.

Also see the discussion about nullable data types under “INSERT” on page 2-90.

This setting can only be toggled using SQL, it cannot be set using the Pervasive Control Center (PCC).

Examples

To toggle the setting and specify that new tables should be created with legacy NULL support, use this SQL statement:

\[
\text{SET TRUENULLCREATE=OFF}
\]

To toggle the setting and return the engine to the default, which is table creation with true NULL support, use this SQL statement:

\[
\text{SET TRUENULLCREATE=ON}
\]
**SET VARIABLE**

SET assigns a value to a declared variable.

**Syntax**

```
SET variable-name = proc-expr
```

**Remarks**

You must declare variables before you can set them. SET is allowed only in stored procedures and triggers.

**Examples**

The following examples assigns a value of 10 to var1.

```
SET: var1 = 10;
```

**See Also**

“CREATE PROCEDURE” on page 2-42

“DECLARE” on page 2-64
SIGNAL

Remarks

The SIGNAL statement allows you to signal an exception condition or a completion condition other than successful completion.

Signalling an SQLSTATE value causes SQLSTATE to be set to a specific value. This value is then returned to the user, or made available to the calling procedure (through the SQLSTATE value). This value is available to the application calling the procedure. SIGNAL is available only inside a stored procedure.

Examples

The following example prints the initial SQLSTATE value “00000,” then prints “SQLSTATE exception caught” after the signal is raised. The final SQLSTATE value printed is “W9001.”

```
CREATE PROCEDURE GenerateSignal();
BEGIN
    SIGNAL 'W9001';
END;
```

```
CREATE PROCEDURE TestSignal() WITH DEFAULT HANDLER;
BEGIN
    PRINT SQLSTATE;
    CALL GenerateSignal();
    IF SQLSTATE < '00000' THEN
        PRINT 'SQLSTATE exception caught';
    END IF;
    PRINT SQLSTATE;
END;
```

See Also

“CREATE PROCEDURE” on page 2-42
SQLSTATE

Remarks

The SQLSTATE value corresponds to a success, warning, or exception condition. The complete list of SQLSTATE values defined by ODBC can be found in the Microsoft ODBC SDK documentation.

When a handler executes, the statements within it affect the SQLSTATE value in the same way as statements in the main body of the compound statement. However, a handler that is intended to take specific action for a specific condition can optionally leave that condition unaffected, by re-assigning that condition at its conclusion. This does not cause the handler to be invoked again; that would cause a loop. Instead, Pervasive.SQL treats the exception condition as an unhandled exception condition, and execution stops.

See Also

“CREATE PROCEDURE” on page 2-42
“SELECT” on page 2-118
“SIGNAL” on page 2-132
**START TRANSACTION**

START TRANSACTION signals the start of a logical transaction. START TRANSACTION must always be paired with a COMMIT or a ROLLBACK.

**Syntax**

```
START TRANSACTION
```

**Remarks**

START TRANSACTION is supported only within stored procedures. You cannot use START TRANSACTION within the SQL Data Manager. (The SQL Data Manager sets AUTOCOMMIT to “on.”)

**Examples**

The following example, within a stored procedure, begins a transaction which updates the Amount_Owed column in the Billing table. This work is committed; another transaction updates the Amount_Paid column and sets it to zero. The final COMMIT WORK statement ends the second transaction.

Statements are delimited with a semi-colon inside stored procedures and triggers.

```
START TRANSACTION;
UPDATE Billing B
    SET Amount_Owed = Amount_Owed - Amount_Paid
    WHERE Student_ID IN
        (SELECT DISTINCT E.Student_ID
         FROM Enrolls E, Billing B
         WHERE E.Student_ID = B.Student_ID);
COMMIT WORK;

START TRANSACTION;
UPDATE Billing B
    SET Amount_Paid = 0
    WHERE Student_ID IN
        (SELECT DISTINCT E.Student_ID
         FROM Enrolls E, Billing B
         WHERE E.Student_ID = B.Student_ID);
COMMIT WORK;
```
See Also

“COMMIT” on page 2-37
“CREATE PROCEDURE” on page 2-42
“ROLLBACK” on page 2-114
UNION

Remarks

SELECT statements that use UNION or UNION ALL allow you to obtain a single result table from multiple SELECT queries. UNION queries are suitable for combining similar information contained in more than one data source.

UNION eliminates duplicate rows. UNION ALL preserves duplicate rows. Using the UNION ALL option is recommended unless you require duplicate rows to be removed.

With UNION, the Pervasive.SQL Engine orders the entire result set which, for large tables, can take several minutes. UNION ALL eliminates the need for the sort.

The Pervasive.SQL Engine does not support LONGVARBINARY columns in UNION statements. LONGVARCHAR is limited to 65500 bytes in UNION statements. The operator UNION cannot be applied to any SQL statement that references one or more views.

The two query specifications involved in a union must be compatible. Each query must have the same number of columns and the columns must be of compatible data types.

Examples

The following example lists the ID numbers of each student whose last name begins with 'M' or who has a 4.0 grade point average. The result table does not include duplicate rows.

```
SELECT Person.ID FROM Person WHERE Last_name LIKE 'M%'
UNION SELECT Student.ID FROM Student WHERE Cumulative_GPA = 4.0
```

The next example lists the column id in the person table and the faculty table including duplicate rows.

```
SELECT person.id FROM person UNION ALL SELECT faculty.id from faculty
```

The next example lists the ID numbers of each student whose last name begins with 'M' or who has a 4.0 grade point average. The result table does not include duplicate rows and orders the result set by the first column.

```
SELECT Person.ID FROM Person WHERE Last_name LIKE 'M'
UNION SELECT Student.ID FROM Student WHERE Cumulative_GPA = 4.0 ORDER BY 1
```
It is common to use the NULL scalar function to allow a UNION select list to have a different number of entries than the parent select list. To do this, you must use the CONVERT function to force the NULL to the correct type.

```
CREATE TABLE t1 (c1 INTEGER, c2 INTEGER)
    INSERT INTO t1 VALUES (1,1)
CREATE TABLE t2 (c1 INTEGER)
    INSERT INTO t2 VALUES (2)
SELECT c1, c2 FROM t1
UNION SELECT c1, CONVERT(NULL(),sql_integer)FROM t2
```

See Also

“SELECT” on page 2-118
UNIQUE

Remarks

To specify that the index not allow duplicate values, include the UNIQUE keyword. If the column or columns that make up the index contains duplicate values when you execute the CREATE INDEX statement with the UNIQUE keyword, Pervasive.SQL returns Status Code 5 and does not create the index.

Note You should not include the UNIQUE keyword in the list of index attributes following the column name you specify; the preferred syntax is CREATE UNIQUE INDEX.

See Also

“ALTER TABLE” on page 2-19
“CREATE INDEX” on page 2-40
“CREATE TABLE” on page 2-50
UPDATE

The UPDATE statement allows you to modify column values in a database.

Syntax

```
UPDATE table-name [ alias-name ]
    SET column-name = < expression | subquery >
    [ , column-name = < expression | subquery > ] ...
    [ WHERE search-condition ]
```

Remarks

INSERT, UPDATE, and DELETE statements behave in an atomic manner. That is, if an insert, update, or delete of more than one row fails, then all insertions, updates, or deletes of previous rows by the same statement are rolled back.

In the SET clause of an UPDATE statement, you may specify a subquery. This feature allows you to update information in a table based on information in another table or another part of the same table.

The UPDATE statement can update only a single table at a time. UPDATE can relate to other tables via a subquery in the SET clause. This can be a correlated subquery that depends in part on the contents of the table being updated, or it can be a non-correlated subquery that depends only on another table.

Correlated Subquery

```
UPDATE T1 SET T1.C2 = (SELECT T2.C2 FROM T2 WHERE T2.C1 = T1.C1)
```

Non-correlated Subquery

```
UPDATE T1 SET T1.C2 = (SELECT SUM(T2.C2) FROM T2 WHERE T2.C1 = 10)
```

The same logic is used to process pure SELECT statements and subqueries, so the subquery can consist of any valid SELECT statement. There are no special rules for subqueries.

If SELECT within an UPDATE returns no rows, then the UPDATE inserts NULL. If the given column(s) is/are not nullable, then the UPDATE fails. If select returns more than one row, then UPDATE fails.
An `UPDATE` statement does not allow the use of join tables in the statement. Instead, use a correlated subquery in the `SET` clause as follows:

```
UPDATE T1 SET T1.C2 = (SELECT T2.C2 FROM T2 WHERE T2.C1 = T1.C1)
```

All data types for data created prior to Pervasive.SQL 2000 (legacy data) report back as nullable. This means that you can `UPDATE` NULL into any legacy column type without pseudo-NULL conversion. The following data types are treated as pseudo-NULL by default:

<table>
<thead>
<tr>
<th>Date</th>
<th>Decimal</th>
<th>Money</th>
<th>Numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumericSA</td>
<td>NumericST</td>
<td>Timestamp</td>
<td></td>
</tr>
</tbody>
</table>

(Normally, when you convert a legacy column to pseudo-NULL, you lose one of the binary values, forfeiting it so that you can query the column for NULL. These data types, however, because of their design, have a different, unique internal value for NULL in addition to their normal data range. With these data types, no binary values are lost if they are converted to NULL so there is no harm considering them as pseudo-NULL by default.)

The rest of the data types are considered “legacy nullable,” meaning that NULL may be updated into them. When values are queried, however, the non-NULL binary equivalent is returned. This same binary equivalent must be used in `WHERE` clauses to retrieve specific values.

The binary equivalents are:

- 0 for Binary types
- Empty string from string and BLOB types (legacy types LV AR and NOTE)

**Examples**

The following examples updates the record in the faculty table and sets salary as 95000 for ID 103657107.

```
UPDATE Faculty SET salary = 95000.00 WHERE ID = 103657107
```
The following example changes the credit hours for Economics 305 in the course table from 3 to 4.

```
UPDATE Course SET Credit_Hours = 4 WHERE Name = 'ECO 305'
```

The following example updates the address for a person in the Person table:

```
UPDATE Person p
    SET p.Street = '123 Lamar',
        p.zip = '78758',
        p.phone = 5123334444
    WHERE p.ID = 131542520
```

Subquery Example A

Two tables are created and rows are inserted. The first table, t5, is updated with a column value from the second table, t6, in each row where table t5 has the value 2 for column c1. Because there is more than one row in table t6 containing a value of 3 for column c2, the first UPDATE fails because more than one row is returned by the subquery. This result occurs even though the result value is the same in both cases. As shown in the second UPDATE, using the DISTINCT keyword in the subquery eliminates the duplicate results and allows the statement to succeed.

```
CREATE TABLE t5 (c1 INT, c2 INT)
CREATE TABLE t6 (c1 INT, c2 INT)
INSERT INTO t5(c1, c2) VALUES (1,3)
INSERT INTO t5(c1, c2) VALUES (2,4)
INSERT INTO t6(c1, c2) VALUES (2,3)
INSERT INTO t6(c1, c2) VALUES (1,2)
INSERT INTO t6(c1, c2) VALUES (3,3)
SELECT * FROM t5
```

Results:

```
c1   c2
---   ---
1     3
2     4
```

```
UPDATE t5 SET t5.c1=(SELECT c2 FROM t6 WHERE c2=3) WHERE t5.c1=2  — Note that the query fails
```

```
UPDATE t5 SET t5.c1=(SELECT DISTINCT c2 FROM t6 WHERE c2=3) WHERE t5.c1=2  — Note that the query succeeds
```
Subquery Example B

Two tables are created and a variety of valid syntax examples are demonstrated. Note the cases where UPDATE fails because the subquery returns more than one row. Also note that UPDATE succeeds and NULL is inserted if the subquery returns no rows (where NULL values are allowed).

```
CREATE TABLE T1 (C1 INT, C2 INT)
CREATE TABLE T2 (C1 INT, C2 INT)

INSERT INTO T1 VALUES (1, 0)
INSERT INTO T1 VALUES (2, 0)
INSERT INTO T1 VALUES (3, 0)
INSERT INTO T2 VALUES (1, 100)
INSERT INTO T2 VALUES (2, 200)

UPDATE T1 SET T1.C2 = (SELECT SUM(T2.C2) FROM T2)
UPDATE T1 SET T1.C2 = 0
UPDATE T1 SET T1.C2 = (SELECT T2.C2 FROM T2 WHERE T2.C1 = T1.C1)
UPDATE T1 SET T1.C2 = @@IDENTITY
UPDATE T1 SET T1.C2 = @@ROWCOUNT
UPDATE T1 SET T1.C2 = (SELECT @@IDENTITY)
UPDATE T1 SET T1.C2 = (SELECT @@ROWCOUNT)

UPDATE T1 SET T1.C2 = (SELECT T2.C2 FROM T2) — update fails
INSERT INTO T2 VALUES (1, 150)
INSERT INTO T2 VALUES (2, 250)
UPDATE T1 SET T1.C2 = (SELECT T2.C2 FROM T2 WHERE T2.C1 = T1.C1) — update fails
UPDATE T1 SET T1.C2 = (SELECT T2.C2 FROM T2 WHERE T2.C1 = 5) — Note that the update succeeds, NULL is inserted for all rows of T1.C2
UPDATE T1 SET T1.C2 = (SELECT SUM(T2.C2) FROM T2 WHERE T2.C1 = T1.C1)
```
<table>
<thead>
<tr>
<th>See Also</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“ALTER TABLE”</td>
<td>on page 2-19</td>
</tr>
<tr>
<td>“CREATE PROCEDURE”</td>
<td>on page 2-42</td>
</tr>
<tr>
<td>“CREATE TRIGGER”</td>
<td>on page 2-59</td>
</tr>
<tr>
<td>“GRANT”</td>
<td>on page 2-80</td>
</tr>
</tbody>
</table>
UPDATE (positioned)

The positioned UPDATE statement updates the current row of a rowset associated with an SQL cursor.

Syntax

```
UPDATE [ table-name ] SET column-name = proc-expr [ , column-name = proc-expr ]...
WHERE CURRENT OF cursor-name
```

Remarks

This statement is allowed in stored procedures, triggers, and at the session level.

Note

Even though positioned UPDATE is allowed at the session level, the DECLARE CURSOR statement is not. Use the SQLGetCursorName() API to obtain the cursor name of the active result set.

The table-name may be specified in the positioned UPDATE statement only when used at the session level. Table-name cannot be specified with a stored procedure or trigger.

Examples

The following sequence of statements provide the setting for the positioned UPDATE statement. The required statements for a positioned UPDATE are DECLARE CURSOR, OPEN CURSOR, and FETCH FROM cursorname.

The positioned UPDATE statement in this example updates the name of the course HIS 305 to HIS 306.

```
CREATE PROCEDURE UpdateClass();
BEGIN
  DECLARE :CourseName CHAR(7);
  DECLARE :OldName CHAR(7);
  DECLARE c1 CURSOR FOR SELECT name FROM course WHERE name = :CourseName;
  OPEN c1;
  SET :CourseName = 'HIS 305';
  FETCH NEXT FROM c1 INTO :OldName;
  UPDATE SET name = 'HIS 306' WHERE CURRENT OF c1;
END;
```
See Also

“CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
WHILE

Use a WHILE statement is used to control flow. It allows code to be executed repeatedly as long as a certain condition is true.

Syntax

```
[ label-name : ] WHILE proc-search-condition
   DO [ proc-stmt ; proc-stmt ] ... END WHILE [ label-name ]
```

Remarks

A WHILE statement can have a beginning label (the statement is referred to as a labeled WHILE statement).

