Developer's Guide

Version 4.2

Proprietary and Trademark Information

Version 4.2

Table of Contents

Preface

Thank you for your interest in Version 4.2 of the PointBase product line.

Purpose

This guide describes how to develop applications using PointBase RDBMS. The following is a list of some things you can expect from this guide.

- PointBase JDBC Tutorials
- Supported SQL Standards and Syntax
- PointBase Database Concepts and Techniques

Audience

This guide is geared towards the Java development community. Because PointBase is the 100% Pure Java Application Database, this guide assumes that you know the following concepts:

- Have basic knowledge of the Standard Query Language (SQL).
- Have basic knowledge of the Java programming language.
- Have basic knowledge of Java Database Connectivity (JDBC).
- Understand basic database concepts.
- Have knowledge of your operating system and server and client concepts.

Release Notes

The following link displays the most up-to-date information on PointBase products.

www.pointbase.com/support/releasenotes.html

Document Feedback

Please send comments or suggestions for all PointBase documentation to the following email address.

docfeedback@pointbase.com

Document Conventions Used in This Guide

Convention	*Identifies*	*Examples*
ALL UPPERCASE LETTERS	• Environment variables • Database table names • SQL Keywords	• PATH • S_LST_OF_VAL • CREATE TABLE
`Courier New font`	• Directory, file, folder, and path names • Code • Data you need to type	• `c:\pointbase\img.bmp` • `Set PointBase =` • `Type Your Company Name Here`
Initial Uppercase Letters	PointBase names, objects, properties, windows, screens, dialog boxes, menus, buttons, tabs, applets, fields, and icons	PointBase Embedded, Business Component object, List Editor window, Main menu, and Cancel button
Italics	• Book titles • Cross references in an index or glossary • Variables • Arguments to statements of functions • First appearance of a new word or phrase • Emphasis	• *User's Guide* • *see also* or *see* • *APPSRVR_4X_ROOT* • *variable, rate, prompt$* • *new word* or *phrase* • Do not do this *before* you do that.
[ ]	Optional italicized arguments or characters inside angle brackets	[*caption$*]
{ | }	Choice from listed arguments; use OR operator (|) to separate	{Goto *label* | Resume Next | Goto 0}

Developer’s Overview

This chapter outlines the PointBase Relational Database Management System (RDBMS). It describes the JDBC driver, the JDBC API, and the SQL standards supported by PointBase. This chapter also describes new features and changes with PointBase RDBMS Version 4.2.

JDBC and PointBase

The core JDBC Application Program Interface (API) consists of a set of call level interfaces found in the java.sql package. The JDBC API is used by Java applications to access and manipulate the data stored in a database by invoking SQL commands. For more details on the JDBC API refer to the Sun Microsystems Inc.’s website: *http://java.sun.com/* or the *Sun Microsystems JDBC* manual.

PointBase fully supports JDBC 1.x, a subset of JDBC 2.0 API, and a subset of JDBC 2.0 Extension Interfaces, which Table 1 describes. PointBase also supports additional JDBC 2.0 Extension Interfaces for “distributed transactions.” (See "JDBC 2.0 Optional Package API" on page 81.) Additionally, PointBase supports a subset of JDBC 3.0 API, which Table 2 describes.You can also view any unsupported methods at, "Appendix B: Unsupported JDBC 2.0 Methods in PointBase" on page 179.

Table 1: JDBC 2.0 API Supported by PointBase

API	*Description*
java.sql.BatchUpdateException	Provides information about errors that occurred during batch operations
java.sql.Blob	Provides access to and manipulation of Binary Large Object data
java.sql.CallableStatement	Provides access to and manipulation of Stored Procedures
java.sql.Clob	Provides access to and manipulation of Character Large Object data
java.sql.Connection	Constructs and manages the connection to the database
java.sql.DatabaseMetaData	Provides metadata information about the database
java.sql.Driver	Provides information about and manages the JDBC driver
java.sql.PreparedStatement	Manages dynamic SQL statements
java.sql.ResultSet	Provides metadata information about the result set
java.sql.ResultSetMetaData	Manages result set metadata information
java.sql.Statement	Manages static SQL statements
javax.sql.DataSource	Provides access to JDBC drivers and manages data sources. [See "Additional PointBase Methods" on page 84.]

Please note that JVM 1.3 does not support the following JDBC 3.0 methods. Use JVM 1.4.

Table 2: JDBC 3.0 API Supported by PointBase

API	*Supported Methods*
java.sql.Connection	PreparedStatement prepareStatemnt (String SQLStmt, Statement.RETURN_GENERATED_KEYS)
java.sql.DatabaseMetaData	boolean supportsGetGeneratedKeys( )
java.sql.ResultSetMetaData	boolean isAutoIncrement(int column)
java.sql.Statement	int executeUpdate(String SQLStmt, Statement.RETURN_GENERATED_KEYS) int execute(String SQLStmt, Statement.RETURN_GENERATED_KEYS) ResultSet getGeneratedKeys()

The PointBase JDBC Driver

The PointBase JDBC driver provides access to the PointBase RDBMS. The driver interprets the database Universal Resource Locator (URL) to connect to the appropriate database. PointBase implements a “Type 4” JDBC driver, directly accessing the PointBase RDBMS using JDBC calls.

To use the PointBase JDBC driver in your application, you must first load and register the driver with the *JDBC DriverManager*, and then provide the URL of the database to which you want to connect. The database URL specifies the connection protocol, database location, "listener" port, and the database name. Please refer to the basic tutorial chapter in this guide for a more detailed explanation.

SQL and PointBase

PointBase RDBMS adheres to SQL-92 Entry and Transition levels, as defined by ANSI and ISO standards. PointBase also implements some features defined in the SQL-99 (SQL3) standard.

For more specific information about using SQL with PointBase, please refer to "Appendix A: SQL Reference," of this guide and the "SQL Data Types" Chapter, which defines the data type mappings from SQL to JDBC and Java.

Your Application and PointBase

This section shows how the PointBase RDBMS interacts with Java applications to provide database functionality.

Figure 1.2 shows PointBase Embedded, which is designed to be deployed as an integral part of your application. Both the Java Application and PointBase Embedded run within the same JVM. Applications can make multiple database connections to the PointBase database.

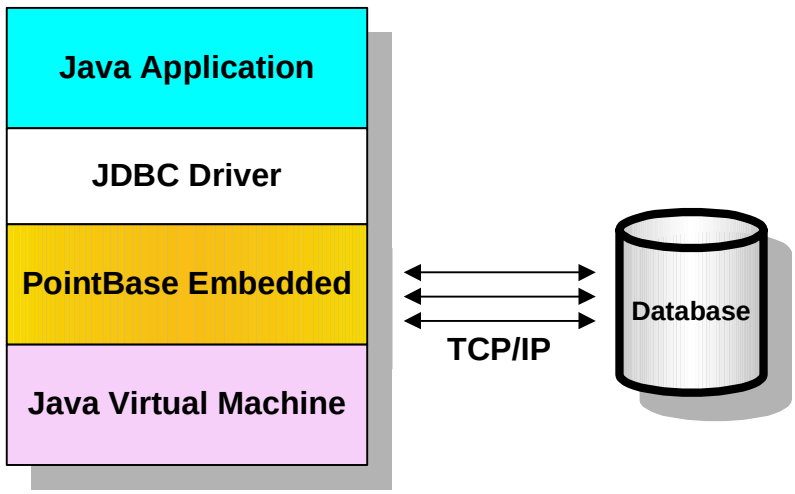


Figure 1.3 Using PointBase Embedded

Figure 1.3 shows PointBase Server, which is deployed using the traditional client-server model. A thin client is deployed as an integral part of the client application that both reside in a single JVM. This connects over the network to PointBase Server that runs in a second JVM.

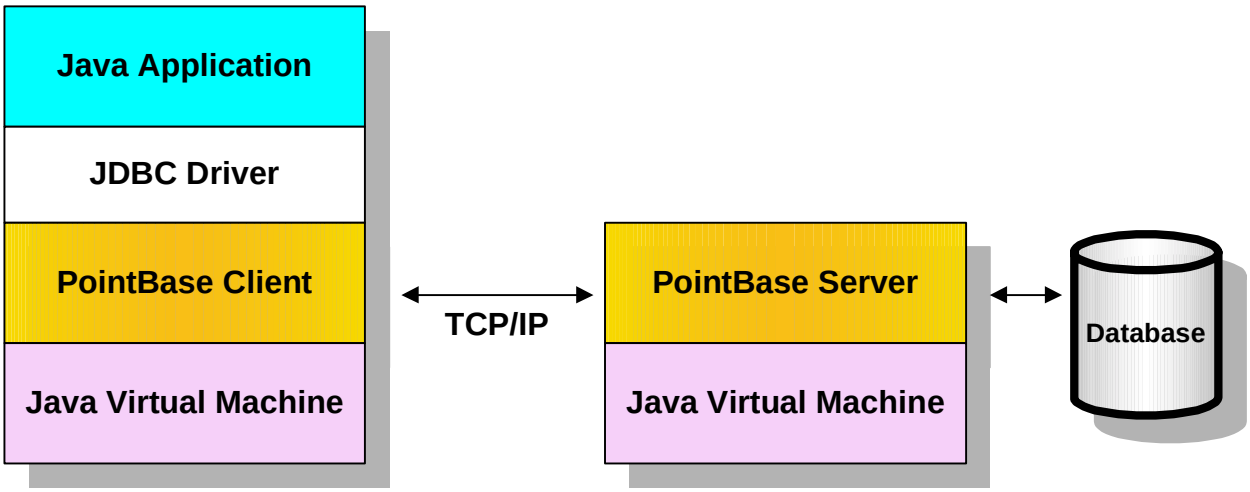


Figure 1.4 Using PointBase Server

What's New With PointBase RDBMS?

This section describes all of the new features and changes to PointBase RDBMS.

PointBase Embedded/Server RDBMS Enhancements

Autoincrement (Identity Columns)

Many applications require a way to generate a unique number for each row inserted into a table. The Autoincrement functionality provides a means to generate unique values, which can be used as Primary Keys. This functionality is also referred to as "Identity Columns." Two callback methods specified by JDBC 3.0 are provided to retrieve the numbers.

Updateable Cursors

Release 4.2 supports Updateable cursors and all the related metadata calls as specified in the JDBC 2.0 specification. Updateable result sets are cursors that can be updated programmatically. Updateable cursor provides the ability to update rows in a result set using methods in the java programming language rather than having to update with an SQL statement. By creating an updateable result set, you will have the ability to insert a new row, update or delete a row to the result set and persist the information to the database. This feature will be helpful to write GUI applications that allow users modify any displayed row.