Examples

The following example increments the variable vInteger by 1 until it reaches a value of 10, when the loop ends.

```
WHILE (:vInteger < 10) DO
   SET :vInteger = vInteger + 1;
END WHILE
```

See Also

“CREATE PROCEDURE” on page 2-42
“CREATE TRIGGER” on page 2-59
Grammar Element Definitions

The following is an alphabetical summary of the element definitions used in the grammar syntax section:

- `alter-option-list ::= alter-option
  | (alter-option [, alter-option] . . .)
- `alter-option ::= ADD [ COLUMN ] column-definition
  | ADD table-constraint-definition
  | DROP [ COLUMN ] column-name
  | DROP CONSTRAINT constraint-name
  | DROP PRIMARY KEY
- `as-or-semicolon ::= AS | ;
- `before-or-after ::= BEFORE | AFTER
- `call-arguments ::= positional-argument [ , positional-argument ] . . .
- `col-constraint ::= NOT NULL
  | UNIQUE
  | PRIMARY KEY
  | REFERENCES table-name [ ( column-name ) ] [ referential-actions ]
- `collation-name ::= 'string' | user-defined-name
- `column-constraint ::= [ CONSTRAINT constraint-name ] col-constraint
- `column-definition ::= column-name data-type [ DEFAULT default-value ] [ column-constraint [ column-constraint ] . . . [ CASE | COLLATE collation-name ]
- `column-name ::= user-defined-name
- `commit-statement ::= see COMMIT statement
- `comparison-operator ::= < | > | <= | >= | = | <>
- `constraint-name ::= user-defined-name
- `correlation-name ::= user-defined-name
- `cursor-name ::= user-defined-name
- `data-type ::= data-type-name [ (precision [ , scale ] ) ]
ODBC Engine Reference

\[
data-type-name ::= \text{see list in Appendix A}
\]

\[
default-value ::= \text{literal}
\]

\[
expression ::= expression \text{ - expression}
| \text{expression + expression}
| \text{expression * expression}
| \text{expression / expression}
| ( \text{expression} )
| -expression
| +expression
| \text{column-name}
| ?
| \text{literal}
| \text{set-function}
| \text{scalar-function}
| \{ \text{fn scalar-function} \}
| \text{USER}
| \text{IF} ( \text{search-condition}, \text{expression}, \text{expression} )
| \text{SQLSTATE}
| : \text{user-defined-name}
\]

\[
expression-or-subquery ::= \text{expression} | ( \text{query-specification} )
\]

\[
fetch-orientation ::= | \text{NEXT}
\]

\[
group-name ::= \text{user-defined-name}
\]

\[
index-definition ::= ( \text{index-segment-definition} [ , \text{index-segment-definition} ] . . . )
\]

\[
index-segment-definition ::= \text{column-name} [ \text{ASC} | \text{DESC} ]
\]

\[
index-name ::= \text{user-defined-name}
\]

\[
ins-upd-del ::= \text{INSERT} | \text{UPDATE} | \text{DELETE}
\]

\[
insert-values ::= \text{values-clause}
| \text{query-specification}
\]
SQL Grammar Elements

join-definition ::= table-reference [ INNER ] JOIN table-reference ON search-condition
   | table-reference CROSS JOIN table-reference
   | outer-join-definition

label-name ::= user-defined-name

literal ::= 'string'
   | number
   | { d 'date-literal' }
   | { t 'time-literal' }
   | { ts 'timestamp-literal' }

order-by-expression ::= expression [ CASE | COLLATE collation-name ] [ ASC | DESC ]

outer-join-definition ::= table-reference outer-join-type JOIN table-reference
ON search-condition

outer-join-type ::= LEFT [ OUTER ] | RIGHT [ OUTER ] | FULL [ OUTER ]

parameter ::= parameter-type-name data-type [ DEFAULT proc-expr ] =
proc-expr

| SQLSTATE

parameter-type-name ::= parameter-name
   | parameter-type parameter-name
   | parameter-name parameter-type

parameter-type ::= IN | OUT | INOUT | IN_OUT

parameter-name ::= [ ] user-defined-name

password ::= user-defined-name | 'string'

positional-argument ::= expression

precision ::= integer
predicate ::= expression [ NOT ] BETWEEN expression AND expression
  | expression comparison-operator expression-or-subquery
  | expression [ NOT ] IN ( query-specification )
  | expression [ NOT ] IN ( value [ , value ]... )
  | expression [ NOT ] LIKE value
  | expression IS [ NOT ] NULL
  | expression comparison-operator ANY ( query-specification )
  | expression comparison-operator ALL ( query-specification )
  | expression comparison-operator SOME ( query-specification )
  | EXISTS ( query-specification )

proc-expr ::= same as normal expression but does not allow IF expression, or ODBC-style scalar functions

proc-search-condition ::= same as normal search-condition, but does not allow any expression that includes a subquery.

proc-stmt ::= [ label-name : ] BEGIN [ ATOMIC ] [ proc-stmt [ ; proc-stmt ]... ] END [ label-name ]
  | CALL procedure-name ( proc-expr [ , proc-expr ]... )
  | CLOSE cursor-name
  | DECLARE cursor-name CURSOR FOR select-statement [ FOR UPDATE | FOR READ ONLY ]
  | DECLARE variable-name data-type [ DEFAULT proc-expr | = proc-expr ]
  | DELETE WHERE CURRENT OF cursor-name
  | delete-statement
  | FETCH [ fetch-orientation [ FROM ] ] cursor-name [ INTO variable-name [ , variable-name ] ]
  | IF proc-search-condition THEN proc-stmt [ ; proc-stmt ]... [ ELSE proc-stmt [ ; proc-stmt ]... ] END IF
  | insert-statement
  | LEAVE label-name
  | [ label-name : ] LOOP proc-stmt [ ; proc-stmt ]... END LOOP [ label-name ]
  | OPEN cursor-name
  | PRINT proc-expr [ , 'string' ]
  | RETURN [ proc-expr ]
  | transaction-statement
procedure-name ::= user-defined-name

public-or-user-group-name ::= PUBLIC | user-group-name

query-specification ::= [ UNION [ ALL ] ] query-specification ... [ ORDER BY order-by-expression [ , order-by-expression ] ... ]

query-specification ::= ( query-specification )

select-statement ::= select-statement-with-into | select-statement

select-statement-with-into ::= select-statement SET variable-name = proc-expr

SIGNAL [ ABORT ] sqlstate-value

START TRANSACTION

update-statement ::= update-statement

UPDATE SET column-name = proc-expr [ , column-name = proc-expr ] ... WHERE CURRENT OF cursor-name

[ label-name : ] WHILE proc-search-condition DO [ proc-stmt [ ; proc-stmt ] ] ... END WHILE [ label-name ]

referencing-alias ::= OLD [ AS ] correlation-name [ NEW [ AS ] correlation-name ]

referential-actions ::= referential-update-action [ referential-delete-action ]

referential-update-action ::= ON UPDATE RESTRICT

referential-delete-action ::= ON DELETE CASCADE

| ON DELETE RESTRICT

release-statement ::= see RELEASE statement

result ::= user-defined-name data-type

rollback-statement ::= see ROLLBACK WORK statement

savepoint-name ::= user-defined-name

scalar-function ::= see Scalar Function list

scale ::= integer
ODEC Engine Reference

search-condition ::= search-condition AND search-condition
| search-condition OR search-condition
| NOT search-condition
| ( search-condition )
| predicate

select-item ::= expression [ [ AS ] alias-name ] | table-name . *

select-list ::= * | select-item [ , select-item ]...

set-function ::= COUNT (*)
| COUNT ( [ DISTINCT | ALL ] expression )
| SUM ( [ DISTINCT | ALL ] expression )
| AVG ( [ DISTINCT | ALL ] expression )
| MIN ( [ DISTINCT | ALL ] expression )
| MAX ( [ DISTINCT | ALL ] expression )

sqlstate-value ::= 'string'

table-constraint-definition ::= [ CONSTRAINT constraint-name ] table-constraint

table-constraint ::= UNIQUE (column-name [ , column-name ] . . . )
| PRIMARY KEY (column-name [ , column-name ] . . . )
| FOREIGN KEY (column-name [ , column-name ])

REFERENCES table-name
| ( column-name [ , column-name ] . . . )
| referential-actions

table-element ::= column-definition
| table-constraint-definition

| WHERE search-condition
| GROUP BY expression [ , expression ] . . .
| HAVING search-condition

table-name ::= user-defined-name
SQL Grammar Elements

table-privilege ::= \texttt{ALL}
  | \texttt{SELECT} \{ ( column-name [ , column-name ] ... ) \}
  | \texttt{UPDATE} \{ ( column-name [ , column-name ] ... ) \}
  | \texttt{INSERT} \{ ( column-name [ , column-name ] ... ) \}
  | \texttt{DELETE}
  | \texttt{ALTER}
  | \texttt{REFERENCES}

table-reference ::= \{ \texttt{OJ} outer-join-definition \}
  | table-name \[ \texttt{AS} \] alias-name
  | join-definition
  | ( join-definition )

transaction-statement ::= commit-statement
  | rollback-statement
  | release-statement

trigger-name ::= user-defined-name

user-password ::= user-name \[ : \] password

user-group-name ::= user-name \| group-name

user-name ::= user-defined-name

table-reference ::= ( value [ , value ] ... )

values-clause ::= \texttt{VALUES} ( expression [ , expression ] ... )

variable-name ::= user-defined-name

view-name ::= user-defined-name
**SQL Statement List**

SqlStatementList is defined as:

```
SqlStatementList
Statement ' ; ' | SqlStatementList ' ; '
Statement ::= StatementLabel ': ' Statement
BEGIN ... END block
CALL Statement
CLOSE CURSOR Statement
COMMIT Statement
DECLARE CURSOR Statement
DECLARE Variable Statement
DELETE Statement
FETCH Statement
IF Statement
INSERT Statement
LEAVE Statement
LOOP Statement
OPEN Statement
PRINT Statement
RELEASE SAVEPOINT Statement
RETURN Statement
ROLLBACK Statement
SAVEPOINT Statement
SELECT Statement
SET Statement
SIGNAL Statement
START TRANSACTION Statement
UPDATE Statement
WHILE Statement
```

**Predicate**

A predicate is defined as:

```
Expression CompareOperator Expression
| Expression [ NOT ] BETWEEN Expression AND Expression
| Expression [ NOT ] LIKE StringLiteral
| Expression IS [ NOT ] NULL
| NOT Predicate
| Predicate AND Predicate
| Predicate OR Predicate
| [ '(' Predicate ')' | CompareOperator ::= '=' | '>= ' | '>' | '<= ' |
| '<' | '<>'
| [ NOT ] IN value-list
```
**Expression**

An expression is defined as:

```
Number
| StringLiteral
| ColumnName
| VariableName
| **NULL**
| **CONVERT** ' (' Expression ', ' DataType ') ' 
| '-' Expression
| Expression '+' Expression
| Expression '-' Expression
| Expression '*' Expression
| Expression '/' Expression
| FunctionName '(' [ ExpressionList ] ')' 
| '(' Expression ')' 
| '(' D StringLiteral ')' 
| '(' T StringLiteral ')' 
| '(' TS StringLiteral ')' 
| @:IDENTITY
| @:ROWCOUNT
| @@IDENTITY
| @@ROWCOUNT
```

An expression list is defined as:

```
ExpressionList ::= Expression [ , Expression ... ]
```
Scalar Functions

The Pervasive ODBC Engine Interface supports ODBC scalar functions which may be included in an SQL statement as a primary expression.

This section lists the scalar functions supported by the Pervasive ODBC Engine Interface.

String Functions

String functions are used to process and manipulate columns that consist of text information, such as CHAR or LONGVARCHAR data types.

Arguments denoted as string can be the name of column, a string literal, or the result of another scalar function.

Table 2-18 String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII (string)</td>
<td>Returns the ASCII value of the left most character of string</td>
</tr>
<tr>
<td>BIT_LENGTH (string)</td>
<td>Returns the length in bits of string</td>
</tr>
<tr>
<td>CHAR (code)</td>
<td>Returns the ASCII character corresponding to ASCII value code. The argument must be an integer value.</td>
</tr>
<tr>
<td>CHAR_LENGTH (string)</td>
<td>Returns the number of characters in string.</td>
</tr>
<tr>
<td>CHARACTER_LENGTH (string)</td>
<td>Same as CHAR_LENGTH.</td>
</tr>
<tr>
<td>CONCAT (string1, string2)</td>
<td>Returns a string that results from combining string1 and string2.</td>
</tr>
<tr>
<td>LCASE or LOWER (string)</td>
<td>Converts all upper case characters in string to lower case.</td>
</tr>
<tr>
<td>LEFT (string, count)</td>
<td>Returns the left most count of characters in string. The value of count is an integer.</td>
</tr>
<tr>
<td>LENGTH (string)</td>
<td>Returns the number of characters in string. Trailing blanks and the string termination character are not returned.</td>
</tr>
</tbody>
</table>
Table 2-18  String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE (string1, string2 [, start])</td>
<td>Returns the starting position of the first occurrence of string1 within string2. The search within string2 begins at the first character position unless you specify a starting position (start). The search begins at the starting position you specify. The first character position in string2 is 1. The string1 is not found, the function returns the value zero.</td>
</tr>
<tr>
<td>LTRIM (string)</td>
<td>Returns the characters of string with leading blanks removed.</td>
</tr>
<tr>
<td>OCTET_LENGTH (string)</td>
<td>Returns the length in bytes of string.</td>
</tr>
<tr>
<td>POSITION (string1, string2)</td>
<td>Returns the position of string1 in string2. If string1 does not exist in string2, a zero is returned.</td>
</tr>
<tr>
<td>REPLACE (string1, string2, string3)</td>
<td>Searches string1 for occurrences of string2 and replaces each with string3. Returns the result. If no occurrences are found, string1 is returned.</td>
</tr>
<tr>
<td>REPPLICATE (string, count)</td>
<td>Returns a character string composed of string repeated count times. The value of count is an integer.</td>
</tr>
<tr>
<td>RIGHT (string, count)</td>
<td>Returns the right most count of characters in string. The value of count is an integer.</td>
</tr>
<tr>
<td>RTRIM (string)</td>
<td>Returns the characters of string with trailing blanks removed.</td>
</tr>
<tr>
<td>SPACE (count)</td>
<td>Returns a character string consisting of count spaces.</td>
</tr>
<tr>
<td>STUFF (string1, start, length, string2)</td>
<td>Returns a character string where length characters in string1 beginning at position start have been replaced by string2. The values of start and length are integers.</td>
</tr>
<tr>
<td>SUBSTRING (string1, start, length)</td>
<td>Returns a character string derived from string1 beginning at the character position specified by start for length characters.</td>
</tr>
<tr>
<td>UCASE or UPPER (string)</td>
<td>Converts all lower case characters in string to upper case.</td>
</tr>
</tbody>
</table>
Queries containing a WHERE clause with scalar functions RTRIM or LEFT can be optimized. For example, consider the following query:

```sql
SELECT * FROM T1, T2 WHERE T1.C1 = LEFT(T2.C1, 2)
```

In this case, both sides of the predicate are optimized if T1.C1 and T2.C2 are index columns. The *predicate* is the complete search condition following the WHERE keyword. Depending on the size of the tables involved in the join, the optimizer chooses the appropriate table to process first.

RTRIM and LEFT cannot be optimized if they are contained in a complex expression on either side of the predicate.

**Examples**

The following example creates a new table with an integer and a character column. It inserts 4 rows with values for the character column only, then updates the integer column of those rows with the ASCII character code for each character.

```sql
CREATE TABLE numchars(num INTEGER, chr CHAR(1) CASE)
INSERT INTO numchars (chr) VALUES('a')
INSERT INTO numchars (chr) VALUES('b')
INSERT INTO numchars (chr) VALUES('A')
INSERT INTO numchars (chr) VALUES('B')
UPDATE numchars SET num=ASCII(chr)
SELECT * FROM numchars
```

Results of SELECT:

<table>
<thead>
<tr>
<th>num</th>
<th>chr</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
</tbody>
</table>

```sql
SELECT num FROM numchars WHERE num=ASCII('a')
```

Results of SELECT:

<table>
<thead>
<tr>
<th>num</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
</tr>
</tbody>
</table>

The following example concatenates the first and last names in the Person table and results in "RooseveltBora".
SELECT CONCAT(First_name, Last_name) FROM Person WHERE First_name = 'Roosevelt'

The next example changes the case of the first name to lowercase and then to upper case, results in "roosevelt", "ROOSEVELT".

SELECT LCASE(First_name), UCASE(First_name) FROM Person WHERE First_name = 'Roosevelt'

The following example results in first name trimmed to three characters beginning from left, the length as 9 and locate results 0. This query results in "Roo", 9, 0

SELECT LEFT(First_name, 3), LENGTH(First_name), LOCATE(First_name, 'a') FROM Person WHERE First_name = 'Roosevelt'

The following example illustrates use of LTRIM and RTRIM functions on strings, results in "Roosevelt", "Roosevelt", "elt".

SELECT LTRIM(First_name), RTRIM(First_name), RIGHT(First_name, 3) FROM Person WHERE First_name = 'Roosevelt'

This substring lists up to three characters starting with the second character in the first name as "oos."

SELECT SUBSTRING(First_name, 2, 3) FROM Person WHERE First_name = 'Roosevelt'

SELECT ID, first_name FROM Person WHERE LCASE(First_name) = 'bruce'

**Numeric Functions**

Numeric functions are used to process and manipulate columns that consist of strictly numeric information, such as decimal and integer values.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS (numeric_exp)</td>
<td>Returns the absolute value of numeric_exp.</td>
</tr>
<tr>
<td>ACOS (float_exp)</td>
<td>Returns the arc cosine of float_exp as an angle, expressed in radians.</td>
</tr>
<tr>
<td>ASIN (float_exp)</td>
<td>Returns the arc sine of float_exp as an angle, expressed in radians.</td>
</tr>
<tr>
<td>ATAN (float_exp)</td>
<td>Returns the arc tangent of float_exp as an angle, expressed in radians.</td>
</tr>
</tbody>
</table>
## Table 2-19 Numeric Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAN2 (float_exp1, float_exp2)</td>
<td>Returns the arc tangent of the x and y coordinates, specified by float_exp1 and float_exp2, respectively, as an angle, expressed in radians.</td>
</tr>
<tr>
<td>CEILING (numeric_exp)</td>
<td>Returns the smallest integer greater than or equal to numeric_exp.</td>
</tr>
<tr>
<td>COS (float_exp)</td>
<td>Returns the cosine of float_exp, where float_exp is an angle expressed in radians.</td>
</tr>
<tr>
<td>COT (float_exp)</td>
<td>Returns the cotangent of float_exp, where float_exp is an angle expressed in radians.</td>
</tr>
<tr>
<td>DEGREES (numeric_exp)</td>
<td>Returns the number of degrees corresponding to numeric_exp radians.</td>
</tr>
<tr>
<td>EXP (float_exp)</td>
<td>Returns the exponential value of float_exp.</td>
</tr>
<tr>
<td>FLOOR (numeric_exp)</td>
<td>Returns the largest integer less than or equal to numeric_exp.</td>
</tr>
<tr>
<td>LOG (float_exp)</td>
<td>Returns the natural logarithm of float_exp.</td>
</tr>
<tr>
<td>LOG10 (float_exp)</td>
<td>Returns the base 10 logarithm of float_exp.</td>
</tr>
<tr>
<td>MOD (integer_exp1, integer_exp2)</td>
<td>Returns the remainder (modulus) of integer_exp1 divided by integer_exp2.</td>
</tr>
<tr>
<td>PI ()</td>
<td>Returns the constant value Pi as a floating point value.</td>
</tr>
<tr>
<td>POWER (numeric_exp, integer_exp)</td>
<td>Returns the value of numeric_exp to the power of integer_exp.</td>
</tr>
<tr>
<td>RADIANS (numeric_exp)</td>
<td>Returns the number of radians equivalent to numeric_exp degrees.</td>
</tr>
<tr>
<td>RAND (integer_exp)</td>
<td>Returns a random floating-point value using integer_exp as the optional seed value.</td>
</tr>
</tbody>
</table>
Table 2-19 Numeric Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND (numeric_exp, integer_exp)</td>
<td>Returns numeric_exp rounded to integer_exp places right of the decimal point. If integer_exp is negative, numeric_exp is rounded to integer_exp (absolute value of integer_exp) places to the left of the decimal point.</td>
</tr>
<tr>
<td>SIGN (numeric_exp)</td>
<td>Returns an indicator of the sign of numeric_exp. If numeric_exp is less than zero, -1 is returned. If numeric_exp equals zero, 0 is returned. If numeric_exp is greater than zero, 1 is returned.</td>
</tr>
<tr>
<td>SIN (float_exp)</td>
<td>Returns the sine of float_exp, where float_exp is an angle expressed in radians.</td>
</tr>
<tr>
<td>SQRT (float_exp)</td>
<td>Returns the square root of float_exp.</td>
</tr>
<tr>
<td>TAN (float_exp)</td>
<td>Returns the tangent of float_exp, where float_exp is an angle expressed in radians.</td>
</tr>
<tr>
<td>TRUNCATE (numeric_exp, integer_exp)</td>
<td>Returns numeric_exp truncated to integer_exp places right of the decimal point. If integer_exp is negative, numeric_exp is truncated to integer_exp (absolute value) places to the left of the decimal point.</td>
</tr>
</tbody>
</table>

Examples

The following example lists the Modulus of the number and capacity columns in a table named room.

SELECT MOD(Number, Capacity) FROM Room

The following example selects all salaries from a table named Faculty that are evenly divisible by 100.

SELECT Salary FROM Faculty WHERE MOD(Salary, 100) = 0
Date and time functions can be used to generate, process, and manipulate data that consists of date or time data types, such as DATE and TIME.

### Table 2-20 Time and Date Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURDATE ()</td>
<td>Returns the current date as a data value.</td>
</tr>
<tr>
<td>CURRENT_DATE ()</td>
<td>Returns the current date. In INSERT statements, use the CURDATE variable in the values clause to insert the current date into a table.</td>
</tr>
<tr>
<td>CURTIME ()</td>
<td>Returns the current local time.</td>
</tr>
<tr>
<td>CURRENT_TIME ()</td>
<td>Returns the current time. In INSERT statements, use the CURTIME variable in the values clause to insert the current time into a table.</td>
</tr>
<tr>
<td>DAYNAME (date_exp)</td>
<td>Returns a character string containing the data source-specific name of the day (for example, Sunday through Saturday or Sun. through Sat. for a data source that uses English, or Sonntag through Samstag for a data source that uses German) for the day portion of date_exp.</td>
</tr>
<tr>
<td>DAYOFMONTH (date_exp)</td>
<td>Returns the day of the month in date_exp as an integer in the range of 1 to 31.</td>
</tr>
<tr>
<td>DAYOFYEAR (date_exp)</td>
<td>Returns the day of the year based on the year field in date_exp as an integer value in the range of 1-366.</td>
</tr>
<tr>
<td>EXTRACT (extract_field, extract_source)</td>
<td>Returns the extract_field portion of the extract_source. The extract_source argument is a date, time or interval expression. The permitted values of extract_field are: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND</td>
</tr>
</tbody>
</table>

These values are returned from the target expression.
**SQL Grammar Elements**

### Table 2-20 Time and Date Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOUR (time_exp)</td>
<td>Returns the hour as an integer in the range of 0 to 23.</td>
</tr>
<tr>
<td>MINUTE (time_exp)</td>
<td>Returns the minute as an integer in the range 0 to 59.</td>
</tr>
<tr>
<td>MONTH (date_exp)</td>
<td>Returns the month as an integer in the range of 1 to 12.</td>
</tr>
<tr>
<td>MONTHNAME (date_exp)</td>
<td>Returns a character string containing the data source-specific name of the month (for example, September through December or Sept. through Dec. for a data source that uses English, or Settembre through Dicembre for a data source that uses Italian) for the month portion of date_exp.</td>
</tr>
<tr>
<td>NOW ()</td>
<td>Returns the current date and time as a timestamp value.</td>
</tr>
<tr>
<td>QUARTER (date_exp)</td>
<td>Returns the quarter in date_exp as an integer value in the range of 1-4, where 1 represents January 1 through March 31.</td>
</tr>
<tr>
<td>SECOND (time_exp)</td>
<td>Returns the second as an integer in the range of 0 to 59.</td>
</tr>
<tr>
<td>TIMESTAMPADD (interval, integer_exp, timestamp_exp)</td>
<td>Returns the timestamp calculated by adding integer_exp intervals of type interval to timestamp_exp.</td>
</tr>
<tr>
<td></td>
<td>The allowed values for interval are:</td>
</tr>
<tr>
<td></td>
<td>SQL_TSI_YEAR</td>
</tr>
<tr>
<td></td>
<td>SQL_TSI_QUARTER</td>
</tr>
<tr>
<td></td>
<td>SQL_TSI_MONTH</td>
</tr>
<tr>
<td></td>
<td>SQL_TSI_WEEK</td>
</tr>
<tr>
<td></td>
<td>SQL_TSI_DAY</td>
</tr>
<tr>
<td></td>
<td>SQL_TSI_HOUR</td>
</tr>
<tr>
<td></td>
<td>SQL_TSI_MINUTE</td>
</tr>
<tr>
<td></td>
<td>SQL_TSI_SECOND</td>
</tr>
<tr>
<td>TIMESTAMPDIFF (interval, timestamp_exp1, timestamp_exp2)</td>
<td>Returns the integer number of intervals of type interval by which timestamp_exp2 is greater than timestamp_exp1.</td>
</tr>
<tr>
<td></td>
<td>The values allowed for interval are the same as for TIMESTAMPADD.</td>
</tr>
</tbody>
</table>
Examples

The following example illustrates the use of hour.

```sql
SELECT c.Name, c.Credit_Hours FROM Course c WHERE c.Name = ANY (SELECT cl.Name FROM Class cl WHERE c.Name = cl.Name AND c.Credit_Hours > (HOUR (Finish_Time - Start_Time) + 1))
```

The following is an example of minute.

```sql
SELECT MINUTE(log) FROM billing
```

The following example illustrates the use of second.

```sql
SELECT SECOND(log) FROM billing
SELECT log FROM billing WHERE SECOND(log) = 31
```

The following example illustrates the use of now.

```sql
SELECT NOW() - log FROM billing
```

The following is a complex example that uses month, day, year, hour and minute.

```sql
SELECT Name, Section, MONTH(Start_Date), DAY(Start_Date), YEAR(Start_Date), HOUR(Start_Time), MINUTE(Start_Time) FROM Class
```

The following example illustrates use of curdate.

```sql
SELECT ID, Name, Section FROM Class WHERE (Start_Date - CURDATE()) <= 2 AND (Start_Date - CURDATE()) >= 0
```

The next example gives the day of the month and day of the week of the start date of class from the class table.

```sql
SELECT DAYOFMONTH(Start_date), DAYOFWEEK(Start_date) FROM Class
SELECT * FROM person WHERE YEAR(Date_Of_Birth) < 1970
```
**System Functions**  
System functions provide information at a system level.

*Table 2-21 System Functions*

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE()</td>
<td>Returns the current database name.</td>
</tr>
<tr>
<td>USER()</td>
<td>Returns the login name of the current user.</td>
</tr>
</tbody>
</table>

**Examples**  
The following examples show how to obtain the name of the current user and database:

```sql
SELECT USER();
SELECT DATABASE();
```

If you want to obtain this information for every record in a table, use the following (the example uses the Person table in DEMODATA):

```sql
SELECT USER(), DATABASE() FROM person
```

**Logical Functions**  
Logical functions are used to manipulate data based on certain conditions.

*Table 2-22 Logical Functions*

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF (predicate, expr1, expr2)</td>
<td>Returns <code>expr1</code> if predicate is true; otherwise, returns <code>expr2</code>.</td>
</tr>
<tr>
<td>NULL ()</td>
<td>Sets a column as NULL values.</td>
</tr>
<tr>
<td>IFNULL (exp, value)</td>
<td>If <code>exp</code> is NULL, <code>value</code> is returned. If <code>exp</code> is not null, <code>exp</code> is returned. The possible data type or types of <code>value</code> must be compatible with the data type of <code>exp</code>.</td>
</tr>
<tr>
<td>NULLIF (exp1, exp2)</td>
<td>NULLIF returns <code>exp1</code> if the two expressions are not equivalent. If the expressions are equivalent, NULLIF returns a NULL value.</td>
</tr>
</tbody>
</table>
The system scalar functions IF and NULL are SQL extensions. IF allows you to enter different values depending on whether the condition is true or false. For example, if you want to display a column with logical values as “true” or “false” instead of a binary representation, you would use the following SQL statement:

```sql
SELECT IF(logicalcol=1, 'True', 'False')
```

The system scalar function NULL allows you to set a column as null values. The syntax is:

```sql
NULL()
```

For example, the following SQL statement retrieves null values:

```sql
SELECT NULL() FROM person
```

The following statements demonstrate the IFNULL and NULLIF scalar functions. You use these functions when you want to do certain value substitution based on the presence or absence of NULLs and on equality.

```sql
CREATE TABLE Demo (col1 CHAR(3))
INSERT INTO Demo VALUES ('abc')
INSERT INTO Demo VALUES (NULL)
INSERT INTO Demo VALUES ('xyz')
```

Since the second row contains the NULL value, 'foo' is substituted in its place.

```sql
SELECT IFNULL(col1, 'foo') FROM Demo
```

This results in three rows fetched from one column:

```
"abc"
"foo"
"xyz"
3 rows fetched from 1 column.
```

The first row contains 'abc,' which matches the second argument of the following NULLIF call.

```sql
SELECT NULLIF(col1, 'abc') FROM Demo
```

A NULL is returned in its place:

```
<Null>
<Null>
"xyz"
3 rows fetched from 1 column.
```
The conversion function converts an expression to a data type.

**Table 2-23 Conversion Function**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONVERT (exp, type)</td>
<td>Converts exp to the type indicated. The possible types are:</td>
</tr>
<tr>
<td></td>
<td>SQL_BIGINT</td>
</tr>
<tr>
<td></td>
<td>SQL_BINARY</td>
</tr>
<tr>
<td></td>
<td>SQL_BIT</td>
</tr>
<tr>
<td></td>
<td>SQL_CHAR</td>
</tr>
<tr>
<td></td>
<td>SQL_DATE</td>
</tr>
<tr>
<td></td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td></td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td></td>
<td>SQL_FLOAT</td>
</tr>
<tr>
<td></td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td></td>
<td>SQL_LONGVARCHAR</td>
</tr>
<tr>
<td></td>
<td>SQL_NUMERIC</td>
</tr>
<tr>
<td></td>
<td>SQL_REAL</td>
</tr>
<tr>
<td></td>
<td>SQL_SMALLINT</td>
</tr>
<tr>
<td></td>
<td>SQL_TIME</td>
</tr>
<tr>
<td></td>
<td>SQL_TIMESTAMP</td>
</tr>
<tr>
<td></td>
<td>SQL_TINYINT</td>
</tr>
<tr>
<td></td>
<td>SQL_VARBINARY</td>
</tr>
<tr>
<td></td>
<td>SQL_VARCHAR</td>
</tr>
<tr>
<td></td>
<td>SQL_LONGVARBINARY</td>
</tr>
</tbody>
</table>

**Examples**

```sql
SELECT CONVERT(id, SQL_CHAR), CONVERT('1995-06-05', SQL_DATE),
       CONVERT('10:10:10', SQL_TIME),
       CONVERT('1990-10-10 10:10:10', SQL_TIMESTAMP),
       CONVERT('1990-10-10', SQL_TIMESTAMP)
FROM Faculty
```

```sql
SELECT Name FROM Class WHERE Start_date > CONVERT ('1995-05-07', SQL_DATE) + 31
```
Other Characteristics

**Creating Indexes**

The maximum column size for varchar columns is 254 bytes if the column does not allow Null values and 253 bytes if the column is nullable.

The maximum column size for char columns is 255 bytes if the column does not allow Null values and 254 bytes if the column is nullable.

The maximum Btrieve key size is 255. When a column is nullable and indexed a segmented key is created with 1 byte for the null indicator and a maximum 254 bytes from the column indexed. Varchar columns differ from char columns in that either length byte (Btrieve lstring) or a zero terminating byte (Btrieve zstring) are reserved, reducing the effective storage by 1 byte.

**Closing an Open Table**

Calling SQLFreeStmt with the SQL_CLOSE option changes the SQLSTATE but does not close the open tables used by the hStmt. To close the tables currently used by hStmt, SQLFreeStmt must be called with the SQL_DROP option.

In the following example, the Emp and Dept tables remain open:

```sql
SQLPrepare(hStmt, "SELECT * FROM Emp, Dept", SQL_NTS)
SQLExecute(hStmt)
SQLFetch until SQL_No_Data_Found
SQLFreeStmt(hStmt, SQL_CLOSE)
```

When SQLPrepare is subsequently called on the hStmt, the tables used in the previous statement are closed. For example, when the following call is made, both the Emp and Dept tables are closed by the Pervasive ODBC Engine Interface:

```sql
SQLPrepare(hStmt, "SELECT * FROM Customer", SQL_NTS)
```

The following call would then close the table Customer:

```sql
SQLFreeStmt(hStmt, SQL_DROP)
```

**Concurrency**

The timeliness of data, dynamic or snapshot, is determined by whether or not execution of a query results in a sort. Queries with DISTINCT, GROUP BY, or ORDER BY result in a temporary sort by
Pervasive ODBC Engine Interface, unless an index exists that satisfies the required ordering.

For those queries which do not result in a temporary sort by Pervasive ODBC Engine Interface, the data fetched is from the data files. For those queries that result in a temporary sort by Pervasive ODBC Engine Interface, the data fetched is from a temporary table. The temporary table is built from the required data in the original data file at SQLExecute time.

**Note** For some sort operations (for example, SELECT statements where long data columns are included in the select-list, or SELECT statements with GROUP BY), Pervasive ODBC Engine Interface may use bookmarks which Pervasive ODBC Engine Interface assumes are persistent within a SELECT statement. The situation may arise whereby another application updates or deletes the row that a bookmark references.

To avoid this situation, an application may set an exclusive lock on the table being sorted through a call to SQLSetStmtOption, with fOption = 1153 and vParam = 1.

---

**Comma as Decimal Separator**

Many locales, especially in Europe, use a comma to separate whole numbers from fractional numbers within a floating point numeric field. For example, these locales would use 1,5 instead of 1.5 to represent the number one-and-one-half.

Starting with Pervasive.SQL 2000i, the engine can support both the period “.” and the comma “,” as decimal separators. The database engine uses the decimal separator that is defined by the regional settings for the operating system.

**Note** When the decimal separator is not a period, numbers appearing in SQL statements must be enclosed in quotes.

---

**Client/Server Considerations**

Support for the comma as decimal separator is based on the locale setting in the operating system. Both the client operating system and the server operating system have a locale setting. The expected behavior varies according to both settings.
If either the server or client locale setting uses the comma as decimal separator, then the SRDE accepts both period-separated values and quoted comma-separated values.

If neither the server nor the client locale setting uses the comma decimal separator, then the SRDE does not accept comma-separated values.

Changing the Locale Setting

Decimal separator information can only be retrieved or changed for a Win32 machine (Windows 95/98/NT/2000).

The decimal setting for NetWare and Unix cannot be configured, and it is set to a period. If you have a NetWare or Unix server engine and you want to use the comma as decimal separator, you must ensure that all your client computers are set to a locale that uses the decimal separator.

➤ To view or change your locale setting

1. From the Start menu, choose Settings | Control Panel.
2. In the Control Panel window, double-click Regional Settings.
3. On the Regional Settings tab, select the desired country.
4. You must stop and restart the Pervasive.SQL services.

Examples

Example A - Server locale uses the comma for decimal separator

Client’s locale uses comma “,” as decimal separator:

```sql
CREATE TABLE t1 (c1 DECIMAL(10,3), c2 DOUBLE)
INSERT INTO t1 VALUES (10.123, 1.232)
INSERT INTO t1 VALUES ('10,123', '1.232')
SELECT * FROM t1 WHERE c1 = 10.123
SELECT * FROM t1 FROM c1 = '10,123'
```

The above two select statements, if executed from the client, return:

10,123, 1,232
10,123, 1,232
Client’s locale uses period “.” as decimal separator:

```sql
CREATE TABLE t1 (c1 DECIMAL(10,3), c2 DOUBLE)
INSERT INTO t1 VALUES (10.123, 1.232)
INSERT INTO t1 VALUES ('10,123', '1.232')
SELECT * FROM t1 WHERE c1 = 10.123
SELECT * FROM t1 WHERE c1 = '10,123'
```

The above two SELECT statements, if executed from the client, return:

```
10.123, 1.232
10.123, 1.232
```

**Example B - Server locale uses the period for decimal separator**

Client’s locale uses comma “,” as DECIMAL separator:

Same as client using comma “,” in Example A.