Role Based Security

Security in a SQL database system is extremely important. Unfortunately, the only security that the SQL2 specification defines is grant/revoke of privileges on database objects to users. SQL3 extends this to allow grant/revoke of privileges on database objects to "Roles", which are declared groups of users. The functionality affects the SQL based authorization aspect only and not the authentication of a user. Roles can be granted to users or to other roles. By granting a privilege to a role, that privilege is implicitly granted to all users or roles that have been granted that role. By using roles, a system administrator can go through the laborious process of granting all the necessary privileges to a typical user for a given group only once, by granting those privileges to a role. Then, the administrator can simply grant the role to each typical user. Without roles, the administrator would have to grant the privileges to each user individually. If all users in the entire system are entitled to a privilege, than this can also be easily done by granting the privilege to PUBLIC, a pseudonym for every user in the system. But roles are extremely helpful if some privileges need to go to a large number of users, but not everyone.

On-Line Database Backup

Online backup functionality facilitates taking backup of the database while the database applications are running. This facility can be used by the applications, which do not want to bring down the database while taking a backup. The functionality will also be useful for applications that want to take backup when some critical event is recorded in the database and want to take backup of the database immediately.

Ability to Update User Password

The users will be able to update their passwords. This was a limitation in the past and has now been removed.

Union Distinct

Release 4.2 adds support for Union Distinct. UNION DISTINCT is a classic set union. Only unique rows are returned; any duplicates are discarded. The overall Union functionality has also been made more robust. UNION is now allowed in all supported DML statements.

API to Load and Unload Database

The load/unload database functionality is now available as an API. An application can now load/unload the database, in a bulk fashion, into a JDBC file (.java) or a SQL file or both.

Enhanced Database Creation

A new flag named “Create” has been added to the URL. This flag can be used to alter the behavior when an attempt is made to create a new database. The user will be able to either create a new database if it does not already exist, or let the RDBMS generate an exception if the database already exists or overwrite an existing database if so desired.

Support for BigInt Data Type

The “BigInt” data type is now supported.

Shutdown Command

A new SQL command “Shutdown” has been added to shutdown a database. This command is needed especially for the distributed transaction (XA) environment where closing all connections to the database does not imply closing the database.

In the case of PointBase Server in a non-XA environment, this command does a remote shutdown of the server.

Quality Improvements

1. PointBase Database Console and PointBase Commander now have the same default autocommit behavior.
2. When a user tries to drop a self-reference table, then the RDBMS automatically drops the constraint of self-reference key.
3. It is no longer possible to create multiple schemas with the same name. A bug has been fixed where under certain situation one could create multiple schemas with the same name.
4. “ALTER Table DROP Column” had a bug, which has now been fixed.
5. Suspended transactions in a JTA environment can now be rolled back from the same session or a different session.
6. In a distributed transaction (XA) environment, users can now connect to a database with a username and password that are different without getting an authentication exception. In the last release the username and password were required to be identical in the getXAConnection() statement.
7. Sub Queries have been made more robust. Null values are now returned for empty set sub queries.
8. DatabaseMetaData.getTables() has been made more robust.

9. Previously a RDBMS failure caused the application to exit in certain cases. This bug has now been fixed.
10. In certain rare situations, two applications could connect to the same database using PointBase Embedded and would result in corrupting the database. This bug has now been fixed.
11. The number of temporary files being used when sorting result sets containing blob/clob columns has now been reduced.

Important Changes in Release 4.2

Please make note of the following changes:

1. To conform to the ANSI/ISO SQL standard, the default user PUBLIC with default password PUBLIC has been changed to username PBPUBLIC with default password PBPUBLIC. The default role for this user is NONE.
2. The default schema name has been changed from PUBLIC to PBPUBLIC.
3. The RDBMS will accept PUBLIC as a synonym user name, password, and schema name for PBPUBLIC. This will occur everywhere except in GRANT and REVOKE statements, where PUBLIC will always mean the pseudo-user PUBLIC.
4. Upon upgrading PointBase from a version earlier than 4.2, PointBase will check if there are users called PBPUBLIC, PBSYSADMIN, PBDBA, or READALL. If any of these users already exist before upgrading to 4.2, an error is raised. You must drop the offending user(s) before re-attempting the upgrade.
5. The JDBC URL had been simplified in Release 4.0 to uniquely identify the different PointBase databases. URLs using the older format continue to be supported. The new format for the URLs is:

   •PointBase Server:

```
jdbc:pointbase:server://<machine_name>:<port_number>/<db_name>
```

   •PointBase Embedded:

```
jdbc:pointbase:embedded:<db_name>
```

   •Pointbase Micro:

```
jdbc:pointbase:micro:<db_name>
```

Best Practices

1. PointBase requires that you use the same version of the client and server jar files. When connecting to the PointBase Server, the PointBase Client jar file version must match the PointBase Server jar file version. For example: use pbserver42GA.jar and pbclient42GA.jar.

PointBase JDBC Basic Tutorial

This tutorial is intended as a quick reference to the JDBC API. PointBase recommends that you consult a JDBC reference manual or http://java.sun.com for more comprehensive and the most up to date information.

The basic tutorial describes fundamental JDBC operations to access and manipulate data using the JDBC API with the PointBase RDBMS. The code snippets in this tutorial are taken from the sample application included in the "<install_folder>\pointbase\samples\server_embedded\" directory of your PointBase installation. The examples in this tutorial include: connecting to the database, creating executable statements and closing the connection to the PointBase database.

Each example provides: a brief description of the code snippet illustrated, a code snippet from the sample application code, and any additional information to explain the code snippet in more detail.

Refreshing the Sample Database

If you have deleted or overwritten the sample database provided with your PointBase installation, you must refresh the sample database by using the following steps:

Step 1. Launch the "embedded_commander.exe" file in the "<install directory>\pointbase\tools\embedded" directory.

Step 2. Follow the prompts to create a new database called "sample."

Step 3. Type `run sample.sql`. You must type the complete path to the "sample.sql" file, for example,
`run c:/pointbase/samples/server_embedded/sample.sql;`

Making a Connection to PointBase

The following section describes the process of connecting to a PointBase database, using the JDBC API.

Loading the PointBase JDBC Driver

This code snippet instantiates the PointBase JDBC driver:

```
// The PointBase Universal JDBC Driver
String l_driver = "com.pointbase.jdbc.jdbcUniversalDriver";

// Load the PointBase JDBC Driver
Class.forName(l_driver).newInstance();
```

Connecting to the PointBase database

This code snippet establishes a connection with the PointBase database by passing the database URL, a username and password. By connecting with the database you create a connection object (m_conn in the sample application). The User name and Password both default to PBPUBLIC if they are not specified explicitly.

```
// The URL for the sample PointBase database
String l_URL = "jdbc:pointbase://" + p_product + "/sample";

// Database UserID
String l_UID = "pbpublic";

// Database Password
String l_PWD = "pbpublic";

// Establish connection with the database and return a Connection object
m_conn = DriverManager.getConnection(l_URL, l_UID, l_PWD);
```

The form of the PointBase URL, depends on which PointBase database you are using. The following gives examples for the PointBase Embedded and Server databases:

- PointBase Embedded

```
"jdbc:pointbase:embedded:sample"
```

- PointBase Server

```
"jdbc:pointbase:server://<server ip address>/sample"
```

or

```
"jdbc:pointbase:server://<server name>/sample"
```

To create a **new database**, you must use one of the specified PointBase flags. The following example uses the *new* flag.

```
"jdbc:pointbase:server://<server name>/sample,new"
```

Make sure you refer to the *PointBase System Guide* before using any flag in the URL. Each flag adheres to different rules when applied. (See the chapter, “Advanced Tips for Starting PointBase,” of the *PointBase System Guide*, and then browse the section, “Variable Descriptions.”)

Using DataSource

Instead of using the DriverManger facility to connect to the PointBase database, you may use a JDBC DataSource by initializing a DataSource object. The following example describes how to connect to a PointBase database using a DataSource object.

```
// The URL for the sample PointBase database
String l_URL = "jdbc:pointbase://" + p_product + "/sample";

// Database UserID
String l_UID = "pbpublic";

// Database Password
String l_PWD = "pbpublic";

// Create DataSource object
jdbcDataSource ds = new jdbcDataSource();
        ds.setDatabaseName(l_URL);
        ds.setUser(l_UID);
        ds.setPassword(l_PWD);
        ds.setCreateDatabase(true);

// Establish connection with the database and return a Connection object
m_conn = ds.getConnection();
```

Creating and Executing Static JDBC Statement

The following code snippet gives an example of how to create and execute static JDBC statements. First, it defines the SQL statement that the statement will execute, a statement is then created and executed to return a read-only, non-scrollable Result Set object. *Updateable* and *scrollable* result sets are discussed further in the advanced JDBC tutorial.

```
// Create the SQL Query
  String SQL_SELECT = "SELECT customer_tbl.name, customer_tbl.city,"
                        + " manufacture_tbl.name, manufacture_tbl.city"
                        + " FROM customer_tbl, manufacture_tbl WHERE"

+ " UPPER(customer_tbl.city) = UPPER(manufacture_tbl.city)";

// Create a static JDBC statement
            m_stmt = m_conn.createStatement();

// Execute the SQL statement and return a Non-Scrollable Result Set
            m_rs = m_stmt.executeQuery(SQL_SELECT);
```

Retrieving Row Values From Non-Scrollable Result Sets

A non-scrollable result set only allows you to retrieve the values stored in the result set in sequential order. The following example describes how to retrieve values from a non-scrollable result set.

When a result set is returned, the cursor is positioned before the first row of the result set. To access the first value of the result set you must advance the cursor to the first row using the `resultSet.next()` method. This method is used to move the cursor from row to row in the result set, and returns a Boolean TRUE value if there is data in the row to which the cursor is pointing.

```
// Scroll through the result set (top to bottom)

while(p_rs.next())
{
       // Loop through the columns
       for (int i = 1; i <= rsColumns; i++)
       {
              // Get the data from the result set
              // Place methods to retrieve data here
       }
```

The following code snippets illustrate how to retrieve specific data types from the result set. These methods would be placed inside the “for” loop of the snippet above.

```
// Retrieve JDBC Char and Varchar data types
String rsString = p_rs.getString(i);

// Retrieve JDBC Integer data types
Integer rsInt = new Integer(p_rs.getInt(i));

// Retrieve JDBC Smallint data types
Short rsShort = new Short(p_rs.getShort(i));

// Retrieve JDBC Boolean data types
Boolean rsBool = new Boolean(p_rs.getBoolean(i));

// Retrieve Float, Double, Numeric and Decimal JDBC data types
Double rsDouble = new Double(p_rs.getDouble(i));
```

NOTE: PointBase recommends that you use the `ResultSet.getBigDecimal()` method to retrieve Numeric and Decimal JDBC data types. This method is omitted in this example for JDK 1.1.8 and JView compatibility.

```
// Retrieve JDBC Real data types
Float rsFloat = new Float(p_rs.getFloat(i));

// Retrieve JDBC Date data types
java.sql.Date rsDate = p_rs.getDate(i);

// Retrieve JDBC Time data types
java.sql.Time rsTime = p_rs.getTime(i);

// Retrieve JDBC Time Stamp data types
java.sql.Timestamp rsTimestamp = p_rs.getTimestamp(i);
```

Closing and Committing Objects

The following examples describe how to close result sets, static JDBC statements and finally database connections. However, before closing a connection to the database or when you have completed a transaction, you must either commit or rollback any changes made.

Rolling Back or Committing the Transaction

The following code snippet describes how the sample application rolls back all changes made to the database up to this point. It uses the `rollback()` method.

```
// Rollback any changes made to the database
// Use m_conn.commit() if you don't wish to rollback the transaction
m_conn.rollback();
```

NOTE: If you fail to commit a transaction prior to disconnecting from the database, and you do not have “auto commit” switched on, the transaction will be rolled back by default and any changes made will be lost.

Closing the Result Set

When you close a result set, you invalidate the result set. That is, it cannot be used for any subsequent operations. The following code snippet describes how the sample application closes the result set object.

```
// Close the Result Set
m_rs.close();
```

Closing the JDBC Statement

The following code snippet describes how the sample application closes the JDBC statement object.

```
// Close the JDBC statement
m_stmt.close();
```

Closing the Connection to the Database

The following code snippet describes how the sample application closes the connection object. This closes the connection to the database.

```
// Close the connection
m_conn.close();
```

PointBase JDBC Advanced Tutorial

This tutorial is intended as a quick reference to the JDBC API. PointBase recommends that you consult a JDBC reference manual or http://java.sun.com for more comprehensive and the most up to date information.

The advanced tutorial describes how to perform more complex operations using the JDBC API with the PointBase RDBMS. The code snippets in this tutorial are taken from the sample application included in the "<install_folder>\pointbase\samples\server_embedded\src" directory of your PointBase installation. The examples in this tutorial include returning scrollable result sets and performing batch updates.

Each example provides: a brief description of the code snippet illustrated, a code snippet from the sample application code, any additional information to explain the code snippet in more detail. The examples assume you have already connected to the PointBase sample database. (Refer to the Basic Tutorial for information about connecting to a PointBase database.)

Creating and Executing a Dynamic JDBC Statement

The following example describes how to create and execute a dynamic JDBC statement. A dynamic JDBC statement can improve performance of applications relative to static JDBC statements. Unlike a static JDBC statement, dynamic or prepared statements are only compiled once, regardless of the number of times that they are used. For example, use a dynamic JDBC statement is when you need multiple executions of a particular SQL statement that has changing values associated with it.

Creating a Prepared Statement

The following code snippet shows an example of an SQL string for use within a prepared statement. The `preparedStatement()` method uses this string as its argument. The prepared statement executes the INSERT statement as many times as required. The question marks indicate dynamic parameters that will be bound to the prepared statement. The prepared statement object is created using the `Connection.prepareStatement()` method.

```
// Initialize SQL for the prepared statement
String SQL_PREP_INSERT = "INSERT INTO order_tbl (order_num, customer_num,"
+ " rep_num, product_num, sales_tax_st_cd, quantity,"
+ " shipping_cost, sales_date, shipping_date,"
+ " delivery_datetime, freight_company) VALUES"
+ " (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";

// Create a prepared statement
m_prepStmt = m_conn.prepareStatement(SQL_PREP_INSERT);
```

Binding the Dynamic Variables to the Prepared Statement

The following code snippet provides an example of binding dynamic variables to the prepared statement and executing the prepared statement. Bind the variables by using the `preparedStatement.set<DataType>` method, for example preparedStatement.setInt(). The first input argument for this method is the bind parameter index (i.e. which question mark it represents), the second input argument is the desired value to be bound. The prepared statement is executed using the `preparedStatement.Execute()` method.

```
// Bind the parameters to the prepared statement
m_prepStmt.setInt(1, var1[i]);
m_prepStmt.setInt(2, var2[i]);
m_prepStmt.setInt(3, var3[i]);
m_prepStmt.setInt(4, var4[i]);
m_prepStmt.setString(5, var5[i]);
m_prepStmt.setInt(6, var6[i]);
m_prepStmt.setDouble(7, var7[i]);
m_prepStmt.setDate(8, var8[i]);
m_prepStmt.setDate(9, var9[i]);
m_prepStmt.setTimestamp(10, var10[i]);
m_prepStmt.setString(11, var11[i]);

// Execute the SQL prepared statement and return a result set
m_prepStmt.execute();
```

Using Scrollable Result Sets

The following examples describe how to create a statement object for returning and manipulating a scrollable result set. By returning this type of result set, you have the capability to retrieve result set row values in any order. Conversely, using a non-scrollable result set, you can only retrieve result set row values as you scroll forward. With scrollable result sets, however, you can scroll either forward or backward. Additionally, you can also scroll by specifying a position in the result set. To begin returning scrollable result sets, you must first specify the result set *type* when you create the SQL statement.

Result Set Types and Concurrency

To create a scrollable result set you must specify its result set type. It defines whether or not the result set is scrollable. In PointBase, you can specify TYPE_SCROLL_INSENSITIVE or TYPE_FORWARD_ONLY. If you do not specify a result set type, the default is TYPE_FORWARD_ONLY (a non-scrollable result set). So, to make the result set scrollable, you must specify TYPE_SCROLL_INSENSITIVE.

In addition to the result set type, you must also specify the result set *concurrency*. It defines whether or not the result set is *read-only* or *updateable*. In PointBase, you can specify CONCUR_READ_ONLY or CONCUR_UPDATEABLE. Using CONCUR_UPDATEABLE, you have the ability to update rows in a result set using methods in the Java programming language rather than having to update them with an SQL statement. For example, you can insert, update, or delete a result set row, and make your changes permanent to the database. Using CONCUR_READ_ONLY, you may read the rows in the result set only; you cannot change them in any way.

Creating a Read-Only Scrollable Result Set Statement Object

The following code snippet illustrates how to create a statement object that can return a read-only scrollable result set. Note that you can also use a prepared statement to return one.

```
// Create a statement and set the Result Set parameters to make it scrollable
m_stmt = m_conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

Moving the Cursor

After returning a scrollable result set using a statement object, you can move the result set cursor. The following examples describe how the sample application moves the cursor in a scrollable result set. Similar to non-scrollable result sets, you access sequential rows of the result set by using the `resultSet.next()` method. You can also move the cursor anywhere in a scrollable result set using the following methods.

First()

The following code snippet describes the `first()` method. It moves the cursor to the first row in the result set.

```
// Move the cursor to the first entry in the result set - this is the data we just
// inserted
m_rs.first();
```

Last()

The following code snippet demonstrates the `last()` method. It moves the cursor to the last value in the result set m_rs

```
// Move the cursor to the last entry in the result set
m_rs.last();
```

Previous()

The following code snippet demonstrates the `previous()` method. It moves the cursor to the previous position in the result set m_rs.

```
// Moving back to the previous entry in the result set
m_rs.previous();
```

Absolute()

The following code snippet demonstrates the `absolute()` method. It moves the cursor to a specific position in the result set. For example, this code snippet describes how to move the cursor to the first row in the result set.

```
// Moving to the first entry in the result set using its absolute row reference
m_rs.absolute(1);
```

BeforeFirst()

The following code snippet demonstrates the `beforeFirst()` method. It moves the cursor before the first value in the result set.

```
// Moving before the first row
m_rs.beforeFirst();
```

AfterLast()

The following code snippet demonstrates the `afterLast()` method. It moves the cursor after the last value in the result set.

```
// Move after the last row
m_rs.afterLast();
```

Relative()

The `relative()` method moves the cursor to the specified position relative to the current position of the cursor. This code snippet demonstrates how to move the cursor two rows forward from the current position of the cursor.

```
// Move cursor relative to current position
m_rs.relative(2);
```

Next()

The following code snippet demonstrates the `next()` method. It moves the cursor to the next row in the result set m_rs.

```
// Move the cursor to the next entry in the result set
m_rs.next();
```

Setting the Direction of the Cursor in Scrollable Result Sets

When you change the direction of the cursor, it effectively reverses all of the previous methods. To set the direction of the cursor you must use the `set.FetchDirection()` method. The fetch direction is set to FETCH_FORWARD by default, and the cursor moves in the forward direction. PointBase supports the two following fetch directions:

FETCH_REVERSE

The following code snippet demonstrates how to reverse the direction of the cursor in the scrollable result set.

```
// Set the cursor to scroll backwards through the Result Set
m_rs.setFetchDirection(ResultSet.FETCH_REVERSE);
```

As an example of cursor behavior with the fetch direction set to FETCH_REVERSE, if you call the `beforeFirst()` method, the cursor is moved after the last row of the result set.

FETCH_FORWARD

The following code snippet demonstrates how to set the fetch direction of the scrollable result set to FETCH_FORWARD.

```
// Set the cursor to scroll forwards through the result set
m_rs.setFetchDirection(ResultSet.FETCH_FORWARD);
```

Retrieving Information About a Result Set

The following examples describe how to retrieve information about a result set. This example refers to only a few of the methods available for retrieving information about the result set. Refer to JDBC API documentation at *http://java.sun.com* or your JDBC reference for a comprehensive list of the available methods, and "Appendix B: Unsupported JDBC 2.0 Methods in PointBase," for the list of methods that PointBase does not support.

ResultSet.getType()

The `resultSet.getType()` method can return TYPE_SCROLL_INSENSITIVE or TYPE_FORWARD_ONLY. The following code snippet describes how to get the type of the result set m_rs.

```
// Check if result set is scroll insensitive
m_rs.getType()
```

ResultSet.getConcurrency()

The `getConcurrency()` method can return CONCUR_READ_ONLY or CONCUR_UPDATEABLE. The following code snippet describes how to get the concurrency of the result set m_rs.