Client’s locale uses period “.” as DECIMAL separator:

```sql
CREATE TABLE t1 (c1 DECIMAL(10,3), c2 DOUBLE)
INSERT INTO t1 VALUES (10.123, 1.232)
INSERT INTO t1 VALUES ('10,123', '1,232')
-- error in assignment
SELECT * FROM t1 WHERE c1 = 10.123
SELECT * FROM t1 WHERE c1 = '10,123'
-- error in assignment
```

The first SELECT statement above, if executed from the client, returns:

```
10.123, 1.232
```

**OEM to ANSI Support**

Applications can now store or retrieve character data in the OEM character set using Pervasive.SQL, while allowing the data to be manipulated and displayed using the ANSI Windows character set. The Pervasive ODBC driver translation DLL can perform all necessary translations between the two character sets. This feature can be turned on or off for each DSN. To access the switch, click **Options...** on the Pervasive ODBC DSN Setup dialog box.
The Pervasive Control Center (PCC) and the SQL Data Manager (SQLDM) are not fully OEM-character aware if you use extended ASCII characters for column or table names. However, any character data that is passed to and from the database is correctly translated between the OEM and ANSI character sets.

If your application connects to the data source using SQLDriverConnect, you can also specify the translation DLL using the connection string option TRANSLATIONDLL=path_and_DLL_name. The translation DLL name for Pervasive is W32BTXL T.DLL.

NOTE: The OEM to ANSI translation option is available only for client or local engine DSNs.
Data Types

Pervasive.SQL Supported Data Types

The following table shows the ODBC SQL data types that Pervasive ODBC Engine Interface supports. The application developer can use SQLGetTypeInfo to determine which of these ODBC SQL data types are supported by a Pervasive ODBC Engine Interface. (See the Microsoft ODBC Programmer’s Reference for further details on SQLGetTypeInfo.)
# Data Types

## Pervasive.SQL Supported Data Types

<table>
<thead>
<tr>
<th>Pervasive.SQL Data Type</th>
<th>ODBC Data Type (ODBC type number)</th>
<th>Precision/ Size</th>
<th>Create Parameters</th>
<th>Unsigned</th>
<th>Pervasive-Oracle</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>SQL_BIT (-7)</td>
<td>1</td>
<td>&lt;none&gt;</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>TINYINT</td>
<td>SQL_TINYINT (-6)</td>
<td>3</td>
<td>&lt;none&gt;</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>UTINYINT</td>
<td>SQL_TINYINT (-6)</td>
<td>3</td>
<td>&lt;none&gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>SQL_LONGVARBINARY (-4)</td>
<td>2147483648</td>
<td>&lt;none&gt;</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>BINARY</td>
<td>SQL_BINARY (-2)</td>
<td>255</td>
<td>max length</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>SQL_LONGVARCHAR (-1)</td>
<td>2147483648</td>
<td>&lt;none&gt;</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>CHAR</td>
<td>SQL_CHAR (1)</td>
<td>255</td>
<td>length</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>SQL_DECIMAL (3)</td>
<td>15</td>
<td>precision,scale</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>SQL_DECIMAL (3)</td>
<td>64</td>
<td>precision,scale</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>INTEGER</td>
<td>SQL_INTEGER (4)</td>
<td>10</td>
<td>&lt;none&gt;</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>UINTTEGER</td>
<td>SQL_INTEGER (4)</td>
<td>10</td>
<td>&lt;none&gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IDENTITY</td>
<td>SQL_INTEGER (4)</td>
<td>10</td>
<td>&lt;none&gt;</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SQL_SMALLINT (5)</td>
<td>5</td>
<td>&lt;none&gt;</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>USMALLINT</td>
<td>SQL_SMALLINT (5)</td>
<td>5</td>
<td>&lt;none&gt;</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SMALLIDENTITY</td>
<td>SQL_SMALLINT (5)</td>
<td>5</td>
<td>&lt;none&gt;</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FLOAT</td>
<td>SQL_FLOAT</td>
<td>7</td>
<td>&lt;none&gt;</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>REAL</td>
<td>SQL_REAL (7)</td>
<td>7</td>
<td>&lt;none&gt;</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>SQL_DOUBLE (8)</td>
<td>15</td>
<td>&lt;none&gt;</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DATE</td>
<td>SQL_DATE (9)</td>
<td>10</td>
<td>&lt;none&gt;</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>TIME</td>
<td>SQL_TIME (10)</td>
<td>8</td>
<td>&lt;none&gt;</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>SQL_TIMESTAMP (11)</td>
<td>19</td>
<td>&lt;none&gt;</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>SQL_VARCHAR (12)</td>
<td>254</td>
<td>length</td>
<td>N/A</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Supported Data Types

The table below also shows the mapping that the Pervasive ODBC Engine Interface performs between Btrieve data types and ODBC data types. The Pervasive ODBC Engine Interface converts these data types to an ODBC default type, unless another data type conversion is specified by the user when `SQLGetData` or `SQLBindCol` is called. (For a discussion of data type conversions, see Appendix D of the *Microsoft ODBC Programmer’s Reference*.) An explanation of the columns in the table appears below the table.

**Table A-2  Fully Supported Data Types**

<table>
<thead>
<tr>
<th>Pervasive.SQL Data Types</th>
<th>Btrieve Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>CHAR(^1)</td>
</tr>
<tr>
<td>BIT</td>
<td>BIT</td>
</tr>
<tr>
<td>TINYINT</td>
<td>INTEGER(1)</td>
</tr>
<tr>
<td>UTINYINT</td>
<td>USIGNEDINT(1)</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>BLOB(^2)</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>CLOB(^2)</td>
</tr>
<tr>
<td>CHAR</td>
<td>CHAR</td>
</tr>
<tr>
<td>DECIMAL (64,64)</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INTEGER(4)</td>
</tr>
<tr>
<td>UINTTEGER</td>
<td>USIGNEDINT(4)</td>
</tr>
<tr>
<td>USMALLINT</td>
<td>USIGNEDINT(2)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>INTEGER(2)</td>
</tr>
<tr>
<td>REAL</td>
<td>FLOAT(4)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT(8)</td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>ZSTRING</td>
</tr>
</tbody>
</table>

The data types in the left column of Table A-2 can be created with a `CREATE TABLE` statement.
Data Types

1 Indexed binary columns are created as Btrieve CHAR keys but are configured in the DDF to hold and retrieve BINARY data. The ODBC type is SQL_BINARY.

2 Fully supported LONGVARCHAR and LONGVARBINARY data types map to column definitions BLOB, CLOB. A BLOB and CLOB are both created as an 8-byte fixed column containing a 4-byte length and 4-byte offset.

Pervasive.SQL now supports multiple LONGVARCHAR and LONGVARBINARY columns per table. The data is stored according to the offset in the variable length portion of the record. The variable length portion of data can vary from the column order of the data depending on how the data is manipulated. Consider the following example.

```sql
CREATE TABLE BlobDataTest (  
    Nbr  UINT,          // Fixed record (Type 14)  
    Clob1 LONGVARCHAR,  // Fixed record (Type 21)  
    Clob2 LONGVARCHAR,  // Fixed record (Type 21)  
    Blob1 LONGVARBINARY,  // Fixed record (Type 21)  
)  
```

On disk, the physical record would normally look like this:

```
[Fixed Data (Nbr, Clob1header, Clob2header, Blob1header)] [ClobData1] [ClobData2] [BlobData1]
```

Now alter column Nbr to a LONGVARCHAR column:

```sql
ALTER TABLE BlobDataTest ALTER Nbr LONGVARCHAR
```

On disk, the physical record now looks like this:

```
[Fixed Data (Nbrheader, Clob1header, Clob2header, Blob1header)] [NbrClobData] [ClobData1] [ClobData2] [BlobData1]
```

As you can see, the variable length portion of the data is not in the column order for the existing data.

For newly inserted records, however, the variable length portion of the data is in the column order for the existing data.

```
[Fixed Data (Nbrheader, Clob1header, Clob2header, Blob1header)] [NbrClobData] [ClobData1] [ClobData2] [BlobData1]
```
Notes on CHAR, VARCHAR, and LONGVARCHAR

- CHAR columns are padded with blanks to "fill" the columns
- VARCHAR/LONGVARCHAR are not padded with blanks to "fill" the columns. The significant data is terminated with a NULL character.
- In all cases the trailing blanks are NOT significant in comparison operations (LIKE and =). However, in the LIKE case, if a space is explicitly entered in the query (like 'abc %'), the space before the wildcard does matter. In this example you are looking for 'abc<space><any other character>'

Notes on BINARY and LONGVARBINARY

- BINARY columns are padded with binary zeros to "fill" the columns
- LONGVARBINARY are NOT padded with blanks to "fill" the columns.
- Current engine does not compare BINARY/LONGVARBINARY. If you try, you get "Cannot compare a binary data type." error message.

<table>
<thead>
<tr>
<th>NEW Data Types</th>
<th>LEGACY Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>INTEGER(8)</td>
</tr>
<tr>
<td>UBIGINT</td>
<td>UNSIGNEDINT(8)</td>
</tr>
<tr>
<td>CURRENCY</td>
<td>CURRENCY</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>IDENTITY</td>
<td>AUTOINC (4)</td>
</tr>
<tr>
<td>SMALLIDENTITY</td>
<td>AUTOINC (2)</td>
</tr>
</tbody>
</table>

Pervasive.SQL supports creating these columns, with the following limitations:

- BIGINT, UBIGINT data is available for read and write but get data and put data operations cannot bind default values for SQL_BIGINT or SQL_NUMERIC. These Data Types currently map to SQL_DECIMAL.
Data Types

- CURRENCY is supported as a create table parameter but it is mapped to ODBC SQL_DECIMAL.
- NUMERIC will be supported as a create table parameter but it is mapped to ODBC SQL_DECIMAL.
- SMALLIDENTITY, IDENTITY are mapped to ODBC SQL_SMALLINT and SQL_INTEGER.

Table A-4 Legacy Data Types

<table>
<thead>
<tr>
<th>Exposed Data Types</th>
<th>Legacy Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>LOGICAL(1)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>LOGICAL(2)</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>LVAR</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>NOTE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERICSA</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>NUMERICSTS</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>MONEY</td>
</tr>
<tr>
<td>REAL</td>
<td>BFLOAT(4)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>BFLOAT(8)</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>LSTRING</td>
</tr>
</tbody>
</table>

The above items in the right column can not be created with Pervasive.SQL grammar. To create these legacy data types please use the SQL Data Manager.

This table outlines the relationship between existing Pervasive.SQL data types and ODBC (SQL) data types. The following table lists the

Table A-5 Pervasive.SQL to ODBC Data Type Mapping

<table>
<thead>
<tr>
<th>Pervasive.SQL 7.0 Data Type</th>
<th>Type Code</th>
<th>Valid Length</th>
<th>ODBC Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOINC</td>
<td>15</td>
<td>2</td>
<td>SQL_SMALLINT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td>BFLOAT</td>
<td>9</td>
<td>4</td>
<td>SQL_REAL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>SQL_DOUBLE</td>
</tr>
</tbody>
</table>
## Pervasive.SQL Supported Data Types

### Table A-5  Pervasive.SQL to ODBC Data Type Mapping

<table>
<thead>
<tr>
<th>Pervasive.SQL 7.0 Data Type</th>
<th>Type Code</th>
<th>Valid Length</th>
<th>ODBC Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>16</td>
<td>1</td>
<td>SQL_BIT</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>0</td>
<td>1-255</td>
<td>SQL_CHAR</td>
</tr>
<tr>
<td>CURRENCY</td>
<td>19</td>
<td>8</td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>DATE</td>
<td>3</td>
<td>4</td>
<td>SQL_DATE</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>5</td>
<td>1-10</td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>FLOAT</td>
<td>2</td>
<td>4</td>
<td>SQL_REAL</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>SQL_DOUBLE</td>
</tr>
<tr>
<td>INTEGER</td>
<td>1</td>
<td>1</td>
<td>SQL_TINYINT</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>SQL_SMALLINT</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td>SQL_INTEGER</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>7</td>
<td>1</td>
<td>SQL_BIT</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>SQL_SMALLINT</td>
</tr>
<tr>
<td>LSTRING</td>
<td>10</td>
<td>2 - 255</td>
<td>SQL_VARCHAR</td>
</tr>
<tr>
<td>LVAR</td>
<td>13</td>
<td>5 - 32Kb</td>
<td>SQL_LONGVARCHAR</td>
</tr>
<tr>
<td>MONEY</td>
<td>6</td>
<td>1 - 10</td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>NOTE</td>
<td>12</td>
<td>5 - 32Kb</td>
<td>SQL_LONGVARCHAR</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>8</td>
<td>1 - 15</td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>NUMERICSA</td>
<td>18</td>
<td>1 - 15</td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>NUMERICSTS</td>
<td>17</td>
<td>2 - 15</td>
<td>SQL_DECIMAL</td>
</tr>
<tr>
<td>STRING</td>
<td>0</td>
<td>1-255</td>
<td>SQL_CHAR</td>
</tr>
<tr>
<td>TIME</td>
<td>4</td>
<td>4</td>
<td>SQL_TIME</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>20</td>
<td>8</td>
<td>SQL_TIMESTAMP</td>
</tr>
<tr>
<td>UNSIGNED</td>
<td>14</td>
<td>1</td>
<td>SQL_TINYINT</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>SQL_SMALLINT</td>
</tr>
</tbody>
</table>
## Data Types

### Table A-5  Pervasive.SQL to ODBC Data Type Mapping

<table>
<thead>
<tr>
<th>Pervasive.SQL 7.0 Data Type</th>
<th>Type Code</th>
<th>Valid Length</th>
<th>ODBC Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>SQL_INTEGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SQL_DECIMAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ZSTRING</td>
<td>11</td>
<td>2 - 255</td>
<td>SQL_VARCHAR</td>
</tr>
</tbody>
</table>

valid length and valid value range for each Pervasive.SQL data type. "N/A means "not applicable.".

### Table A-6  Data Type Valid Lengths and Value Ranges

<table>
<thead>
<tr>
<th>Pervasive.SQL Data Type Name</th>
<th>ODBC Data Type (ODBC type number)</th>
<th>Valid Value Range</th>
<th>Valid Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>SQL_BIT (-7)</td>
<td>0 or 1</td>
<td>1</td>
</tr>
<tr>
<td>TINYINT</td>
<td>SQL_TINYINT (-6)</td>
<td>-128 - 127</td>
<td>1</td>
</tr>
<tr>
<td>UTINYINT</td>
<td>SQL_TINYINT (-6)</td>
<td>0 - 255</td>
<td>1</td>
</tr>
<tr>
<td>LONGVARBINARY</td>
<td>SQL_LONGVARBINARY (-4)</td>
<td>2147483647</td>
<td>n/a</td>
</tr>
<tr>
<td>BINARY</td>
<td>SQL_BINARY (-2)</td>
<td>N/A</td>
<td>1 - 255</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>SQL_LONGVARCHAR (-1)</td>
<td>2147483647</td>
<td>n/a</td>
</tr>
<tr>
<td>CHAR</td>
<td>SQL_CHAR (1)</td>
<td>N/A</td>
<td>1 - 255</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>SQL_DECIMAL (3)</td>
<td>-9223372036854775808 - 9223372036854775807</td>
<td>1 - 15</td>
</tr>
<tr>
<td>BIGINT</td>
<td>SQL_DECIMAL</td>
<td>9.223372036855e+18</td>
<td>8</td>
</tr>
<tr>
<td>UBIGINT</td>
<td>SQL_DECIMAL</td>
<td>1.844674407371e+19</td>
<td>8</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>SQL_DECIMAL (3)</td>
<td>Depends on the length and number of decimal places.</td>
<td>1 - 64</td>
</tr>
<tr>
<td>INTEGER</td>
<td>SQL_INTEGER (4)</td>
<td>-2147483648 – 2147483647</td>
<td>4</td>
</tr>
<tr>
<td>UINTTEGER</td>
<td>SQL_INTEGER (4)</td>
<td>0 – 4294967295</td>
<td>4</td>
</tr>
<tr>
<td>IDENTITY</td>
<td>SQL_INTEGER (4)</td>
<td>1 – 2147483647</td>
<td>4</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SQL_SMALLINT (5)</td>
<td>-32768 – +32767</td>
<td>2</td>
</tr>
<tr>
<td>USMALLINT</td>
<td>SQL_SMALLINT (5)</td>
<td>0 – 65535</td>
<td>2</td>
</tr>
</tbody>
</table>
Limitations on LONGVARCHAR and LONGVARBINARY

- The LIKE predicate operates on the first 65500 characters of the column data.
- All other predicates operate on the first 256 characters of the column data.
- SELECT statements with GROUP BY, DISTINCT, and ORDER BY return all the data but only order on the first 256 characters of the column data.
- In a single call to SQLGetData, the maximum number of characters returned by Pervasive ODBC Engine Interface for a LONGVARCHAR or LONGVARBINARY columns is 65500. Multiple calls must be made to SQLGetData to retrieve column data over 65500 characters.
- Though the maximum amount of data that can be inserted into a LONGVARCHAR/LONGVARBINARY column is 2GB, using a literal in an INSERT statement reduces this amount to 1000 characters. You can insert more than 1000 characters by using a parameterized insert.
Comparison of Floats

Pervasive ODBC Engine Interface compares floating point numbers in comparison predicates using an almost equals algorithm. For example, 12.203 = 12.20300000000001, and 12.203 is >= 12.20300000000001. The epsilon value defined as DBL_EPSILON is (.2204460492503131e-016). This feature works for large numbers, but > and < will not be detected for small numbers; small numbers will be detected as equal.