```
// Check the concurrency of the result set
m_rs.getConcurrency()
```

ResultSet.getMetaData()

The `getMetaData()` method obtains information about the result set, for example, the column names and column data types. The following code snippet describes how to get the meta data of the result set m_rs.

```
// Retrieve Result Set Meta Data to obtain result set properties
m_rsmd = m_rs.getMetaData();
```

Setting the Number of Returned Rows in Scrollable Result Sets

The following code snippets demonstrate how to set the fetch size or number of returned rows in a scrollable result set using two different methods. This is applicable to PointBase Server only. Also note that in most cases the default fetch size is optimal.

ResultSet.setFetchSize(int p_Rows)

The result set can change its default fetch size using this method. It will only affect the specified result set.

```
m_rs.setFetchSize(2);:
```

To set the default fetch size for all result sets created by a statement object, you can use the set fetch size using the Statment object. This method affects all result sets generated by this statement. For example:

```
Statement.setFetchSize( int p_Rows )
```

Creating an Updateable Scrollable Result Set Statement Object

The following code snippet illustrates how to create a statement object that can return an updateable scrollable result set. Note that you can also use a prepared statement to return one.

```
// Create a statement and set the Result Set parameters to make it scrollable
m_stmt = m_conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

NOTE: You may also return non-scrollable, updateable result sets, if you specify TYPE_FORWARD_ONLY and CONCUR_UPDATEABLE.

Locks

While updating, inserting, or deleting a row in an updateable scrollable result set, PointBase will change the lock on the row to an exclusive lock. If PointBase cannot acquire the lock, it will throw an exception.

Autocommit

While using updateable scrollable result sets, PointBase encourages you to set autocommit to *false*. If you set it to *true*, PointBase commits the result sets, which invalidates them.

Verification

Before inserting any new rows or updating any row values, PointBase will perform any necessary checking, including constraints and reference integrities. If a new row or row value fails to satisfy any of them, PointBase will throw an exception. Also, while inserting a new row, make sure to define all column values, because PointBase automatically sets undefined column values to the database default.

Restrictions

PointBase enforces the following restrictions for updateable scrollable result sets:

- A query that returns a result set can select from only a single table, and cannot contain any join operation.
- A query that returns a result set must select table columns only. It cannot select derived columns or aggregates.
- A query that returns a result set cannot have ORDER BY, GROUP BY, or HAVING clause.

Behavior

The PointBase JDBC driver will automatically determine the most feasible scrollable result set concurrency, if it observes the following behavior:

- The PointBase JDBC Driver will change the concurrency of a result set to read-only, if you attempt any of the previously mentioned restrictions.
- The PointBase JDBC Driver will change a result set type to TYPE_SCROLL_INSENSITIVE, if you specify the type TYPE_SCROLL_SENSITIVE.

Updating Row Values in Scrollable Result Sets

To update a row value in a result set, PointBase provides you with four different methods. Among their uses, you can set the row value of the result set that you want to update and most importantly, perform the actual update to the underlying database. PointBase also provides two additional methods that you can use to perform the following: cancel all updates to a row or verify a row value you just updated.

updateXXX( )

To update a row value in a result set, you must first set the value using the method, `updateXXX()`. It has two different forms:

- `update<datatype>( int columnName, value )`
- `update<datatype>( String columnName, value )`

This method supports all PointBase data types. The following example sets the *quantity* column value in the current row to *150* using the *Int* data type:

```
// m_rs.updateInt() method updates the field in question with supplied integer value
m_rs.updateInt("quantity", 150);
```

updateRow( )

To update the row value of the actual underlying database on the next transaction commit, you use the method, `updateRow()`. After updating a row value, you will be able to view your updated row value in the current result set. The following is an example of how to use this method:

```
// m_rs.updateRow() method updates the row in the database.
m_rs.updateRow();
```

rowUpdated( )

To verify that you updated the row value in the underlying database, you may use the method, `rowUpdated()`. The following is an example of how to use this method:

```
m_rs.rowUpdated()
```

cancelRowUpdates( )

To cancel the updated row value in the result set, you may use the method, `cancelRowUpdates()`. You cannot cancel the update if you have already made the change to the underlying database; that is, you cannot cancel the update after calling the `updateRow()` method. The following is an example of how to use this method:

```
// m_rs.cancelRowUpdates() cancels in case a wrong update has been made.
m_rs.cancelRowUpdates();
```

Inserting Rows Into Scrollable Result Sets

To insert a new row into a result set, PointBase provides you with four methods. Using them, you perform the following things: place the cursor to the insertion row in case it is not currently on the row, to which you want to insert; set the new values of the row, similar to updating a row value; and, insert a new row making it permanent to the underlying database. After inserting a new row, you must use another method to move the cursor from the insertion row to the current row, a non-insertion row.

moveToInsertRow( )

To move the result set cursor to the row into which you want to insert, you must use the method, `moveToInsertRow()`. The following is an example of how to use this method:

```
m_rs.moveToInsertRow();
```

updateXXX( )

You must use the method, `updateXXX()` to set the row values for the new row, as you similarly used this method to update a row value. See previous section on updating row values.

insertRow()

To permanently insert the new row into the underlying database on the next transaction commit, you use the method, `insertRow().` The following is an example of how to use this method:

```
m_rs.insertRow();
```

moveToCurrentRow( )

To move the cursor to a non-insertion row, if you do not want to insert another row, you must use the method, `moveToCurrentRow()`. The following is an example of how to use this method.

```
m_rs.moveToCurrentRow();
```

Deleting Rows From Scrollable Result Sets

To delete rows from result sets, PointBase provides you with two methods. For example, one method deletes the row permanently from the underlying database on the next transaction commit. The second method verifies if the row has been deleted from the database. Please note that if you try to retrieve a deleted row value from the current result set, PointBase will return only NULL values.

deleteRow( )

To permanently delete a row from the underlying database, use the method, `deleteRow()`. The following is an example of how to use this method:

```
// Deleting currentrow.
m_rs.deleteRow();
```

rowDeleted( )

To verify whether or not a row still exists in the current result set, use the method, `rowDeleted()`. The following is an example of how to use this method:

```
mrs.rowDeleted()
```

Flushing the Database Log

The following examples describe how to switch to a fresh database log file. The old log file is deleted as soon as it is no longer required by the DBMS. The database log file is flushed in different ways for embedded and server. The code snippets below illustrate log file switching fro both products:

```
// Switch log file for PointBase Embedded
((com.pointbase.jdbc.jdbcConnection)m_conn).switchLogFile();

// Switch log file for PointBase Server
((com.pointbase.net.netJDBCConnection)m_conn).switchLogFile();
```

Performing Batch Operations

The following examples demonstrates how to perform batch operations. Batch updates can improve performance for large numbers of SQL operations. You can use them for any SQL operation that returns an integer update count, but not a result set for example, INSERT, UPDATE, or DELETE. You can also use batch operations for any SQL DDL statement, for example, CREATE TABLE, DROP TABLE, or ALTER TABLE.

NOTE: Batch updates offer the most significant performance improvement when used with PointBase Server, due to reduced network access.

The following code snippet demonstrates the creation of a prepared statement, binding of variables, and adding the prepared statement to a batch using the `preparedStatement.addBatch()` method. The batch is executed, using the `preparedStatement.executeBatch()` method, once all the required prepared statements have been added.

```
// Create a SQL statement for the batch update
String SQL_BATCH_UPDATE = "UPDATE sales_tax_code_tbl SET effect_date = ?, rate = ? where
state_code = ?";

// Prepare a statement
m_prepStmt = m_conn.prepareStatement(SQL_BATCH_UPDATE);

for (int i=0; i<=9; i++)
{
    // Binding variables to the prepared statement
    m_prepStmt.setDate(1, java.sql.Date.valueOf(BATCH_DATA[1][i]));
    m_prepStmt.setFloat(2, (float)Float.valueOf(BATCH_DATA[2][i]).floatValue());
    m_prepStmt.setString(3, BATCH_DATA[0][i]);

    // Adding the prepared statement to the batch
    m_prepStmt.addBatch();
}

    // Execute the batch
int[] updateCounts = m_prepStmt.executeBatch();
```

NOTE: If Auto commit is set ON, the transaction will be committed when the `preparedStatement.executeBatch()` method is called.

Retrieving Data From BLOB Columns

The following code snippet shows how the sample application retrieves BLOB values from the result set using the `getBLOB()` method to retrieve the column value. The final two operations create a binary stream from the BLOB object to read it into a byte array. This byte array can then be used as required by your application.

```
// Retrieve the BLOB containing the sales rep image from the second column of
   // the result set and find out its length
Blob image = m_rs.getBlob(2);
int lob_length = (int)image.length();

// Create a Buffered input stream from the BLOB data and read it into a byte
// array
BufferedInputStream bufferedInStream = new BufferedInputStream( image.getBinaryStream()
);
byte[] byteBuffer = new byte[ lob_length ];
bufferedInStream.read( byteBuffer, 0, lob_length );
bufferedInStream.close();
```

Retrieving Data From CLOB Columns

The following code snippet shows how the sample application retrieves CLOB values from the result set using the `getCLOB()` method to retrieve the row value. The final two operations create a character stream from the CLOB object to read it into a character array. This character array can then be used as required by your application.

```
// Retrieve the CLOB containing the sales rep resume from the result set and determine
its length
Clob resume = m_rs.getClob(3);
lob_length = (int)resume.length();

// Create a buffered reader to read the character stream into a character array
BufferedReader bufferedReader = new BufferedReader( resume.getCharacterStream() );
char[] charBuffer = new char[ lob_length ];
bufferedReader.read( charBuffer, 0, lob_length );
bufferedReader.close();
```

Creating Functions

This section describes functions in PointBase. Using a function, you can transparently convert data to be stored in a PointBase database. Functions may only return a single value of the type specified in the CREATE FUNCTION SQL statement. To create a function (stored function), you must use the CREATE FUNCTION statement and specify an external Java method for the stored function to invoke. This section explains how to create and use stored functions in PointBase.

External Java Methods and Functions

In PointBase, functions may be implemented using external Java methods. These user-defined methods manipulate SQL data when the function is called by the database.

Creating an External Function

Suppose you want to INSERT a european formatted date into a table making sure that the date format is Y2K compatible. The following external Java method, dateConvert, is called from the stored function in the database. This external Java method converts a date from dd-mm-yyyy to yyyy-mm-dd, and then converts it to a java.sql.Date type.

```
public static java.sql.Date dateConvert(String p_value)
    {
        String l_day = new String(p_value.substring(0,2));
        String l_month = new String(p_value.substring(2,6));
        String l_year = new String(p_value.substring(6,10));

        return(java.sql.Date.valueOf(l_year + l_month + l_day));
    }
```

Specifying the External Function in a Stored Function

To invoke the dateConvert external Java method from a stored function, you must use the CREATE FUNCTION statement. The dateConvert external Java method is called from the class, SampleExternalMethods.

In order for the database to access this external Java method, the class SampleExternalMethods must be included in the database CLASSPATH. For PointBase Server, it must be in the Server CLASSPATH, but not in the Client CLASSPATH.

```
// SQL statement to Create a function
            String SQL_CREATE_FUNC = "CREATE FUNCTION dateConvert( IN P1 VARCHAR(20) )"
                                   + " RETURNS Date"
                                   + " LANGUAGE Java"
                                   + " NO SQL"
                                 + " EXTERNAL NAME \"SampleExternalMethods::dateConvert\""
                                   + " PARAMETER STYLE SQL";

// Create a statement and execute the SQL
            m_stmt = m_conn.createStatement();
            m_stmt.executeUpdate(SQL_CREATE_FUNC);

            // Close the statement
            m_stmt.close();
```

NOTE: The stored function converts the data before inserting it into the database, and after selecting data from the database.

Using the Function

The following code snippet describes how the dateConvert function is used in a SELECT statement by the Sample Database Application.

```
// SQL SELECT using the external function to convert the date in the WHERE clause
            String SQL_USE_FUNC = "SELECT city FROM office_tbl WHERE open_date ="
                                  + " dateConvert('01-02-1993')";

            // Create the statement
            m_stmt = m_conn.createStatement();

            // Execute the statment
            m_rs = m_stmt.executeQuery(SQL_USE_FUNC);
```

Creating Stored Procedures

You can create and use PointBase stored procedures in a similar way to functions. Stored Procedures may also use external Java methods to perform the procedure action. In addition, stored procedures may take any number of input parameters and return any number of output parameters, unlike functions which may only return one parameter. Stored procedures are invoked explicitly using JDBC callable statements or may be invoked using the CALL command in a trigger action. However, they cannot be invoked within SQL statements like a function.

Using INOUT and OUT Parameters

When using a stored procedure with java external methods, special care must be taken to properly handle parameters passed to the procedure. Parameters may be of type IN, OUT, or INOUT. Java passes arguments by value, not by reference; therefore, it is generally impossible to use stored procedures with argument values that need to be returned through the parameters. PointBase has added special JDBC Wrapper classes to remedy this issue. This section explains how you can use this wrapper with INOUT and OUT parameters.

Using JDBC Wrapper Classes

The jdbcInOut Wrappers are used by the database to enable the database to return values from Java methods using Callable Statements. They are only required for OUT or INOUT parameters. Each wrapper class has two constructors, a get and set method, and a toString method. The wrapper classes are contained in the package "com.pointbase.jdbc" included in your PointBase jar file.

The wrapper name corresponds to the JAVA data type represented by the wrapper. All mappings between SQL and JAVA data types are compliant with the JDBC specification. For the JDBC Binary and BLOB data types, a wrapper is not required,and a java byte array is passed as the input argument to your Java method.

- jdbcInOutDateWrapper—>Date Data Type
- jdbcInOutTimeWrapperTime—>Time Data Type
- jdbcInOutTimeStampWrapper—>TimeStamp Data Type
- jdbcInOutBooleanWrapper—>Boolean Data Type
- jdbcInOutLongWrapper—>BigInt Data Type
- jdbcInOutDoubleWrapper—>Double and Float Data Types
- jdbcInOutFloatWrapper—>Real Data Type
- jdbcInOutIntWrapper—>Integer Data Type
- jdbcInOutStringWrapper—>Char, Varchar, Clob Data Types
- jdbcInOutShortWrapper—>SmallInt Data Types
- jdbcInOutBigDecimalWrapper—>Decimal and Numeric Data Types

Creating an External Procedure Using JDBC Wrapper Classes

The code snippet below defines the getCost external procedure found in the class SampleExternalMethods. Initially, you must first use a constructor to obtain a connection to the database.

```
*/

import java.sql.*;
import com.pointbase.jdbc.jdbcInOutDoubleWrapper;

public class SampleExternalMethods
{
    // A connection object to allow database callback
    private Connection m_conn;

    // Constructor accepts a java.sql.Connection object to allow database callback
    public SampleExternalMethods(Connection p_conn)
    {
        m_conn = p_conn;
    }
```

The following Java method is called as a stored procedure by the database. Procedure uses the net order cost (INOUT) and state code (IN) to return the net order cost (INOUT). This particular procedure also makes a callback into the database

NOTE: A jdbcInOutDoubleWrapper is passed into this method as an argument rather than the FLOAT JDBC data type that was bound to the callable statement.

```
     public static void getCost(String p_productInfo, String p_state, jdbcInOutDoubleWra
pper p_price)
    {
        try
        {

            // Query the database for the sales tax rate
            Statement l_stmt = l_conn.createStatement();
            ResultSet l_rs = l_stmt.executeQuery( "SELECT rate FROM public.sales_tax_cod
e_tbl"
                                                  + " WHERE state_code ='" + p_state + "'"
);

            // Calculate the totoal cost of the item using the sales tax rate
            // obtained from the database.
            l_rs.next();
            float total_cost = (float)p_price.get() * (1 + (l_rs.getFloat(1)/100));


            // Bind the total cost to the INOUT variable to return
            p_price.set(total_cost);

            // Close the result set
            l_rs.close();

            // Close the statement
            l_stmt.close();


        }
```

Executing a Stored Procedure

To allow a stored procedure to call out from the database system to an external procedure, follow these two mandatory steps:

Create a stored procedure in the database.

The code snippet below shows how to create stored procedure, getCost in PointBase, where EXTERNAL NAME refers to the class and the getCost external procedure.

In the following example, getCost is a method contained within the class SampleExternalMethods.

```
// SQL statement to create a stored procedure
            String SQL_CREATE_PROC = "CREATE PROCEDURE getCost(IN P1 VARCHAR(2), INOUT P
2 FLOAT )"
                                   + " LANGUAGE JAVA"
                                   + " SPECIFIC getCost"
                                   + " DETERMINISTIC"
                                   + " NO SQL"
                                   + " EXTERNAL NAME \"SampleExternalMethods::getCost\""
                                   + " PARAMETER STYLE SQL";

            // Create a SQL statement
            m_stmt = m_conn.createStatement();
```

```
            // Execute the SQL
            m_stmt.executeUpdate(SQL_CREATE_PROC);

            // Close the statement
            m_stmt.close();
```

Create a JDBC CallableStatement that executes the stored procedure.

The code snippet below is an example of how to create a CallableStatement that invokes the stored procedure.

You must set the appropriate inbound arguments with values. After the execution of the CallableStatement, you may obtain the values for each applicable outbound argument.

```
// Create SQL to invoke a stored procedure
            String SQL_USE_PROC = "{ call getCost(?,?) }";

            // Create a callable statement with two binding parameters
            m_callStmt = m_conn.prepareCall(SQL_USE_PROC);

            m_callStmt.setString(1, "CA");
            m_callStmt.setFloat(2, 449.00F);

            m_callStmt.executeQuery();

            // Close the callable statement
            m_callStmt.close();
```

For further details on OUT and INOUT parameters, see ‘JDBC API Tutorial and Reference’, Second Edition, Sun Microsystems, by White, Fisher, Cattell, Hamilton and Harper.

Basic SQL Data Objects

This section describes basic data objects relative to the PointBase RDBMS. It describes each data object individually and explains how PointBase data objects interact with one another. Read this chapter before creating a database to fully understand the behavior of each data object within the PointBase RDBMS.

Data Objects Within the PointBase RDBMS

The following diagram illustrates the relationship between basic data objects in the PointBase RDBMS. The database itself is a data object that encompasses all other data objects. A database contains Schema objects, which in turn contain Table objects. Tables whose values are derived from other tables are called Derived Tables or Views. Finally, a Column is located within a Table. Columns are the smallest data object within PointBase RDBMS.

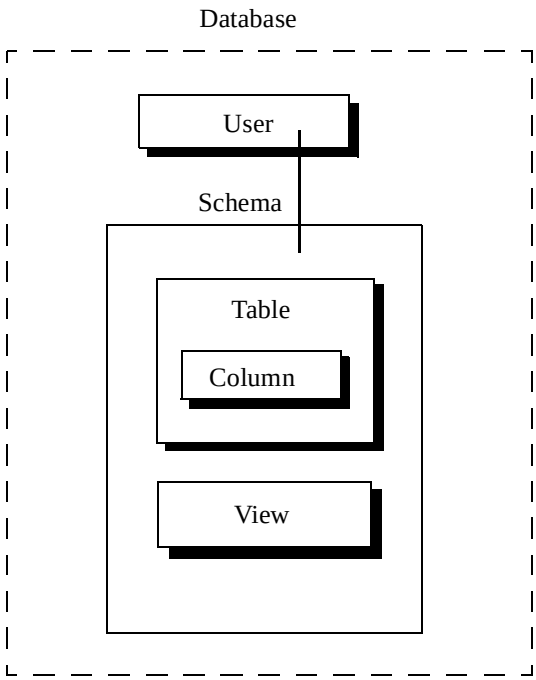


Figure 1.1 PointBase RDBMS Data Objects

Database

The PointBase RDBMS can contain one or more database(s). A database is at the highest level of abstraction and is simply an operating system file. PointBase stores all data in dbn files and all log information in wal files. For example, the sample database file is "`sample.dbn`" and the sample log file is "sample.wal." You can locate these files in the directory, "<install directory>\pointbase\databases."

PointBase automatically creates other `.dbn` or `.wal` files like `sample$$1.dbn` or `sample$$1.wal` when a `.dbn` or `.wal` file reaches its maximum size. All automatically created `.dbn` and `.wal` files have the same page size as the original `.dbn` or `.wal` file.

Database Size Limit

For the default page size of 4 K, the database size is limited to 0.5 terabytes. If the default page size is 1 K, the database size is limited to 128 GB, and for the default page size of 32 K, the database is limited to 4 terabytes.

Because PointBase supports multiple page sizes for a database, the previous limits are true assuming that the database does not use additional page sizes. If the database has more than one page size, the database size limit increases. For example, if the database has two different page sizes, one page size of 4K (0.5 terabytes), plus another page size of 32K (4 terabytes), the total database size limit is 4.5 terabytes.

Concurrent Databases

PointBase supports multiple databases, but only one database concurrently. If multiple connections are made to the PointBase RDBMS, then each connection needs to access the same database. When the set of connections to a particular database is completed, then the next set of connections can be initiated to another database.