Note If you require precision to many decimal places, use the Decimal data type instead of the Real or Float data type.

Here is the comparison routine that Pervasive ODBC Engine Interface uses for the SQL_DOUBLE data type (which maps to the C double type). For the SQL_REAL data type (which maps to the C float type), Pervasive ODBC Engine Interface uses FLT_EPSILON, which is (.2204460492503131e-016).

SHORT sCnvDblCmp(
   DOUBLE  d1,
   DOUBLE  d2)
{
   if (d1 == d2)
      return 0;
   if (d1 > d2)
   {
      if (d1 > d2 + DBL_EPSILON)
         return(1);
   }
   else
   {
      if (d2 > d1 + DBL_EPSILON)
         return(-1);
   }

   return(0);
}

Representation of Infinity

When Pervasive ODBC Engine Interface is required by an application to represent infinity, it can do so in either a 4-byte (C float type) or 8-byte (C double type) form, and in either a
hexadecimal or character representation, as shown in the following table:

**Table A-7  Infinity Representation**

<table>
<thead>
<tr>
<th>Value</th>
<th>Float Hexadecimal</th>
<th>Float Character</th>
<th>Double Hexadecimal</th>
<th>Double Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Positive</td>
<td></td>
<td></td>
<td>0x7FEFFFFFFFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>Maximum Negative</td>
<td></td>
<td></td>
<td>0xFFEFFFFFFFFFFFFF</td>
<td></td>
</tr>
<tr>
<td>Infinity Positive</td>
<td>0x7F800000</td>
<td>1E999</td>
<td>0x7FF000000000000</td>
<td>1E999</td>
</tr>
<tr>
<td>Infinity Negative</td>
<td>0xFF800000</td>
<td>-1E999</td>
<td>0xFFF000000000000</td>
<td>-1E999</td>
</tr>
</tbody>
</table>
**Data Types**

**Btrieve Data Types**

For historical reasons, the two standard data types, STRING and UNSIGNED BINARY, are also offered as extended data types.

Internally, the MicroKernel compares string keys on a byte-by-byte basis, from left to right. The MicroKernel sorts string keys according to their ASCII value, however, you can define string keys to be case insensitive or to use an alternate collating sequence (ACS).

The MicroKernel compares unsigned binary keys one word at a time. It compares these keys from right to left because the Intel 8086 family of processors reverses the high and low bytes in an integer.

If a particular data type is available in more than one size (for example, both 4- and 8-byte FLOAT values are allowed), the Key Length parameter (used in the creation of a new key) defines the size that will be expected for all values of that particular key. Any attempt to define a key using a Key Length that is not allowed results in a Status 29 (Invalid Key Length).

Table A-8 lists the extended key types and their associated codes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Code</th>
<th>Type</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>0</td>
<td>LSTRING</td>
<td>10</td>
</tr>
<tr>
<td>INTEGER</td>
<td>1</td>
<td>ZSTRING</td>
<td>11</td>
</tr>
<tr>
<td>FLOAT</td>
<td>2</td>
<td>UNSIGNED BINARY</td>
<td>14</td>
</tr>
<tr>
<td>DATE</td>
<td>3</td>
<td>AUTOINCREMENT</td>
<td>15</td>
</tr>
<tr>
<td>TIME</td>
<td>4</td>
<td>NUMERICSTS</td>
<td>17</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>5</td>
<td>NUMERICSA</td>
<td>18</td>
</tr>
<tr>
<td>MONEY</td>
<td>6</td>
<td>CURRENCY</td>
<td>19</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>7</td>
<td>TIMESTAMP</td>
<td>20</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>8</td>
<td>WSTRING</td>
<td>25</td>
</tr>
<tr>
<td>BFLOAT</td>
<td>9</td>
<td>WZSTRING</td>
<td>26</td>
</tr>
</tbody>
</table>
The following sections, arranged alphabetically by key type, describe the extended key types and their internal storage formats.

**AUTOINCREMENT**

The AUTOINCREMENT key type is a signed Intel integer that can be either two or four bytes long. Internally, AUTOINCREMENT keys are stored in Intel binary integer format, with the high-order and low-order bytes reversed within a word. The MicroKernel sorts AUTOINCREMENT keys by their absolute (positive) values, comparing the values stored in different records a word at a time from right to left. AUTOINCREMENT keys may be used to automatically assign the next highest value when a record is inserted into a file.

The following restrictions apply to AUTOINCREMENT keys:

- An AUTOINCREMENT key must be defined as unique.
- An AUTOINCREMENT key cannot be segmented. However, an AUTOINCREMENT key can be included as an integer segment of another key, as long as the AUTOINCREMENT key has been defined as a separate, single key first, and the AUTOINCREMENT key number is lower than the segmented key number.
- An AUTOINCREMENT key cannot overlap another key.
- All AUTOINCREMENT keys must be ascending.

The MicroKernel treats AUTOINCREMENT key values as follows when you insert records into a file:

- If you specify a value of binary 0 for the AUTOINCREMENT key, the MicroKernel assigns a value to the key based on the following criteria:
  - If you are inserting the first record in the file, the MicroKernel assigns the value of 1 to the AUTOINCREMENT key.
  - If records already exist in the file, the MicroKernel assigns the key a value that is one number higher than the highest existing absolute value in the file.
If you specify a nonzero value for the AUTOINCREMENT key, the MicroKernel inserts the record into the file and uses the specified value as the key value. If a record containing that value already exists in the file, the MicroKernel returns an error status code, and does not insert the record.

When you delete a record containing an AUTOINCREMENT key, the MicroKernel completely removes the record from the file. The MicroKernel does not reuse the deleted key value unless you specify that value when you insert another record into the file, or unless you deleted the record with the highest value.

As mentioned previously, the MicroKernel always sorts AUTOINCREMENT keys by their absolute values. For example, you can do the following:

- Specify a negative value for an AUTOINCREMENT key when you insert a record.
- Update a record and negate the value for the AUTOINCREMENT key.

In any case, the MicroKernel sorts the key according to its absolute value. This allows you to use negation to flag records without altering the record’s position in the index. In addition, when you perform a Get operation and specify a negative value in the key buffer, the MicroKernel treats the absolute value as the key. Also, if a given value is already used, the negative that value cannot be inserted since it is considered a duplicate.

You can initialize the value of a field in all or some records to zero and later add an index of type AUTOINCREMENT. This feature allows you to prepare for an AUTOINCREMENT key without actually building the index until it is needed.

When you add the index, the MicroKernel changes the zero values in each field appropriately, beginning its numbering with a value equal to the greatest value currently defined in the field, plus one. If nonzero values exist in the field, the MicroKernel does not alter them. However, the MicroKernel returns an error status code if nonzero duplicate values exist in the field.
BFLOAT

The BFLOAT key type is a single or double-precision real number. A single-precision real number is stored with a 23-bit mantissa, an 8-bit exponent biased by 128, and a sign bit. The internal layout for a 4-byte float is as follows:

The representation of a double-precision real number is the same as that for a single-precision real number, except that the mantissa is 55 bits instead of 23 bits. The least significant 32 bits are stored in bytes 0 through 3.

The BFLOAT type is commonly used in legacy BASIC applications. Microsoft refers to this data type as MBF (Microsoft Binary Format), and no longer supports this type in the Visual Basic environment.

CHAR

Note In previous versions of Pervasive.SQL, this data type was referred to as STRING

The CHAR key type is a sequence of characters ordered from left to right. Each character is represented in ASCII format in a single byte, except when the MicroKernel is determining whether a key value is null.

CURRENCY

The CURRENCY key type represents an 8-byte signed quantity, sorted and stored in Intel binary integer format; therefore, its internal representation is the same as an 8-byte INTEGER data type. The CURRENCY data type has an implied four digit scale of decimal places, which represents the fractional component of the currency data value.
Data Types

DATE

The DATE key type is stored internally as a 4-byte value. The day and the month are each stored in 1-byte binary format. The year is a 2-byte binary number that represents the entire year value. The MicroKernel places the day into the first byte, the month into the second byte, and the year into a two-byte word following the month.

An example of C structure used for date fields would be:

```c
TYPE dateField {
    char day;
    char month;
    integer year;
}
```

The year portion of a date field is expected to be set to the integer representation of the entire year. For example, 2,001 for the year 2001.

DECIMAL

The DECIMAL key type is stored internally as a packed decimal number with two decimal digits per byte. The internal representation for an $n$-byte DECIMAL field is as follows:

```
    | byte 0 | byte 1 | byte n-1 |
    | 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 | ... | ... |
    | digit 1 digit 2 digit 3 digit 4 | ... | digit 2n-1 sign nibble |
```

The sign nibble is either $0xF$ or $0xC$ for positive numbers and $0xD$ for negative numbers. The decimal point is implied; no decimal point is stored in the DECIMAL field. Your application is responsible for tracking the location of the decimal point for the value in a
DECIMAL field. All the values for a DECIMAL key type must have the same number of decimal places in order for the MicroKernel to collate the key correctly. The DECIMAL type is commonly used in COBOL applications.

An eight-byte decimal can hold 15 digits plus the sign. A ten-byte decimal can hold 19 digits plus the sign. The decimal value is expected to be left-padded with zeros.

FLOAT

The FLOAT key type is consistent with the IEEE standard for single and double-precision real numbers. The internal format for a 4-byte FLOAT consists of a 23-bit mantissa, an 8-bit exponent biased by 127, and a sign bit, as follows:

A FLOAT key with 8 bytes has a 52-bit mantissa, an 11-bit exponent biased by 1023, and a sign bit. The internal format is as follows:
Data Types

INTEGER
The INTEGER key type is a signed whole number and can contain any number of digits. Internally, INTEGER fields are stored in Intel binary integer format, with the high-order and low-order bytes reversed within a word. The MicroKernel evaluates the key from right to left. The sign must be stored in the high bit of the rightmost byte. The INTEGER type is supported by most development environments.

Table A-9 INTEGER Key Type

<table>
<thead>
<tr>
<th>Length in Bytes</th>
<th>Value Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 – 255</td>
</tr>
<tr>
<td>2</td>
<td>-32768 – 32767</td>
</tr>
<tr>
<td>4</td>
<td>-2147483648 – 2147483647</td>
</tr>
<tr>
<td>8</td>
<td>-9223372036854775808 – 9223372036854775807</td>
</tr>
</tbody>
</table>

LOGICAL
The LOGICAL key type is stored as a 1 or 2-byte value. The MicroKernel collates LOGICAL key types as strings. Doing so allows your application to determine the stored values that represent true or false.

LSTRING
The LSTRING key type has the same characteristics as a regular STRING type, except that the first byte of the string contains the binary representation of the string’s length. The LSTRING key type is limited to a maximum size of 255 bytes. The length stored in byte 0 of an LSTRING key determines the number of significant bytes. The MicroKernel ignores any values beyond the specified length of the string when sorting values. The LSTRING type is commonly used in Pascal applications.

MONEY
The MONEY key type has the same internal representation as the DECIMAL type, with an implied two decimal places.
NUMERIC

NUMERIC values are stored as ASCII strings, right justified with leading zeros. Each digit occupies one byte internally. The rightmost byte of the number includes an embedded sign with an EBCDIC value. Table A-10 indicates how the rightmost digit is represented when it contains an embedded sign for positive and negative numbers.

Table A-10  Rightmost Digit with Embedded Sign

<table>
<thead>
<tr>
<th>Digit</th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>J</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>K</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>L</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>N</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>O</td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td>P</td>
</tr>
<tr>
<td>8</td>
<td>H</td>
<td>Q</td>
</tr>
<tr>
<td>9</td>
<td>I</td>
<td>R</td>
</tr>
<tr>
<td>0</td>
<td>{</td>
<td>}</td>
</tr>
</tbody>
</table>

For positive numbers, the rightmost digit can be represented by 1 through 0 instead of A through }. The MicroKernel processes positive numbers represented either way. The NUMERIC type is commonly used in COBOL applications.

NUMERICSA

The NUMERICSA key type (sometimes called NUMERIC SIGNED ASCII) is a COBOL data type that is the same as the NUMERIC data type, except that the embedded sign has an ASCII value instead of an EBCDIC value.
Data Types

**NUMERICSTS**

The NUMERICSTS key type (sometimes called SIGN TRAILING SEPARATE) is a COBOL data type that has values resembling those of the NUMERIC data type. NUMERICSTS values are stored as ASCII strings and right justified with leading zeros. However, the rightmost byte of a NUMERICSTS string is either “+” (ASCII 0x2B) or “-” (ASCII 0x2D). This differs from NUMERIC values that embed the sign in the rightmost byte along with the value of that byte.

**TIME**

The TIME key type is stored internally as a 4-byte value. Hundredths of a second, second, minute, and hour values are each stored in 1-byte binary format. The MicroKernel places the hundredths of a second value into the first byte, followed respectively by the second, minute, and hour values.

**TIMESTAMP**

The TIMESTAMP data type represents a time and date value. In SQL applications, use this data type to stamp a record with the time and date of the last update to the record. TIMESTAMP values are stored in 8-byte unsigned values representing septa seconds \(10^{-7}\) second) since January 1, 0001 in a Gregorian calendar.
TIMESTAMP is intended to cover time and data values made up of the following components: year, month, day, hour, minute, and second. The following table indicates the valid values of each of these components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Valid Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>0001 to 9999</td>
</tr>
<tr>
<td>MONTH</td>
<td>01 to 12</td>
</tr>
<tr>
<td>DAY</td>
<td>01 to 31, constrained by the value of MONTH and YEAR in the Gregorian calendar.</td>
</tr>
<tr>
<td>HOUR</td>
<td>00 to 23</td>
</tr>
<tr>
<td>MINUTE</td>
<td>00 to 59</td>
</tr>
<tr>
<td>SECOND</td>
<td>00 to 59</td>
</tr>
</tbody>
</table>

**UNIGNED BINARY**

The MicroKernel sorts UNSIGNED BINARY keys as unsigned INTEGER keys. An UNSIGNED BINARY key could contain any even number of bytes. The MicroKernel compares UNSIGNED BINARY keys from right to left.

An UNSIGNED BINARY key is sorted in the same manner as an INTEGER key. The differences between an UNSIGNED BINARY key and an INTEGER key are that an INTEGER has a sign bit, while an UNSIGNED BINARY type does not, and an UNSIGNED BINARY key can be longer than 4 bytes.

**WSTRING**

WSTRING is a Unicode string that is not null-terminated. The length of the string is determined by the field length.

**WZSTRING**

WZSTRING is a Unicode string that is double null-terminated. The length of this string is determined by the position of the Unicode NULL (two null bytes) within the field. This corresponds to the ZSTRING type supported in Btrieve.
ZSTRING

The ZSTRING key type corresponds to a C string. It has the same characteristics as a regular string type except that a ZSTRING type is terminated by a binary 0. The MicroKernel ignores any values beyond the first binary 0 it encounters in the ZSTRING, except when the MicroKernel is determining whether a key value is null.
Reserved words have one or more specific meaning and are recognized as having such in SQL. For example, SELECT is a reserved word and has special meaning as a statement. It is important to remember that reserved words should not be used as variable or dictionary names.

This appendix contains the following topic:

- “List of Reserved Words” on page B-2
**List of Reserved Words**

<table>
<thead>
<tr>
<th>#</th>
<th>:</th>
<th>:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td>ACTION</td>
<td>ADD</td>
<td>AFTER</td>
</tr>
<tr>
<td>ALL</td>
<td>ALTER</td>
<td>AND</td>
<td>ANY</td>
</tr>
<tr>
<td>AS</td>
<td>ASC</td>
<td>ATOMIC</td>
<td>AUTHORIZATION</td>
</tr>
<tr>
<td>AVG</td>
<td>BEFORE</td>
<td>BEGIN</td>
<td>BETWEEN</td>
</tr>
<tr>
<td>BY</td>
<td>CALL</td>
<td>CASCADE</td>
<td>CASE</td>
</tr>
<tr>
<td>CHECK</td>
<td>CLOSE</td>
<td>COBOL</td>
<td>COLLATE</td>
</tr>
<tr>
<td>COLUMN</td>
<td>COMMIT</td>
<td>COMMITTED</td>
<td>CONSTRAINT</td>
</tr>
<tr>
<td>CONVERT</td>
<td>COUNT</td>
<td>CREATE</td>
<td>CREATETAB</td>
</tr>
<tr>
<td>CROSS</td>
<td>CS</td>
<td>CURRENT</td>
<td>CURSOR</td>
</tr>
<tr>
<td>D</td>
<td>DECLARE</td>
<td>DEFAULT</td>
<td>DELETE</td>
</tr>
<tr>
<td>DESC</td>
<td>DIAGNOSTICS</td>
<td>DISTINCT</td>
<td>DO</td>
</tr>
<tr>
<td>DROP</td>
<td>EACH</td>
<td>ELSE</td>
<td>END</td>
</tr>
<tr>
<td>ESCAPE</td>
<td>EX</td>
<td>EXISTS</td>
<td>FETCH</td>
</tr>
<tr>
<td>FOR</td>
<td>FOREIGN</td>
<td>FROM</td>
<td>FULL</td>
</tr>
<tr>
<td>GRANT</td>
<td>GROUP</td>
<td>HANDLER</td>
<td>HAVING</td>
</tr>
<tr>
<td>IF</td>
<td>IN</td>
<td>INDEX</td>
<td>INNER</td>
</tr>
<tr>
<td>INOUT</td>
<td>INSERT</td>
<td>INTERNAL</td>
<td>INTO</td>
</tr>
<tr>
<td>IS</td>
<td>ISOLATION</td>
<td>JOIN</td>
<td>KEY</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td>LEAVE</td>
<td>LEFT</td>
<td>LEVEL</td>
</tr>
<tr>
<td>LIKE</td>
<td>LOGIN</td>
<td>LOOP</td>
<td>MAX</td>
</tr>
<tr>
<td>MIN</td>
<td>MODIFIABLE</td>
<td>MODULE</td>
<td>NEW</td>
</tr>
<tr>
<td>NEXT</td>
<td>NO</td>
<td>NOT</td>
<td>NULL</td>
</tr>
<tr>
<td>OF</td>
<td>OLD</td>
<td>ON</td>
<td>ONLY</td>
</tr>
</tbody>
</table>
### Table B-1  SQL Reserved Words and Symbols

<table>
<thead>
<tr>
<th>OPEN</th>
<th>OPTION</th>
<th>OR</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT</td>
<td>OUTER</td>
<td>PRIMARY</td>
<td>PRINT</td>
</tr>
<tr>
<td>PRIVILEGES</td>
<td>PROCEDURE</td>
<td>PUBLIC</td>
<td>READ</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>REFERENCING</td>
<td>RELEASE</td>
<td>REPEAT</td>
</tr>
<tr>
<td>REPEATABLE</td>
<td>RESTRICT</td>
<td>RETURN</td>
<td>RETURNS</td>
</tr>
<tr>
<td>REVOKE</td>
<td>RIGHT</td>
<td>ROLLBACK</td>
<td>ROQ</td>
</tr>
<tr>
<td>SAVEPOINT</td>
<td>SCHEMA</td>
<td>SECURITY</td>
<td>SELECT</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>SET</td>
<td>SIGNAL</td>
<td>SIZE</td>
</tr>
<tr>
<td>SQLSTATE</td>
<td>SSP_EXPRE</td>
<td>SSP_PRED</td>
<td>START</td>
</tr>
<tr>
<td>SUM</td>
<td>SVBEGIN</td>
<td>SVEND</td>
<td>TABLE</td>
</tr>
<tr>
<td>THEN</td>
<td>TO</td>
<td>TRANSACTION</td>
<td>TRIGGER</td>
</tr>
<tr>
<td>TS</td>
<td>UNCOMMITTED</td>
<td>UNION</td>
<td>UNIQUE</td>
</tr>
<tr>
<td>UNTIL</td>
<td>UPDATE</td>
<td>USER</td>
<td>USING</td>
</tr>
<tr>
<td>VALUES</td>
<td>VIEW</td>
<td>WHEN</td>
<td>WHERE</td>
</tr>
<tr>
<td>WHILE</td>
<td>WITH</td>
<td>WORK</td>
<td>WRITE</td>
</tr>
</tbody>
</table>
SQL Reserved Words
This appendix contains tables of Pervasive.SQL and ODBC functions. The tables represent the following categories of functions:

- “Connection and Session Control APIs” on page C-2
- “Preparing and Executing SQL Request APIs” on page C-3
- “Data Retrieval APIs” on page C-8
- “Statement Termination APIs” on page C-8
- “Database and Driver Information APIs” on page C-8
- “Metadata Information APIs” on page C-8
- “Transaction APIs” on page C-9
- “Deprecated Scalable SQL APIs” on page C-9
## SQL API to ODBC Mapping Tables

### Table C-1  Connection and Session Control APIs

<table>
<thead>
<tr>
<th>Deprecated Scalable SQL API</th>
<th>Recommended ODBC API</th>
</tr>
</thead>
</table>
| XQLLogin                    | 1. SQLAllocHandle or SQLAllocEnv for environment handles.  
|                             | 2. SQLSetEnvAttr (Optional) to set driver attributes.  
|                             | 3. SQLAllocHandle or SQLAllocConnect for a connection handle.  
|                             | 4. SQLSetConnectAttr (Optional) to set connection attributes.  
|                             | 5. SQLConnect or SQLBrowseConnect or SQLDriverConnect to make a connection. |
| XQLLogout                   | 1. SQLDisconnect to break a connection.  
|                             | 2. SQLFreeHandle or SQLFreeConnect to release connection handle.  
|                             | 3. SQLFreeHandle or SQLFreeEnv to release an environment handle. |
| XGetSessionID               | You will have a session ID once you are connected to your database. This is the Connection handle, allocated and connected during login. |
| XPutSessionID               | You will not need to put session because all connections you make are available concurrently. |
### Table C.2  Preparing and Executing SQL Request APIs

<table>
<thead>
<tr>
<th>Deprecated Scalable SQL API</th>
<th>Recommended ODBC API</th>
</tr>
</thead>
</table>
| XQLCursor                   | 1. SQLAllocHandle or SQLAllocStmt to allocate a statement handle.  
|                             | 2. SQLSetCursorName may be used to associate a name with the cursor handle. |
| XQLCompile                  | SQLExecDirect or SQLEexecute |
| XQLSubst                    | SQLPrepare, SQLBindParameter, SQLParamOptions, SQLPutData |
| XQLExec                     | SQLExecDirect or SQLEexecute |
| xAccess                     | 1. Grants privileges: no direct ODBC API, but available through the GRANT statement.  
|                             | 2. Revokes privileges: no direct ODBC API, but available through the REVOKE statement.  
|                             | 3. Fetch privileges: SQLTablePrivileges and SQLColumnPrivileges are the associated ODBC APIs. The Master user also has the option of using a SELECT statement to retrieve the privileges that each user has. |
SQL API Mapping to ODBC

Table C-2 Preparing and Executing SQL Request APIs

<table>
<thead>
<tr>
<th>Deprecated Scalable SQL API</th>
<th>Recommended ODBC API</th>
</tr>
</thead>
<tbody>
<tr>
<td>xChar</td>
<td>1. Define NULL values for Scalable SQL data types: no ODBC API and not available through SQL. This is applicable for legacy databases and applications, where the application made use of some value other than the default NULL values. That is, legacy NULL behavior.</td>
</tr>
<tr>
<td></td>
<td>2. Retrieve the blank replacement char, or one of the null values: no ODBC API, and not available through SQL. Null values are applicable for legacy databases and applications, where the application made use of some non-default value. Blank replacement char is no longer applicable since dictionary names with blanks can be accessed by putting double quotes around them.</td>
</tr>
</tbody>
</table>
Table C-2 Preparing and Executing SQL Request APIs

<table>
<thead>
<tr>
<th>Deprecated Scalable SQL API</th>
<th>Recommended ODBC API</th>
</tr>
</thead>
<tbody>
<tr>
<td>xDD</td>
<td>1. Create a dictionary: no ODBC API, and not available through SQL. With Pervasive.SQL 2000, we eliminated support for standalone dictionaries. Dictionaries can only be accessed through named databases. The ability to create a named database programatically is available through the Distributed Tuning Interface (DTI), which is documented in the Pervasive.SQL Software Development Kit. The DTI allows creating named databases AND the dictionary files.</td>
</tr>
<tr>
<td></td>
<td>2. Remove an existing dictionary: no ODBC API, and not available through SQL. DTI will also allows deleting a named database AND the dictionary files, and then the named database can be re-added. This causes a delete of the dictionary files only.</td>
</tr>
<tr>
<td></td>
<td>3. Replace an existing dictionary: No ODBC API, not available through SQL. You can delete the named database and the dictionary files, and then recreate both, causing a replacement of the dictionary.</td>
</tr>
</tbody>
</table>
SQL API Mapping to ODBC

Table C-2  Preparing and Executing SQL Request APIs

<table>
<thead>
<tr>
<th>Deprecated Scalable SQL API</th>
<th>Recommended ODBC API</th>
</tr>
</thead>
<tbody>
<tr>
<td>xDDAttr</td>
<td></td>
</tr>
</tbody>
</table>

1. Masks and Headings (Add, Modify, Remove, Fetch): no ODBC API, and not available through SQL. Through the ODBC API, data is transferred in common formats, and even conversions from one data type to another are well defined behaviorally. This means that masks and headings are no longer necessary, or appropriate.

2. Other attributes (Add, Modify, Remove, Fetch): no ODBC API, and not available through SQL. The existing attributes are enforced with Pervasive.SQL, but they cannot be changed or removed. New default values can be defined with SQL, but no check constraints (or character lists, value lists, range lists, etc.) can be defined.
### Table C-2  Preparing and Executing SQL Request APIs

<table>
<thead>
<tr>
<th>Deprecated Scalable SQL API</th>
<th>Recommended ODBC API</th>
</tr>
</thead>
<tbody>
<tr>
<td>xRemove</td>
<td>1. This API is partially supported through the ODBC Cursor Library support for SQLSetPos, SQLExtendedFetch, and ODBC named cursors. Using DELETE ... WHERE CURRENT OF &lt;cursor name&gt;, an application can remove the current record based on the positioning within a cursor.</td>
</tr>
</tbody>
</table>
| xUser                       | 1. Add a user/group: no ODBC API, available through the GRANT LOGIN or CREATE GROUP statement in SQL.  
2. Drop a user/group: no ODBC API, available through the REVOKE LOGIN or DROP GROUP statement in SQL.  
3. Retrieve a list of users: the Master user can query the "$User" system table to find all users, what group each is in (if any), whether the user is a group instead of a user, and whether the user/group has the create table privilege.  
4. Grant/Revoke the "create table" privilege: no ODBC API, available through the GRANT/REVOKE CREATETAB statements. |
### SQL API Mapping to ODBC

#### Table C-3  Data Retrieval APIs

<table>
<thead>
<tr>
<th>Scalable SQL</th>
<th>ODBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>XQLFetch</td>
<td>SQLFetch, SQLExtendedFetch, or SQLGetData</td>
</tr>
<tr>
<td>XQLDescribe</td>
<td>SQLDescribeCol, SQLColAttributes or SQLColAttribute</td>
</tr>
<tr>
<td>XQLStatus</td>
<td>SQLError</td>
</tr>
<tr>
<td>xFetch</td>
<td>SQLFetch, SQLExtendedFetch</td>
</tr>
<tr>
<td>xStatus</td>
<td>SQLError</td>
</tr>
</tbody>
</table>

#### Table C-4  Statement Termination APIs

<table>
<thead>
<tr>
<th>Scalable SQL</th>
<th>ODBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>xReset</td>
<td>SQLFreeStmt, SQLCancel, SQLTransact</td>
</tr>
<tr>
<td>XQLFree</td>
<td>SQLFreeHandle</td>
</tr>
</tbody>
</table>

#### Table C-5  Database and Driver Information APIs

<table>
<thead>
<tr>
<th>Scalable SQL</th>
<th>ODBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLVersion</td>
<td>SQLGetInfo</td>
</tr>
</tbody>
</table>

#### Table C-6  Metadata Information APIs

<table>
<thead>
<tr>
<th>Scalable SQL</th>
<th>ODBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>xDDField</td>
<td>SQLColAttribute</td>
</tr>
<tr>
<td>xDDFile</td>
<td>SQLTables</td>
</tr>
<tr>
<td>xDDIndex</td>
<td>SQLStatistics</td>
</tr>
</tbody>
</table>
### Table C-7  Transaction APIs

<table>
<thead>
<tr>
<th>Scalable SQL</th>
<th>ODBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Transaction</td>
<td>SQLSetConnectAttr</td>
</tr>
<tr>
<td>Commit</td>
<td>SQLTransAct</td>
</tr>
<tr>
<td>Rollback</td>
<td>SQLTransAct</td>
</tr>
</tbody>
</table>

### Table C-8  Deprecated Scalable SQL APIs

<table>
<thead>
<tr>
<th>Scalable SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>xInsert</td>
</tr>
<tr>
<td>xPassword</td>
</tr>
<tr>
<td>xUpdate</td>
</tr>
<tr>
<td>XShareSessionID</td>
</tr>
<tr>
<td>XQLFormat</td>
</tr>
<tr>
<td>XQLMask</td>
</tr>
<tr>
<td>XQLConvert</td>
</tr>
<tr>
<td>SQLGetCountDatabaseNames</td>
</tr>
<tr>
<td>SQLGetCountRemoteDatabaseNames</td>
</tr>
<tr>
<td>SQLGetRemoteDatabaseNames</td>
</tr>
<tr>
<td>SQLUnloadDBnames</td>
</tr>
<tr>
<td>XQLValidate</td>
</tr>
</tbody>
</table>
This appendix describes the Pervasive.SQL system tables. For each system table, the following table indicates the name of the associated file and briefly describes the system table’s contents.

**Note** Some data in the system tables cannot be displayed. For example, information about stored views and procedures, other than their names, is available only to Pervasive.SQL. In addition, some data (such as user passwords) displays in encrypted form.

<table>
<thead>
<tr>
<th>System Table</th>
<th>Dictionary File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>X$File</td>
<td>FILE.DDF</td>
<td>Names and locations of the tables in your database.</td>
</tr>
<tr>
<td>X$Field</td>
<td>FIELD.DDF</td>
<td>Column and named index definitions.</td>
</tr>
<tr>
<td>X$Index</td>
<td>INDEX.DDF</td>
<td>Index definitions.</td>
</tr>
<tr>
<td>X$Attrib</td>
<td>ATTRIB.DDF</td>
<td>Column attributes definitions.</td>
</tr>
<tr>
<td>X$View</td>
<td>VIEW.DDF</td>
<td>View definitions.</td>
</tr>
<tr>
<td>X$Proc</td>
<td>PROC.DDF</td>
<td>Stored procedure definitions.</td>
</tr>
<tr>
<td>X$User</td>
<td>USER.DDF</td>
<td>User names, group names, and passwords.</td>
</tr>
<tr>
<td>X$Rights</td>
<td>RIGHTS.DDF</td>
<td>User and group access rights definitions.</td>
</tr>
<tr>
<td>X$Relate</td>
<td>RELATE.DDF</td>
<td>Referential integrity (RI) information.</td>
</tr>
</tbody>
</table>
When you issue a CREATE DICTIONARY statement, Pervasive.SQL creates the X$File, X$Field, and X$Index system tables and the associated dictionary files. Pervasive.SQL creates the other system tables as follows:

- **X$Attrib**—When you define column attributes, Pervasive.SQL creates this table and stores the definitions.
- **X$View**—When you define views, Pervasive.SQL creates this table and stores the definitions.
- **X$Proc**—When you define stored procedures, Pervasive.SQL creates this table and stores the definitions.
- **X$User and X$Rights**—When you set up data security on the database, Pervasive.SQL creates these two tables. In X$User, Pervasive.SQL stores information about user names, group names, and passwords. In X$Rights, Pervasive.SQL stores information about the access rights assigned to users and groups. When you disable security, Pervasive.SQL deletes these two tables.
- **X$Relate**—When you define RI constraints for the database, Pervasive.SQL creates this table and stores information about foreign key references.
- **X$Trigger and X$Depend**—When you define triggers for tables in the database, Pervasive.SQL creates these two tables. In X$Trigger, Pervasive.SQL stores information about the triggers. In X$Depend, Pervasive.SQL stores information about the trigger dependencies.

Because the system tables are part of the database, you can query them to retrieve information about the database. However, to update the system tables, you must use data definition statements. You cannot update them with data manipulation statements as you would standard data tables; this may corrupt the dictionary.

<table>
<thead>
<tr>
<th>System Table</th>
<th>Dictionary File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>X$Trigger</td>
<td>TRIGGER.DDF</td>
<td>Trigger information.</td>
</tr>
<tr>
<td>X$Depend</td>
<td>DEPEND.DDF</td>
<td>Trigger dependencies such as tables, views, and procedures.</td>
</tr>
</tbody>
</table>

When you issue a CREATE DICTIONARY statement, Pervasive.SQL creates the X$File, X$Field, and X$Index system tables and the associated dictionary files. Pervasive.SQL creates the other system tables as follows:

- **X$Attrib**—When you define column attributes, Pervasive.SQL creates this table and stores the definitions.
- **X$View**—When you define views, Pervasive.SQL creates this table and stores the definitions.
- **X$Proc**—When you define stored procedures, Pervasive.SQL creates this table and stores the definitions.
- **X$User and X$Rights**—When you set up data security on the database, Pervasive.SQL creates these two tables. In X$User, Pervasive.SQL stores information about user names, group names, and passwords. In X$Rights, Pervasive.SQL stores information about the access rights assigned to users and groups. When you disable security, Pervasive.SQL deletes these two tables.
- **X$Relate**—When you define RI constraints for the database, Pervasive.SQL creates this table and stores information about foreign key references.
- **X$Trigger and X$Depend**—When you define triggers for tables in the database, Pervasive.SQL creates these two tables. In X$Trigger, Pervasive.SQL stores information about the triggers. In X$Depend, Pervasive.SQL stores information about the trigger dependencies.