Typically, multiple databases separate data for different applications. Schemas can be used to accomplish the same effect. Refer to "Schemas" in this chapter for more information

Read-Only Support

Using PointBase, you may query a database on a CD. In this section we use the term "read-only database," when the database files are on a CD or, when the database files are set to the operating system property "read-only." PointBase supports only SELECT statements for read-only databases. Using any other statements, such as INSERT, CREATE TABLE,... etc. with a read-only database causes PointBase to throw an exception. The error message states "Invalid statement."

To have a database on a CD, you must first create the database on a writable drive. After creating the database, connect to it using the PointBase Commander or any java program [see *PointBase System Guide*], and then close the connection without performing any other operations during the connection.

By performing this step, you ensure that all the data is completely recovered from the log (.wal) before loading the `.dbn` and `.wal` files on a CD. You cannot recover data from a database on a CD. If the database on a CD needs recovery, the application terminates with the following message on the screen (standard system output): “Database needs recovery from log. This version does not support recovery.”

To connect to the database on CD or any other location use the `pointbase.ini` file’s "`database.home`" parameter or the java command line -D option to specify the location of the database. See the *PointBase System Guide* for more information about starting PointBase.

Restrictions

Operations that involve writing to the database (dbn) or log (wal) files are not allowed. Additionally, PointBase does not allow the following statements, because they use temporary tables and writes into the database.

- Non-correlated subqueries that are part of IN predicate
- Read-only views
- Scrollable Cursors

User

Databases contain collections of users. Users are a means of providing security at the schema level. Each schema has explicit user(s) associated with it, one of which must own the schema. The schema owner has full access to the schema and determines the access privileges of the other users. To manage users, use the CREATE USER and DROP USER SQL statements.

When you create a PointBase database using PointBase Commander, PointBase Console, or the JDBC API, the system creates a default user PBPUBLIC with the password PBPUBLIC who owns the default schema PBPUBLIC. Only the PBSYSADMIN, the database owner, or users with the PBDBA role may create new users. (See "SQL Security and Privileges" on page 90.)

You cannot connect to a database as a user who does not exist in the SYSUSERS table, which is one of the system tables in the POINTBASE schema. For a list of predefined system tables and their attributes within the POINTBASE schema, please refer to “Appendix A: System Tables” of the *PointBase System Guide*.

Schema

Databases contain collections of independent schemas. A schema is a logical grouping of tables, indexes, triggers, routines, and other data objects under one qualifying name. Internationalization characteristics and user-level security can also be defined for schema objects.

When a database is created using PointBase Commander, PointBase Console, or the JDBC API, the PointBase RDBMS creates two schemas:

- An internal schema called POINTBASE, in which the system keeps all of the system catalogs and tables
- A default schema called PBPUBLIC

You cannot create any user-defined data objects within the POINTBASE schema. For a list of predefined system tables and their attributes within the POINTBASE schema, please refer to “Appendix A: System Tables” in the *PointBase System Guide*.

Previous Schema PUBLIC

In versions 4.1 and earlier, PointBase used the default schema, PUBLIC. By default, it also has the password and user, PUBLIC. These names will still remain effective in versions 4.2 and later; however, PointBase will now use PUBLIC for superficial purposes only. That is, you may still connect to the database using PUBLIC. But internally, PointBase converts the user and the password, PUBLIC, to PBPUBLIC every time you connect, and PointBase recognizes the schema, PUBLIC as if it were the schema, PBPUBLIC. Additionally, you cannot drop the superficial schema name, PUBLIC. However, you may create and later drop a *new* schema called PUBLIC, and PointBase will not affiliate it with the schema, PBPUBLIC.

Schema Owners

The PointBase predefined user, PBPUBLIC, with the password, PBPUBLIC, is the owner of the PBPUBLIC schema and has full access to all objects within this schema. The predefined user, PBSYSADMIN, has access to *all* objects in the database. (See "Predefined Users" on page 91.)

Unless you specify a different user explicitly, you automatically become the owner of a schema if you created it. The schema owner has full access privileges and must grant access privileges to other users for them to access that schema. PointBase recommends that you create new schemas with the same name as your user name (if you are the schema owner) or with the same names as the user who owns the schema. When you access the database, PointBase will automatically search for the schema with the same name as the current user, making this the current schema.

Schema Referencing

Data objects are mapped to the current schema by default, without the need for an explicit reference. The CURRENT_SCHEMA special register contains the name of the current schema. Please refer to the “SQL Scalar and Aggregate Functions” chapter in this guide for more information about the CURRENT_SCHEMA special register.

In databases with multiple schemas, data objects must explicitly reference the schema for which they are intended. If no explicit reference is made, PointBase automatically tries to associate the data object with the current schema. If the data object cannot be logically associated with the current schema, it references the default (PBPUBLIC) schema.

In databases with multiple schemas, when referencing a data object that is not in the current schema, you must append the schema name to the data object name, separated by a period. For example, if you have a schema named Employee_Info, which contains a table named Employees. Then, you must refer to that table in the following way:

```
Employee_Info.Employees
```

Managing Schemas

To manage schemas, use the CREATE SCHEMA and DROP SCHEMA SQL statements. CREATE SCHEMA initially creates a schema and conversely, DROP SCHEMA drops a schema. The user that creates the schema owns the schema unless the optional AUTHORIZATION qualifier is used to specify another user. The schema owner can grant applicable privileges to the appropriate users.

Table

A table comprises of a number of column objects and contains rows of data. A row is a nonempty sequence of values that correspond to the column objects in the table. Every row of the same table has the same number of columns and contains a value for every column of that table.

The following are two types of tables used in the PointBase RDBMS:

- Base Table - a table whose data is actually stored in the database.
- Derived Table – a table obtained from other tables directly or indirectly through the evaluation of a query expression.

NOTE: Due to known limitations, it is highly recommended that you always use uppercase letters when specifying table names or column names wherever applicable.

View

Derived Tables or “Virtual Tables” are also known as Views. They provide an alternative way to look at the data of one or more tables. This virtual table or view derives its values from the evaluation of a query expression in a CREATE VIEW statement. The query expression can reference base tables, other views, aliases, etc. Essentially, a view is a stored SELECT statement, of which you can retrieve the results at a later time by querying the view as though it were a table. See also "CREATE VIEW" on page 119. A view can be read-only or updateable. **Currently, PointBase supports Read-Only Views.**

The definition of each view is stored in PointBase's system catalog SYSVIEWS. If no errors are encountered, PointBase adds the view name to the SYSVIEWS catalog table. Additionally, all referenced columns of all referenced tables will be added to the SYSVIEWTABLES catalog table.

Security for Views

Because a view is a type of table, you can grant privileges on it, and the privileges can be different than the privileges on any base table from which the view was derived. Unlike base tables, however, an owner of a view does not automatically have the authority to grant privileges on the view to others.

To grant privileges on the view to others, you must have grant privileges on every referenced column and table in the view's query expression. If you have privileges revoked on any of the referenced columns or tables, you also have the same privileges revoked on the view.

Revoking privileges on a view using the RESTRICT option will raise an error, if any users of that view had the grant option privilege and they granted that privilege to other users. If you revoke privileges on a view using the CASCADE option, you will revoke all the users' privileges on that view. Likewise, you must verify if the view has any dependent views, and verify the privileges on those as well.

NOTE: Revoking privileges on a view does not affect base table privileges.

Column

Each PointBase table can have a maximum limit of 32,000 columns and a minimum of one. All values contained within a specific column are of the same data type and every column has an associated default value. The system uses the default value when data is entered into a table without specifying a value for the column. The default value for a column is NULL unless the column specifies the NOT NULL constraint or a different default value. If a column specifies the NOT NULL constraint and has no default value defined, then you must specify a value for this column whenever data is inserted or updated in the table.

NOTE: Due to known limitations, it is highly recommended that you always use uppercase letters when specifying table names or column names wherever applicable.

IDENTITY Property for Autoincrement

PointBase has *autoincremental* capability using the IDENTITY property. By defining it for a column (making it an IDENTITY column), PointBase or you can generate values for every row in a table. You can define it for only a column that has either of the data types:

- INTEGER
- SMALLINT
- NUMERIC
- DECIMAL

You can create a table with an IDENTITY column or you can add an IDENTITY column at a later time using the ALTER TABLE statement. Please note, however, each table may have only one IDENTITY column, and once you have created a table with an IDENTITY column or added it at a later time, you cannot update any values in the IDENTITY column.

PointBase Generated Values

If you create, alter, or insert into a table without specifying a value for the IDENTITY column, PointBase automatically assigns incremental values to every row in a table. If you allow PointBase to generate the values, the default value for the first row is 1 (one). By default, PointBase will also assign increments of 1 to the rows that follow. For example, PointBase automatically assigns the default value of 1 to the first row of the table and continues to give the value 2 for the second row, 3 to the third row, and so on. (See “identity_property” on page 111.) If you *insert* a row value into an IDENTITY column without specifying a value for the IDENTITY column, PointBase will continue to generate incremental values based on the highest value assigned for the column—even if the highest value was deleted. (See “insert_column_list” on page 156.)

User-defined Values

You can also opt to specify the values yourself. If you are *creating* or *adding* the IDENTITY column and specifying its values, you must specify the value of the first row, and you must specify the incremental value, which affects the rest of the rows in the table. (See “identity_property” on page 111.) If you are *inserting* a row value into an IDENTITY column, you must specify only the column value. PointBase will continue to generate incremental values based on the highest value assigned for the column—even if the highest value was deleted. (See “insert_column_list” on page 156.) Please note that PointBase recommends that you allow PointBase to generate the IDENTITY column values when inserting new rows.

Deleting Rows

Additionally, PointBase supports deleting rows from an IDENTITY column. However, once you delete a row value from an IDENTITY column, PointBase will not generate that value again; PointBase generates only *unique* values. **PointBase will generate incremental values based on the highest row value assigned for the column—even if the highest value was deleted.**

SQL Data Types

This chapter describes all of the SQL data types that PointBase supports. Data types define what type of data a column can contain. The following sections describe each PointBase data type in detail and discuss converting data types. Tables are provided at the end of the chapter to show the mappings between PointBase data types and industry standard and other common non-standard data types.

Data Types

PointBase supports the following data types for its column and parameter declarations.