Because the system tables are part of the database, you can query them to retrieve information about the database. However, to update the system tables, you must use data definition statements. You cannot update them with data manipulation statements as you would standard data tables; this may corrupt the dictionary.
Installing System Tables and Data Dictionary Files

The system tables included with Pervasive.SQL contain the data dictionary files for the sample database. When you install Pervasive.SQL, you can copy this data dictionary to the appropriate device on your system and log in to the sample database. After logging in, you can create a new data dictionary in another directory of your choice. Alternatively, you can create a new data dictionary using the Configuration utility in the Pervasive Control Center (PCC) to define a bound or unbound named database.
The X$File system table is associated with the file FILE.DDF. For each table defined in the database, X$File contains the table name, the location of the associated table, and a unique internal ID number that Pervasive.SQL assigns. The structure of X$File is as follows:

**Table D-2  X$File System Table Structure**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xf$Id</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Internal ID Pervasive.SQL assigns.</td>
</tr>
<tr>
<td>Xf$Name</td>
<td>CHAR</td>
<td>20</td>
<td>Yes</td>
<td>Table name.</td>
</tr>
<tr>
<td>Xf$Loc</td>
<td>CHAR</td>
<td>64</td>
<td>No</td>
<td>File location (pathname).</td>
</tr>
<tr>
<td>Xf$Flags</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>File flags. If bit 4=1, the file is a dictionary file. If bit 4=0, the file is user-defined. If bit 6=1, the table supports true nullable columns.</td>
</tr>
<tr>
<td>Xf$Reserved</td>
<td>CHAR</td>
<td>10</td>
<td>No</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

Two indexes are defined for the X$File table.

**Table D-3  X$File System Table Index Definitions**

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xf$Id</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Xf$Name</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note** Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
The X$Field system table is associated with the file FIELD.DDF. X$Field contains information about all the columns and named indexes defined in the database. The structure of X$Field is as follows:

Table D-4  **X$Field System Table Structure**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xe$Id</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Internal ID Pervasive.SQL assigns, unique for each field in the database.</td>
</tr>
<tr>
<td>Xe$File</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>ID of table to which this column or named index belongs. It corresponds to</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Xf$Id in X$File.</td>
</tr>
<tr>
<td>Xe$Name</td>
<td>CHAR</td>
<td>20</td>
<td>Yes</td>
<td>Column name or index name.</td>
</tr>
<tr>
<td>Xe$DataType</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>Control field - Column data type (range 0–26). If value is 227, it represents</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a constraint name. If value is 255, it represents an index name.</td>
</tr>
<tr>
<td>Xe$Offset</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Column offset in table; index number if named index. Offsets are zero-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>relative.</td>
</tr>
<tr>
<td>Xe$Size</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Column size, representing the internal storage, in bytes, required for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>field.</td>
</tr>
</tbody>
</table>
The column Xe$File corresponds to the column Xf$Id in the X$File system table and is the link between the tables and the columns they contain. So, for example, the following query shows you all of the field definitions in order for the Billing table.

```
SELECT "X$Field".*
FROM X$File,X$Field
WHERE Xf$Id=Xe$File AND Xf$Name = 'Billing'
ORDER BY Xe$Offset
```

The integer values in column Xe$DataType are codes that represent the Pervasive.SQL data types. See “Data Types” on page A-1 for the codes.
Five indexes are defined for the X$Field table, as follows:

Table D-5  X$Field System Table Index Definitions

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X$Id</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X$File</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>X$Name</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>X$File</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>X$Name</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>X$File</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>X$Offset</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>X$Dec</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
</tbody>
</table>

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
**X$Index**

The X$Index system table is associated with the file INDEX.DDF. X$Index contains information about all the indexes defined on the tables in the database. The structure of X$Index is as follows:

*Table D-6  X$Index System Table Structure*

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X$i$File</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Unique ID of the table to which the index belongs. It corresponds to Xf$I$Id in X$File.</td>
</tr>
<tr>
<td>X$i$Field</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Unique ID of the index column. It corresponds to Xe$I$Id in X$Field.</td>
</tr>
<tr>
<td>X$i$Number</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Index number (range 0–119).</td>
</tr>
<tr>
<td>X$i$Part</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Segment number (range 0–119).</td>
</tr>
<tr>
<td>X$i$Flags</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Index attribute flags.</td>
</tr>
</tbody>
</table>

The X$i$File column corresponds to the Xf$I$Id column in the X$File system table. The X$i$Field column corresponds to the Xe$I$Id column in the X$Field system table. Thus, an index segment entry is linked to a file and to a field.

The X$i$Flags column contains integer values that define the index attributes. The following table describes how Pervasive.SQL interprets each bit position when the bit has the binary value of 1. Bit position 0 is the rightmost bit in the integer.

*Table D-7  X$Index System Table Index Definitions*

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Decimal Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Index allows duplicates.</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Index is modifiable.</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Indicates an alternate collating sequence.</td>
</tr>
</tbody>
</table>
The value in the Xi$Flags column for a particular index is the sum of the decimal values that correspond to that index’s attributes. Three indexes are defined for the X$Index table, as follows:

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Decimal Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>8</td>
<td>Null values are not indexed (refers to Btrieve NULLs, not SQL true NULLs).</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>Another segment is concatenated to this one in the index.</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>Index is case-insensitive.</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>Index is collated in descending order.</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>Index is a named index.</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>Index is a Btrieve extended key type.</td>
</tr>
<tr>
<td>13</td>
<td>8192</td>
<td>Index is a foreign key.</td>
</tr>
<tr>
<td>14</td>
<td>16384</td>
<td>Index is a primary key referenced by some foreign key.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xi$File</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Xi$Field</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Xi$File</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Xi$Number</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Xi$Part</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
</tbody>
</table>

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
To see the information about the index segments defined for the Billing table, for example, issue the following query:

```sql
SELECT Xe$Name, Xe$Offset, "X$Index".*
FROM X$File, X$Index, X$Field
WHERE Xf$Id=Xi$File and Xi$Field=Xe$Id and Xf$Name = 'Billing'
ORDER BY Xi$Number, Xi$Part
```
X$Attrib

The X$Attrib system table is associated with the file ATTRIB.DDF. X$Attrib contains information about the column attributes of each column in the database; there is an entry for each column attribute you define. The structure of X$Attrib is as follows:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xa$Id</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Corresponds to Xe$Id in X$Field.</td>
</tr>
<tr>
<td>Xa$Type</td>
<td>CHAR</td>
<td>1</td>
<td>No</td>
<td>C for character, D for default, H for heading, M for mask, O for column collation, R for range, or V for value.</td>
</tr>
<tr>
<td>Xa$ASize</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Length of text in Xa$Attrib.</td>
</tr>
<tr>
<td>Xa$Attrs</td>
<td>LONGVARCHAR</td>
<td>&lt;=20</td>
<td>N/A</td>
<td>Text that defines the column attribute.</td>
</tr>
</tbody>
</table>

When you define multiple attributes for a single column, the X$Attrib system table contains multiple entries for that column ID—one for each attribute you define. If you do not define column attributes for a particular column, that column has no entry in the X$Attrib table. The text in the Xa$Attrs column appears exactly as you define it with Pervasive.SQL. One index is defined for the X$Attrib table, as follows:

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xa$Id</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Xa$Type</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.

**Note** Attribute type C, H, M, R and V are legacy validation types valid only in a Pervasive.SQL 7 or Scalable SQL environment. Pervasive.SQL 2000i uses only the D (default) and O (column collation) attributes.
The X$View system table is associated with the file VIEW.DDF. X$View contains view definitions, including information about joined tables and the restriction conditions that define views. You can query the X$View table to retrieve the names of the views that are defined in the dictionary.

The first column of the X$View table contains the view name; the second and third columns describe the information found in the LVAR column, Xv$Misc. The structure of X$View is as follows:

### Table D-11 X$View System Table Structure

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xv$Name</td>
<td>CHAR</td>
<td>20</td>
<td>Yes</td>
<td>View name.</td>
</tr>
<tr>
<td>Xv$Ver</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>Version ID.</td>
</tr>
<tr>
<td>Xv$Id</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>Sequence number.</td>
</tr>
<tr>
<td>Xv$Misc</td>
<td>LONGVARCHAR</td>
<td>&lt;=2000</td>
<td>N/A</td>
<td>Pervasive.SQL internal definitions.</td>
</tr>
</tbody>
</table>

Two indexes are defined for the X$View table, as follows:

### Table D-12 X$View System Table Index Definitions

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xv$Name</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Xv$Name</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Xv$Ver</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Xv$Id</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
</tbody>
</table>

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
**X$Proc**

The X$Proc system table is associated with the file PROC.DDF. X$Proc contains the compiled structure information for every stored procedure defined. The structure of X$Proc is as follows:

*Note* Stored procedures and external procedures were supported in versions prior versions of Pervasive.SQL. Only stored procedures are supported in Pervasive.SQL 2000i.

One index is defined for the X$Proc table, as follows:

---

**Table D-13 X$Proc System Table Structure**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xp$Name</td>
<td>CHAR</td>
<td>30</td>
<td>Yes</td>
<td>Stored procedure name.</td>
</tr>
<tr>
<td>Xp$Ver</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>Version ID.</td>
</tr>
<tr>
<td>Xp$Id</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>0-based Sequence Number.</td>
</tr>
<tr>
<td>Xp$Flags</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>1 for stored statement, 2 for stored procedure or 3 for external procedure.</td>
</tr>
<tr>
<td>Xp$Misc</td>
<td>LONGVARCHAR (LVAR)</td>
<td>990</td>
<td>N/A</td>
<td>Internal representation of stored procedure.</td>
</tr>
</tbody>
</table>

---

**Table D-14 X$Proc System Table Index Definitions**

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xp$Name</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Xp$Id</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
</tbody>
</table>

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
### X$User

The X$User system table is associated with the file USER.DDF. X$User contains the name and password of each user and the name of each user group. Pervasive.SQL uses this table only when you enable the security option. The structure of X$User is as follows:

**Table D-15  X$User System Table Structure**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xu$Id</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Internal ID assigned to the user or group.</td>
</tr>
<tr>
<td>Xu$Name</td>
<td>CHAR</td>
<td>30</td>
<td>Yes</td>
<td>User or group name.</td>
</tr>
<tr>
<td>Xu$Password</td>
<td>CHAR</td>
<td>9</td>
<td>No</td>
<td>User password (encrypted)</td>
</tr>
<tr>
<td>Xu$Flags</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>User or group flags.</td>
</tr>
</tbody>
</table>

**Note** For any row in the X$User system table that describes a group, the column value for Xu$Password is NULL.
The Xu$Flags column contains integer values whose rightmost 8 bits define the user or group attributes. The following table describes how Pervasive.SQL interprets each bit position when the bit has the binary value of 1. Bit position 0 is the rightmost bit in the integer.

Table D-16  Xin$Flags System Table Bit Position Definitions

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Decimal Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Reserved.</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Reserved.</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Reserved.</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>Reserved.</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>Reserved.</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>Reserved.</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>Name is a group name.</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>User or group has the right to define tables in the dictionary.</td>
</tr>
</tbody>
</table>

The value in the Xu$Flags column for a particular user or group is the sum of the decimal values corresponding to the attributes that apply to the user or group.

Two indexes are defined for the Xin$User table, as follows:

Table D-17  Xin$User System Table Index Definitions

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xin$Id</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Xin$Name</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Index Number corresponds to the value stored in the Xin$Number column in the Xin$Index system table. Segment Number corresponds to the value stored in the Xin$Part column in the Xin$Index system table.
The X$Rights system table is associated with the file RIGHTS.DDF. X$Rights contains access rights information for each user. Pervasive.SQL uses this table only when you enable the security option. The structure of X$Rights is as follows:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xr$User</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>User ID</td>
</tr>
<tr>
<td>Xr$Table</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Table ID</td>
</tr>
<tr>
<td>Xr$Column</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Column ID</td>
</tr>
<tr>
<td>Xr$Rights</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>Table or column rights flag</td>
</tr>
</tbody>
</table>

The Xr$User column corresponds to the Xu$Id column in the X$User table. The Xr$Table column corresponds to the Xf$Id column in the X$File table. The Xr$Column column corresponds to the Xe$Id column in the X$Field table.

**Note** For any row in the system table that describes table rights, the value for Xr$Column is null.

The Xr$Rights column contains integer values whose rightmost 8 bits define the users' access rights. The following table describes how Pervasive.SQL interprets the value. Values from this table may be combined into a single Xr$Rights value.
A decimal equivalent of 0 implies no rights.

The value in the XSRights column for a particular user is the bitwise or of the hex values corresponding to the access rights that apply to the user.

Three indexes are defined for the XSRights table, as follows:

### Table D-20 XSRights System Table Index Definitions

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>XSRUser</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>XSRUser</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>XSRTable</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>XSRColumn</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>XSRTable</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>XSRColumn</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
</tbody>
</table>

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
X$Relate

The X$Relate system table is associated with the file RELATE.DDF. X$Relate contains information about the referential integrity (RI) constraints defined on the database. X$Relate is automatically created when the first foreign key is created, since this results in a relationship being defined.

The structure of X$Relate is as follows:

Table D-21  X$Relate System Table Structure

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>X$r$PId</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Primary table ID.</td>
</tr>
<tr>
<td>X$r$PIndex</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Index number of primary key in primary table.</td>
</tr>
<tr>
<td>X$r$FId</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Dependent table ID.</td>
</tr>
<tr>
<td>X$r$FIndex</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Index number of foreign key in dependent table.</td>
</tr>
<tr>
<td>X$r$Name</td>
<td>CHAR</td>
<td>20</td>
<td>Yes</td>
<td>Foreign key name.</td>
</tr>
<tr>
<td>X$r$UpdateRule</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>1 for restrict.</td>
</tr>
<tr>
<td>X$r$DeleteRule</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>1 for restrict, 2 for cascade.</td>
</tr>
<tr>
<td>X$r$Reserved</td>
<td>CHAR</td>
<td>30</td>
<td>No</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
Five indexes are defined for the X$Relate table, as follows:

Table D-22  X$Relate System Table Index Definitions

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xr$Pld</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Xr$Fld</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Xr$Name</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Xr$Pld</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Xr$Name</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>Xr$Fld</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Xr$Name</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
The X$Trigger system table is associated with the file TRIGGER.DDF. X$Trigger contains information about the triggers defined for the database. The structure of X$Trigger is as follows:

**Table D-23  X$Trigger System Table Structure**

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Type</th>
<th>Size</th>
<th>Case Insensitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xt$Name</td>
<td>CHAR</td>
<td>30</td>
<td>Yes</td>
<td>Trigger name.</td>
</tr>
<tr>
<td>Xt$Version</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Trigger version. A 4 indicates Scalable SQL v4.</td>
</tr>
<tr>
<td>Xt$File</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>File on which trigger is defined.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Corresponds to Xt$Id in X$File.</td>
</tr>
<tr>
<td>Xt$Event</td>
<td>UNSIGNED</td>
<td>1</td>
<td>N/A</td>
<td>0 for INSERT, 1 for DELETE, 2 for UPDATE.</td>
</tr>
<tr>
<td>Xt$ActionTime</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>0 for BEFORE, 1 for AFTER.</td>
</tr>
<tr>
<td>Xt$ForEach</td>
<td>UTINYINT</td>
<td>1</td>
<td>N/A</td>
<td>0 for ROW (default), 1 for STATEMENT.</td>
</tr>
<tr>
<td>Xt$Order</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>Order of execution of trigger.</td>
</tr>
<tr>
<td>Xt$Sequence</td>
<td>USMALLINT</td>
<td>2</td>
<td>N/A</td>
<td>0-based sequence number.</td>
</tr>
<tr>
<td>Xt$Misc</td>
<td>LONGVARCHAR</td>
<td>&lt;=4054</td>
<td>N/A</td>
<td>Internal representation of trigger.</td>
</tr>
</tbody>
</table>

A given trigger may require multiple entries in Trigger.DDF. Each entry has the same trigger name in the Xt$Name field, and is used in the order specified by the Xt$Sequence field.
Three indexes are defined for the X$Trigger table, as follows:

**Table D-24 X$Trigger System Table Index Definitions**

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xt$Name</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Xt$Sequence</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Xt$File</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Xt$Name</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>Xt$File</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Xt$Event</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Xt$ActionTime</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Xt$ForEach</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Xt$Order</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Xt$Sequence</td>
<td>Yes</td>
<td>N/A</td>
<td>No</td>
</tr>
</tbody>
</table>

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
The X$Depend system table is associated with the file DEPEND.DDF. X$Depend contains information about trigger dependencies such as tables, views, and procedures. The structure of X$Depend is as follows:

**Table D-25  X$Depend System Table Structure**

| Column Name      | Type    | Size | Case Insensitive | Description                                                                 |
|------------------|---------|------|------------------|                                                                            |
| Xd$Trigger       | CHAR    | 30   | Yes              | Name of trigger. It corresponds to Xf$Name in X$File.                     |
| Xd$DependType    | UNSIGNED| 1    | N/A              | 1 for Table, 2 for View, 3 for Procedure.                                 |
| Xd$DependName    | CHAR    | 30   | Yes              | Name of dependency with which the trigger is associated. It corresponds to either Xf$Name in X$File, Xv$Name in X$View, or Xp$Name in X$Proc. |

Two indexes are defined for the X$Depend table, as follows:

**Table D-26  X$Depend System Table Index Definitions**

<table>
<thead>
<tr>
<th>Index Number</th>
<th>Segment Number</th>
<th>Column Name</th>
<th>Duplicates</th>
<th>Case Insensitive</th>
<th>Segmented</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Xd$Trigger</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Xd$DependType</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>Xd$DependName</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Xd$DependType</td>
<td>Yes</td>
<td>N/A</td>
<td>Yes</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Xd$DependName</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
System Tables

Index Number corresponds to the value stored in the Xi$Number column in the X$Index system table. Segment Number corresponds to the value stored in the Xi$Part column in the X$Index system table.
Index

Symbols
@@IDENTITY global variable 2-13
   Example of 2-13
@@ROWCOUNT global variable 2-14
   Example of 2-15

A
ADD
   grammar supported 2-17
ALL
   grammar supported 2-18
ALTER TABLE
   grammar supported 2-19
   IN DICTIONARY keyword 2-20
   USING keyword 2-22
   WITH REPLACE keyword 2-24
ANDed predicate
   maximum in SQL statement 2-2
ANY
   grammar supported 2-30
API functions
   ODBC supported 2-5
APIs
   connection and session control C-2
   Data retrieval C-8
   SQL request C-3
Approximate-numeric-literal
   grammar supported 2-121
Arguments
   number in a parameter list for stored procedures 2-2
AS
   grammar supported 2-31
ATTRIB.DDF D-1
   attributes ignored by Pervasive.SQL 2000i D-11
AUTOINCREMENT data type A-13

B
BEGIN
   grammar supported 2-32
   Between-predicate
   grammar supported 2-122
   BFLOAT data type A-15
   BINARY
      notes about A-5
   Bound databases
      IN DICTIONARY not permitted 2-20
   Btrieve data types A-12
   Btrieve Owner Names
      in secured databases 1-9

C
CALL
   calling a stored procedure 2-48
   grammar supported 2-33
CASCADE
   grammar supported 2-34
   ON DELETE 2-52
CASE
   grammar supported 2-35
Changing
   System Tables D-2
CHAR
   notes about A-5
Character translation
   OEM to ANSI 2-171
Clauses
   ON DELETE 2-52
   ON UPDATE 2-52
CLOSE
   grammar supported 2-36
Closing
   open tables 2-168
Codes, data type A-12
Columns
   attributes
      system table (X$Attrib) D-11
   limit 2-2
   maximum in a database 2-2
   maximum in a table 2-2
   maximum name length 2-2
   maximum number in a select list 2-2

Index 1
maximum size of 2-2
number allowed in a trigger 2-2
system table (X$Field) D-5
Comma
as decimal separator 2-169
COMMIT
grammar supported 2-37
Concurrency 2-168
Connection and session control
APIs C-2
Connection strings
TRANSLATIONDLL 2-172
Connection strings, DSN 2-3
Connections
maximum 2-2
Conversion functions supported 2-167
Correlated subquery 2-139
Correlation-name
grammar supported 2-122
COUNT( ), COUNT(*) functions
differences 2-45, 2-85
CREATE GROUP
grammar supported 2-39
CREATE INDEX
grammar supported 2-40
limitations 2-40
CREATE PROCEDURE
grammar supported 2-42
CREATE TABLE
grammar supported 2-50
IN DICTIONARY keyword 2-53
USING keyword 2-53
WITH REPLACE keyword 2-54
CREATE TRIGGER
grammar supported 2-59
CREATE VIEW
grammar supported 2-62
Creating
groups 1-10
stored procedures 2-48
tables with legacy null support 2-130
users and groups 2-81
views 1-8
CURDATE 2-162
CURRENCY data type A-15
Current Date
adding to an INSERT statement 2-91
Current Time
adding to an INSERT statement 2-91
Cursors
limits 2-49
CURTIME 2-162

D
Data
retrieval C-8
Data control statements 1-9
Data definition statements 1-3
Data dictionaries
list of system tables D-1
Data files
binary compatible cross-platform 2-23, 2-54
replacing existing 2-24, 2-54
system table (X$File) D-4
Data Manipulation
statements 1-6
Data Source Name connection string keyword 2-3
Data Source Name connection string keywords
clients 2-4
engine 2-4
Data Types
Btrieve A-12
code conversion A-3
lengths and ranges A-8
supported A-3
Data types
AUTOINCREMENT A-13
BFLOAT A-15
codes A-12
CURRENCY A-15
DATE A-16
DECIMAL A-16
extended A-12
FLOAT A-17
INTEGER A-18
LOGICAL A-18
LSTRING A-18
MONEY A-18
NUMERIC A-19
NUMERICSA A-19
NUMERICSTS A-20
STRING A-15

2 Index
Supported A-2
that cannot be indexed 2-41
TIME A-20
TIMESTAMP A-20
UNSIGNED BINARY A-21
WSTRING A-21
WZSTRING A-21
ZSTRING A-22
Database
maximum number of columns 2-2
Database names
valid characters 1-11
Date
adding in INSERT statements 2-91
Date arithmetic
grammar supported 2-125
DATE data type A-16
Date functions supported 2-162
Date-literal
grammar supported 2-123
DBQ entry in odbc.ini
length of 2-2
DECIMAL data type A-16
Decimal separator
comma as 2-169
DECLARE
grammar supported 2-64
DECLARE CURSOR
grammar supported 2-65
DELETE
grammar supported 2-66, 2-67
Delete
rule 2-52
cascade 2-52
restrict 2-52
Deleting
Views 1-8
Delimited identifiers
in SQL Statements 2-13
Delimiter
SQL statement in PCC 2-16
DEPEND.DDF D-2
Disabling security 1-9
DISTINCT 2-128
grammar supported 2-68
in subquery 2-141
DROP INDEX
grammar supported 2-70
DROP PROCEDURE
grammar supported 2-71
DROP TABLE
grammar supported 2-72
IN DICTIONARY keyword 2-72
DROP TRIGGER
grammar supported 2-73
DROP VIEW
grammar supported 2-74
DSN connection strings 2-3

E
Enabling security 1-9
END
grammar supported 2-75
Exact-numeric-literal
grammar supported 2-122
EXISTS
grammar supported 2-76
Expression
in stored procedures 2-155
Extended data types A-12

F
FETCH
grammar supported 2-77
Fetching
data C-8
FIELD.DDF D-1
FILE.DDF D-1
FLOAT data type A-17
FOREIGN KEY
cannot ALTER column 2-25
grammar supported 2-78
Functions
conversion 2-167
date 2-162
logical 2-165
numeric 2-159
string 2-156
system 2-165
time 2-162

Index 3
G
Global variables 2-13
@@IDENTITY 2-13
@@ROWCOUNT 2-14
Grammar Element Definitions 2-147
GRANT
  grammar supported 2-80
Granting
    rights 1-10
GROUP BY
  grammar supported 2-85
  with a HAVING clause 2-87
Grouped views 2-62
Groups
  creating 1-10

H
HAVING
  in a GROUP BY expression 2-87

I
IDENTITY global variable. See @@IDENTITY
IF
  grammar supported 2-88
IF (with SELECT)
  grammar supported 2-125
IN
  grammar supported 2-89
IN DICTIONARY keyword 2-20, 2-53, 2-72
  not permitted on bound databases 2-20
Index
  maximum indexed nullable columns 2-40
Index names
  maximum length of 2-2
INDEX.DDF D-1
Indexes
  creating 1-4
  data types that cannot be indexed 2-41
  dropping named 1-4
  system tables
    X$Field D-5
    X$Index D-8
Infinity
  representation A-10
In-predicate

4 Index
maximum size of quoted string in SQL statement 2-2
of Pervasive ODBC Engine interface 2-2
SQL variables and parameters 2-49
stored procedures 2-48
arguments in a parameter list 2-2
procedure name 2-48
variable name 2-48
triggers
number of columns 2-2
when using long data 2-49
Locale-specific behavior
comma as decimal separator 2-169
LOGICAL data type A-18
Logical functions supported 2-165
Long data
limits when using 2-49
LONGVARBINARY
limitation A-9
limitations in INSERT statements A-9
notes about A-5
using SQLGetdata A-9
LONGVARCHAR
limitation A-9
limitations in INSERT statements A-9
limitations in UNION statements 2-136
notes about A-5
using SQLGetdata A-9
LOOP
grammar supported 2-102
LSTRING data type A-18

M
Master user 2-81
Maximum
column name length 2-2
maximum length of view name 2-62
Modifying data
statements for 1-7
MONEY data type A-18

N
Name length
maximum
for columns 2-2
Named Database
and file names 2-22, 2-53
Non-correlated subquery 2-139
NOT
grammar supported 2-103
NULL
cannot make column nullable 2-25
inserted by UPDATE if subquery returns no rows 2-139
NULL support
setting 2-130
Null support
creating tables with legacy 2-130
Nullable columns
maximum number of indexed 2-40
NUMERIC data type A-19
Numeric functions supported 2-159
NUMERICSA data type A-19
NUMERICSTS data type A-20

O
ODBC
security 2-81
ODBC API conformance 2-5
exceptions to 2-7
ODBC API functions
supported 2-5
ODBC.ini
DBQ entry length 2-2
OEM to ANSI
character translation 2-171
connection string 2-172
ONLY
grammar supported 2-104
Owner name 2-81

P
Padding
in BINARY columns A-5
in CHAR columns A-5
in LONGVARBINARY columns A-5
in LONGVARCHAR columns A-5
in VARCHAR columns A-5
Parameter List
number of arguments for stored procedures 2-2
Password
for Master user 2-81

Index 5
table 2-9
Path name
   length in USING 2-22, 2-53
Permissions
   granting 1-10
   revoking 1-10
Pervasive Control Center
   OEM characters and 2-172
Pervasive ODBC Engine interface
   data types supported A-3
   limits of 2-2
   SQL conformance 2-11
Pervasive.SQL 7
   status code 59 2-54
Predicate
   in stored procedures 2-154
PRIMARY KEY
   cannot ALTER column 2-25
   grammar supported 2-105
PRINT
   grammar supported 2-108
PROC.DDF D-1
Procedures
   creating 2-48
PUBLIC
   grammar supported 2-107

Q
Quote mark, representing single 2-2
Quoted string in SQL statement
   maximum size 2-2

R
Real Infinity
   representation A-10
Referential integrity (RI)
   delete rules 2-52
   update rules 2-52
   $Relate system table D-19
Regional settings
   comma as decimal separator 2-169
RELATE.DDF D-1
Relational security 2-81
RELEASE SAVEPOINT
   grammar supported 2-109
Replacing data files 2-24, 2-54
   Reserved words B-1
   RESTRICT
      grammar supported 2-111
      ON DELETE 2-52
Retrieving data C-8
   about 1-6
   RETURNS
      example of in CREATE PROCEDURE 2-46
REVOKE
   grammar supported 2-112
   Revoking rights 1-10
   Rights
      granting 1-10
      revoking 1-10
   RIGHTS.DDF D-1
ROLLBACK WORK
   grammar supported 2-114
   ROWCOUNT global variable. See @@ROWCOUNT
   Rows
      limits 2-2

S
SAVEPOINT
   grammar supported 2-115
Scalar functions 2-156
   conversion 2-167
   date 2-162
   numeric 2-159
   string 2-156
   system 2-165
   time 2-162
Secured databases
   with Btrieve Owner Names 1-9
Security
   enabling and disabling 1-9
   owner name 2-81
   password of Master user 2-81
   relational 2-81
   setting 2-129
   system tables
      rights (XSRights) D-17
      users (X$User) D-15
SELECT
   grammar supported 2-118
   maximum number of columns 2-2
SELECT (with INTO)
grammar supported 2-117
Set Function
grammar supported in SELECT statements 2-123
SET SECURITY
grammar supported 2-129
SET TRUENULLSUPPORT
grammar supported 2-130
SET VARIABLE
grammar supported 2-131
SIGNAL
grammar supported 2-132
Single quote, representing 2-2
Sort order in keys
string A-12
Spacing
in BINARY columns A-5
in CHAR columns A-5
in LONGVARBINARY columns A-5
in LONGVARCHAR columns A-5
in VARCHAR columns A-5
SQL
description and purpose 1-1
extensions to standards B-1
request C-3
reserved words B-1
security 2-81
statement delimiter
changing in PCC 2-16
system tables. See System tables
SQL Data Manager
OEM characters and 2-172
SQL Grammar Elements 2-16
SQL Relational Database Engine. See SRDE
SQL Statement List 2-154
SQL Statements
data control 1-9
data definition 1-3
data manipulation 1-6
delimited identifiers 2-13
maximum length 2-2
maximum size of quoted string in 2-2
types of 1-1
SQL Variables
limits 2-49
parameters 2-49
SQLAllocConnect 2-5
SQLAllocEnv 2-5
SQLAllocStmt 2-5
SQLBindCol 2-5
SQLBindParameter 2-5
SQLBrowseConnect 2-5
SQLCancel 2-5
SQLColAttributes 2-5
SQLColumnPrivileges 2-5
SQLColumns 2-5
SQLConnect 2-5
SQLDataSources 2-5
SQLDescribeCol 2-5
SQLDescribeParam 2-5
SQLDisconnect 2-5
SQLDriverConnect 2-6
SQLDrivers 2-6
SQLError 2-6
SQLExecDirect 2-6
SQLExecute 2-6
SQLExtendedFetch 2-6
SQLFetch 2-6
SQLForeignKeys 2-6
SQLFreeConnect 2-6
SQLFreeEnv 2-6
SQLFreeStmt 2-6, 2-168
SQLGetConnectOption 2-6, 2-9
SQLGetCursorName 2-6
SQLGetData 2-6
SQLGetFunctions 2-6
SQLGetInfo 2-6
SQLGetStmtOption 2-6, 2-8
SQLGetTypeInfo 2-6, 2-10
SQLMoreResults 2-6, 2-7
SQLNativeSql 2-6
SQLNumParams 2-6
SQLNumResultCols 2-6
SQLParamData 2-6
SQLPrepare 2-6
SQLPrimaryKeys 2-6
SQLProcedureColumns 2-7
SQLProcedures 2-7
SQLPutData 2-7
SQLRowCount 2-7

Index 7
Index

Supported ODBC APIs
  in Pervasive.SQL 2000 2-5
System functions supported 2-165
System tables D-1
  updating D-2
  X$Attrib D-11
  X$Depend D-23
  X$Field D-5
  X$File D-4
  X$Index D-8
  X$Proc D-14
  X$Relate D-19
  X$Rights D-17
  X$Trigger D-21
  X$User D-15
  X$View D-13

T
Table
  locking 2-9
  maximum number of columns 2-2
  maximum number of rows 2-2
  password 2-9
Table names
  maximum length of 2-2
Tables
  closing in an SQL statement 2-168
  creating 1-3
  creating with legacy null support 2-130
  deleting 1-3
  modifying 1-3
  system D-1
  updating System D-2
Time
  adding in INSERT statements 2-91
TIME data type A-20
Time functions supported 2-162
Time-literal
  grammar supported 2-123
Timestamp
  adding to an INSERT statement 2-91
TIMESTAMP data type A-20
Timestamp-literal
  grammar supported 2-124
Transactions
  defining 1-7

8 Index
TRANSLATIONDLL connection string 2-172
TRIGGER.DDF D-2

Triggers
- calling 1-8
- creating 1-4
- dependencies
  - system table (X$Depend) D-23
- dropping 1-4
- executing 1-8
- number of columns allowed 2-2
- system table (X$Trigger) D-21

TRUENULLCREATE 2-130

U
- Unicode data types A-21
- Unicode key types A-12
- UNION
  - grammar supported 2-136
  - limitations 2-136
- UNIQUE
  - grammar supported 2-138
- UNSIGNED BINARY data type A-21
- UPDATE
  - fails if subquery returns multiple rows 2-139
  - grammar supported 2-139
- Update
  - rule 2-52
- UPDATE (positioned)
  - grammar supported 2-144
- Updating
  - system tables D-2
- User
  - Master 2-81
- User name
  - maximum length 2-2
- USER.DDF D-1
- Users and user groups. See Security
- USING keyword 2-22, 2-53
  - length of path name 2-22, 2-53

V
- Valid Value Range
  - data types A-8
- VARCHAR
  - notes about A-5
- Variable
  - for adding current date 2-91
  - for adding current time 2-91
  - for adding timestamp 2-91

Variables
- CURDATE 2-162
- CURTIME 2-162
- Global. See Global variables

Vendor Strings
- embedded 2-94
- in LEFT OUTER JOIN statements 2-95

VIEW.DDF D-1

Views
- creating
  - about 1-8
- database 2-62
- deleting 1-8
- grouped 2-62
- system table (X$View) D-13

W
- W32BTXLT 2-172
- WHILE
  - grammar supported 2-146
- WITH REPLACE keyword 2-24, 2-54

Words
- reserved B-1
- WSTRING data type (Unicode) A-21
- WZSTRING data type (Unicode) A-21

X
- XSAttrib D-1
- XSAttrib system table D-11
  - attributes ignored by Pervasive.SQL 2000i D-12
- X$Depend D-2
- X$Depend system table D-23
- X$Field D-1
- X$Field system table D-5
- X$File D-1
- X$File system table D-4
- X$Index D-1
- X$Index system table D-8
- X$Proc D-1
- X$Proc system table D-14
- X$Relate D-1
- X$Relate system table D-19
- X$Rights D-1

Index 9
X$Rights system table  D-17
X$Trigger  D-2
X$Trigger system table  D-21
X$User  D-1
X$User system table  D-15
X$View  D-1
X$View system table  D-13

Z

ZSTRING data type  A-22