- CHARACTER [(length)] or CHAR [(length)]
- VARCHAR (length)
- BOOLEAN
- SMALLINT
- INTEGER or INT
- DECIMAL [(p[,s])] or DEC [(p[,s])]
- NUMERIC [(p[,s])]
- REAL
- FLOAT(p)
- DOUBLE PRECISION
- DATE
- TIME
- TIMESTAMP
- CLOB [(length)] or CHARACTER LARGE OBJECT [(length)] or CHAR LARGE OBJECT [(length)]
- BLOB [(length)] or BINARY LARGE OBJECT [(length)]

CHARACTER [(length)] or CHAR [(length)]

The CHARACTER data type accepts character strings, including Unicode, of a fixed length. The length of the character string should be specified in the data type declaration; for example, CHARACTER(n) where n represents the desired length of the character string. If no length is specified during the declaration, the default length is 1.

The minimum length of the CHARACTER data type is 1 and it can have a maximum length up to the table page size. Character strings that are larger than the page size of the table can be stored as a Character Large Object (CLOB).

NOTE: CHARACTER(0) is not allowed and raises an exception.

If you assign a value to a CHARACTER column containing fewer characters than the defined length, the remaining space is filled with blanks characters. Any comparisons made to a CHARACTER column must take these trailing spaces into account.

Attempting to assign a value containing more characters than the defined length results in the truncation of the character string to the defined length. If any of the truncated characters are not blank, an error is raised.

Character String Examples:

CHAR(10) or CHARACTER(10)

- Valid

```
'Race car'
'RACECAR'
'24865'
'1998-10-25'
'1998-10-25   ' (Blank characters are truncated)
```

- Invalid

```
24865
1998-10-25
'Date: 1998-10-25'
```

VARCHAR (length)

The VARCHAR data type accepts character strings, including Unicode, of a variable length is up to the maximum length specified in the data type declaration.

A VARCHAR declaration must include a positive integer in parentheses to define the maximum allowable character string length. For example, VARCHAR(n) can accept any length of character string up to *n* characters in length. The length parameter may take any value from 1, to the current table page size minus 42 bytes. For example, the maximum length parameter for a page size of 4k (4096) would be 4096 minus 42 bytes, equaling 4054 bytes. Attempting to assign a value containing more characters than the defined maximum length results in the truncation of the character string to the defined length. If any of the truncated characters are not blank, an error is raised.

NOTE: VARCHAR(0) is not allowed and raises an exception.

If you need to store character strings that are longer than the current table page size, the Character Large Object (CLOB) data type should be used.

Examples

VARCHAR(10)

- Valid

```
'Race car'
'RACECAR'
'24865'
'1998-10-25'
'1998-10-25   '
```

- Invalid

```
24865
1998-10-25
'Date: 1998-10-25'
```

BOOLEAN

The BOOLEAN data type accepts a single value that can be TRUE or FALSE. No parameters are required when declaring a BOOLEAN data type.

Use the case insensitive keywords TRUE or FALSE to assign a value to a BOOLEAN data type. Comparisons using the BOOLEAN data type should also use these keywords. If you attempt to assign any other value to a BOOLEAN data type, an error is raised.

Examples

BOOLEAN

- Valid

```
TRUE
true
True
False
```

- Invalid

```
1
0
Yes
No
```

SMALLINT

The SMALLINT data type accepts a 16 bit signed integer value with an implied scale of zero. It stores any integer value between the range 2^ -15 and 2^15 -1. Attempting to assign values outside this range causes an error.

If you assign a numeric value with a precision and scale to a SMALLINT data type, the scale portion truncates, without rounding.

NOTE: To store values beyond the range (2^-15) to (2^15)-1, use the INTEGER data type.

Examples

SMALLINT

- Valid

```
-32768
0
-30.3 (digits to the right of the decimal point are trun-
cated)
32767
```

- Invalid

```
-33,000,567
-32769
32768
1,897,536,000
```

INTEGER or INT

The INTEGER data type accepts a 64-bit signed integer value with an implied scale of zero. It stores any integer value between the range 2^ -31 and 2^31 -1. Attempting to assign values outside this range causes an error.

If you assign a numeric value with a precision and scale to an INTEGER data type, the scale portion truncates, without rounding.

NOTE: To store integer values beyond the range (2^-31) to (2^31)-1, use the DECIMAL data type with a scale of zero.

Examples

INTEGER or INT

- Valid

```
-2147483648
-1025
0
1025.98 (digits to the right of the decimal point are
truncated)
2147483647
```

- Invalid

```
-1,025,234,000,367
-2147483649
2147483648
1,025,234,000,367
```

BIGINT

The BIGINT data type can accept numeric values up to 8 bytes. It can be used in place of the LONG data type. It stores any integer value between the range of 9223372036854775807 and -9223372036857447808. Attempting to assign values outside this range causes an error.

Examples

BIGINT

- Valid

```
-3372036857447808
-857447808
0
23372036854775807
```

- Invalid

```
-1,025,234,000,367
-2147483649
2147483648
1,025,234,000,367
```

DECIMAL [(p[,s])] or DEC [(p[,s])]

The DECIMAL data type accepts fixed-precision decimal values, for which you may define a precision and a scale in the data type declaration. The precision is a positive integer that indicates the number of digits that the number will contain. The scale is a positive integer that indicates the number of these digits that will represent decimal places to the right of the decimal point. The scale for a DECIMAL cannot be larger than the precision.

DECIMAL data types can be declared in one of three different ways. The declaration of it controls how the number is presented to an SQL query, but not how it is stored.

- DECIMAL – Precision defaults to 38, Scale defaults to 0
- DECIMAL(p) – Scale defaults to 0
- DECIMAL(p, s) – Precision and Scale are defined by the user

In the above examples, *p* is an integer representing the precision and *s* is an integer representing the scale.

NOTE: If you exceed the number of digits expected to the left of the decimal point, an error is thrown. If you exceed the number of expected digits to the right of the decimal point, the extra digits are truncated.

Examples

DECIMAL(10,3)

- Valid

```
1234567
1234567.123
1234567.1234 (Final digit is truncated)
-1234567
-1234567.123
-1234567.1234 (Final digit is truncated)
```

- Invalid

```
12345678
12345678.12
12345678.123
-12345678
-12345678.12
-12345678.123
```

NUMERIC [(p[,s])]

PointBase treats the NUMERIC data type in exactly the same way as the DECIMAL data type.

REAL

The REAL data type accepts single-precision floating point number values, up to a precision of 64. No parameters are required when declaring a REAL data type. If you attempt to assign a value with a precision greater than 64 an error is raised.

Examples

REAL

- Valid

```
-2345
0
1E-3
1.245
12345678901234567890123456789O
```

- Invalid

```
123,456,789,012,345,678,901,234,567,890,123
```

FLOAT(p)

The FLOAT data type accepts a single or double precision floating point number value, for which you may define a precision up to a maximum of 64. If no precision is specified during the declaration, the default precision is 64. Attempting to assign a value lager than the declared precision will cause an error to be raised.

Examples

FLOAT(8)

- Valid

```
12345678
1.2
123.45678
-12345678
-1.2
-123.45678
```

- Invalid

```
123456789
123.456789
-123456789
-123.456789
```

DOUBLE PRECISION

The REAL data type accepts a double precision floating point value, up to a precision of 64. No parameters are required when declaring a DOUBLE PRECISION data type. If you attempt to assign a value with a precision greater than 64 an error is raised.

Examples

DOUBLE PRECISION

- Valid

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890

-12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

- Invalid

```
123,456,789,012,345,678,901,234,567,890,123,123,456,789,

012,345,678,901,234,567,890

-123,456,789,012,345,678,901,234,567,890,123,123,456,789,

012,345,678,901,234,567,890
```

DATE

The DATE data type accepts date values, consisting of year, month, and day. No parameters are required when declaring a DATE data type. Date values should be specified in the form: YYYY-MM-DD. However, PointBase will also accept single digits entries for month and day values.

Month values must be between 1 and 12, day values should be between 1 and 31 depending on the month and year values should be between 0 and 9999.

Values assigned to the DATE data type should be enclosed in single quotes, preceded by the case insensitive keyword DATE; for example, DATE '1999-04-04'.

Examples

DATE

- Valid

```
DATE ’1999-01-01’
DATE ’2000-2-2’
date ’0-1-1’
```

- Invalid

```
DATE ’1999-13-1’
date ’2000-2-30’
’2000-2-27’
date 2000-2-27
```

TIME

The TIME data type accepts time values, consisting of hours, minutes, and seconds. No parameters are required when declaring a TIME data type. Date values should be specified in the form: HH:MM:SS. An optional fractional value can be used to represent nanoseconds.

The minutes and seconds values must be two digits. Hour values should be between zero 0 and 23, minute values should be between 00 and 59 and second values should be between 00 and 61.999999.

Values assigned to the TIME data type should be enclosed in single quotes, preceded by the case insensitive keyword TIME; for example, TIME '07:30:00'.

Examples

TIME

- Valid

```
TIME ’00:00:00’
TIME ’1:00:00’
TIME ’23:59:59’
time ’23:59:59.99’
```

- Invalid

```
TIME ’00:62:00’
TIME ’00:3:00’
TIME ’23:01’
’24:01:00’
```

TIMESTAMP

The TIMESTAMP data type accepts timestamp values, which are a combination of a DATE value and a TIME value. No parameters are required when declaring a TIMESTAMP data type. Timestamp values should be specified in the form: YYYY-MM-DD HH:MM:SS. There is a space separator between the date and time portions of the timestamp.

All specifications and restrictions noted for the DATE and TIME data types also apply to the TIMESTAMP data type.

Values assigned to the TIMESTAMP data type should be enclosed in single quotes, preceded by the case insensitive keyword TIMESTAMP; for example, TIMESTAMP '1999-04-04 07:30:00'.

Examples

TIMESTAMP

- Valid

```
TIMESTAMP ‘1999-12-31 23:59:59.99’
TIMESTAMP ‘0-01-01 00:00:00’
```

- Invalid

```
1999-00-00 00:00:00
TIMESTAMP ‘1999-01-01 00:64:00’
```

CLOB [(length)] or CHARACTER LARGE OBJECT [(length)] or CHAR LARGE OBJECT [(length)]

The Character Large Object (CLOB) data type accepts character strings longer than those that are allowed in the CHARACTER [(length)] or VARCHAR (length) data types. The CLOB declaration uses the following syntax to specify the length of the CLOB in bytes:

```
n [K | M | G]
```

In the above syntax, n is an unsigned integer that represents the length. K, M, and G correspond to Kilobytes, Megabytes or Gigabytes, respectively. If K, M, or G is specified in addition to n, then the actual length of n is the following:

- K = n * 1024
- M = n * 1,048,576
- G = n * 1,073,741,824

The maximum size allowed for CLOB data types is 2 gigabytes. If a length is not specified, then a default length of one byte is used. CLOB values can vary in length from one byte up to the specified length.

NOTE: The CLOB data type supports Unicode data.

BLOB [(length)] or BINARY LARGE OBJECT [(length)]

The Binary Large Object (BLOB) data type accepts binary values. The BLOB declaration uses the following syntax to specify the length in bytes:

```
n [K | M | G]
```

In the above syntax, *n* is an unsigned integer that represents the length. K, M, and G correspond to Kilobytes, Megabytes or Gigabytes, respectively. If K, M, or G is specified in addition to *n*, then the actual length of n is the following:

- K = n * 1024
- M = n * 1,048,576
- G = n * 1,073,741,824

The maximum size allowed for BLOB data types is 2 gigabytes. If a length is not specified, then a default length of one byte is used. BLOB values can vary in length from one byte up to the specified length.

NOTE: BLOB data types cannot be used with SQL scalar functions.

Data Conversions and Assignments

The PointBase database allows two types of data conversions - implicit and explicit. An implicit data conversion is automatically performed between data types that are in the same data type family. Table 1 describes the data type families supported by PointBase. Implicit data conversions are performed as needed and are transparent to the user.

PointBase handles explicit data conversion using the SQL Scalar CAST function. This function converts a value from one PointBase data type to another in the same data type family.

Table 1: Data Type Families and Data Types

Data Type Family	*Data Types*
Character String	CHARACTER, VARCHAR, CLOB
Boolean	BOOLEAN
Binary String	BLOB
Date Time	DATE, TIME, TIMESTAMP
Number	SMALLINT, INTEGER, DECIMAL, NUMERIC, REAL, FLOAT, DOUBLE

Table 2: Mapping Standard Data Types to PointBase SQL Data Types

JDBC Data Types	*Java Data Types*	*PointBase SQL Data Types*
BIT	boolean	boolean
TINYINT	byte	smallint
SMALLINT	short	smallint
INTEGER	int	integer

Table 2: Mapping Standard Data Types to PointBase SQL Data Types

JDBC Data Types	*Java Data Types*	*PointBase SQL Data Types*
BIGINT	long	numeric/decimal
FLOAT	double	float
REAL	float	real
DOUBLE	double	double
NUMERIC	java.math.BigDecimal	numeric
DECIMAL	java.math.BigDecimal	decimal
CHAR	String	char
VARCHAR	String	varchar
LONGVARCHAR	String	clob
DATE	java.sql.Date	date
TIME	java.sql.Time	time
TIMESTAMP	java.sql.Timestamp	timestamp
BINARY	byte[]	blob
VARBINARY	byte[]	blob
LONGVARBINARY	byte[]	blob
BLOB	Blob	blob
CLOB	Clob	clob

PointBase also supports other non-SQL standard data types. Table 3 describes the mapping of non-SQL standard data types from other database vendors to PointBase data types.

Table 3: Mapping Non-standard Data Types to PointBase SQL Data Types

Oracle Data Types	*Sybase and Microsoft Data Types*	*DB2 Data Types*	*PointBase Data Types*
NUMBER			DECIMAL
	TINYINT		SMALLINT
VARCHAR2			VARCHAR
LONGVARCHAR	TEXT		CLOB
LONG			CLOB
			CLOB
RAW			BLOB
LONGRAW			BLOB
	BINARY		BLOB
	VARBINARY		BLOB
	IMAGE		BLOB
		CHAR for BIT DATA	BLOB
		VAR CHAR for BIT DATA	BLOB

SQL Scalar and Aggregate Functions

This chapter describes the SQL Scalar Functions supported in PointBase. PointBase provides these ready to use functions to perform in-statement operations when querying or inserting data into the database. For example, you can use the CAST function to convert data types to other data types or use a numeric function to perform calculations. The following sections describe the behavior of these functions and examples of how to use them.

NOTE: Unless specified otherwise, when applying any of the following functions to a column containing NULLS, the NULL rows are not counted or used and the following warning is given:

java.sql.SQLWarning: Warning--null value eliminated in set function

To eliminate this warning and ignore the NULLs in aggregate functions, you can use the DISTINCT keyword in front of the column reference, for example:

```
select (count(DISTINCT product_code)) from product_tbl
```

SQL Scalar Numeric Functions

The Scalar Numeric Function operates on numeric values (i.e. INTEGER, SMALLINT, DECIMAL, FLOAT, DOUBLE and NUMERIC data types). The PointBase database supports the following standard Numeric Functions:

- Multiplication
- Division
- Addition
- Subtraction

The numeric functions are evaluated in the following order. Numeric Functions within parentheses are evaluated from the innermost set of parentheses, following the same rules of precedence:

1. Multiplication (*) and division (/) from left to right
2. Addition (+) and subtraction (-) from left to right

Numeric Functions are calculated as floating point numbers with a precision of 17 significant digits (and a rounding error). However, if you use these functions when inserting or updating data the accuracy is dependent up on the data type of the column for which the data is intended.

Examples

2 + 3 * 4 / 2 = 8

2 + (3 * 4) / 2 = 8

2 + 3 / 2 = 3.5

100/3 = + 3 / 2 = 33.333333333333333

SQL Scalar Character String Functions

Scalar Character String Functions operate on character strings. These functions all return either character strings or numeric values. PointBase currently supports the following functions.

CONCATENATION

The concatenation operator (||) joins the values of two or more character strings into a single string. You may use the concatenated string expression anywhere you would use a character string and there is no limit to the number of string expressions you can concatenate. The following is the CONCATENATION Function syntax:

```
string_value || string_value [{|| string_value}...]
```

Examples:

```
'$' || ' ' || '150' ----> '$150'
```

```
SELECT order_num, sales_tax_st_cd, 'Shipping Cost', '$' || shipping_cost FROM order_tbl
WHERE shipping_cost > 300 AND UPPER(sales_tax_st_cd) NOT LIKE '%FL' ORDER BY order_num
ASC;
```

SUBSTRING

The SUBSTRING Function extracts a specified portion of the character string on which it is operating. The following is the SUBSTRING Function syntax:

```
SUBSTRING (string_value FROM start [FOR length])
```

In the previous syntax, the start variable is an integer that represents the starting position for the sub string. The first character in a string is considered to be position 1. The length variable is optional and indicates the length of the sub string; if it is missing, the SUBSTRING Function returns the characters from the start position to the end of the character string.

Examples

```
SUBSTRING('George Valentie' FROM 3) ----> 'orge Valentine'
SUBSTRING('George Valentie' FROM 3 FOR 2) ----> 'or'
```

CHARACTER_LENGTH

The CHARACTER_LENGTH function returns the length of a character string as the numeric data type. There are two syntax variations for the CHARACTER_LENGTH function:

1. CHARACTER_LENGTH (string_value)

2. CHAR_LENGTH (string_value).

Examples

```
CHAR_LENGTH('George Valentine') ----> 16
CHARACTER_LENGTH('$150') ----> 4
```

POSITION

The POSITION function searches for a specified string pattern in another string. If the pattern is found, a value is returned that indicates the beginning position of the location of the pattern. If the pattern is not found, then a value of zero is returned. If the pattern is a string length of zero (0, a NULL string), then a value of one is returned. All returned values are of the numeric data type. The following illustrates the syntax for the POSITION Function:

```
POSITION (string_pattern IN string_value)
```

Examples

```
POSITION('Valentine' IN 'George Valentine') ----> 8
POSITION('' IN 'George Valentine') ----> 1
```

TRIM

The TRIM function allows you to strip trailing and/or leading characters from a character string. The following illustrates the syntax for the TRIM Function:

```
TRIM (LEADING | TRAILING | BOTH 'character' FROM string_value)
```

Although it is common only to strip a blank characters (’ ’) from the start and ends of character strings, using the TRIM function you can strip any character. The character variable, enclosed in single quotes, represents the character that is to be stripped from the character string. The keywords LEADING, TRAILING, and BOTH indicate whether you strip the character variable from the front of the character string, at the end of the character string, or both.

Examples

```
TRIM (LEADING ‘ ‘ FROM ‘ George Valentine ‘)
----> 'George Valentine '

TRIM (TRAILING ‘ ‘ FROM ‘ George Valentine ‘)
----> ‘ George Valentine ‘

TRIM (BOTH ‘ ‘ FROM ‘ George Valentine ‘)
----> ‘George Valentine’

TRIM (LEADING ‘$’ FROM ‘$150’)
----> ‘150’
```

UPPER and LOWER

The UPPER function returns the value specified in the character string entirely in upper case letters, regardless of the initial capitalization of the character string. The LOWER Function returns the value specified in the character string entirely in lower case letters, regardless of the initial capitalization of the character string variable. The following syntax is used for the Case Functions:

```
UPPER(string_value)
LOWER(string_value)
```

Examples

```
LOWER(’George Valentine’) ----> ’george valentine’
UPPER(’George Valentine’) ----> ’GEORGE VALENTINE’
```

SQL Scalar Date/Time Functions

The SQL Scalar Date Time Functions operate on date/time values and return of date/time values. PointBase supports the following Date/Time Functions.

CURRENT_DATE

The CURRENT_DATE Function returns the current system date from the machine that is hosting the PointBase database as a DATE data type. You may use the CURRENT_DATE Function anywhere you specify a DATE value.

Example

```
UPDATE order_tbl SET shipping_date = CURRENT_DATE
```

If the current date is April 4, 1998, the CURRENT_DATE Function returns: 1998-04-04.

CURRENT_TIME

The CURRENT_TIME Function returns the current system time from the machine that is hosting the PointBase database as a TIME data type. You may use the CURRENT_TIME Function anywhere you specify a time value.

Example

if the current time is exactly 9:00 AM, the CURRENT_TIME Function returns: 09:00:00.

CURRENT_TIMESTAMP

The CURRENT_TIMESTAMP Function returns the current system date and time from the machine that is hosting the PointBase database as a TIMESTAMP data type. You may use the CURRENT_TIMESTAMP Function anywhere you specify a timestamp value.

Example

```
UPDATE order_tbl SET delivery_datetime = CURRENT_DATE
```

If the current date and time is 9:00 AM on April 4, 1998, the CURRENT_TIMESTAMP Function returns: 1998-04-04 09:00:00.

EXTRACT

The EXTRACT Function returns a portion of a DATE, TIME, or TIMESTAMP value. It extracts the year, month, or day from a DATE value; an hour, minute, or second from a TIME value; or any of these intervals from a TIMESTAMP value. The EXTRACT Function always returns a numeric data type. The following syntax is for the EXTRACT Function.

```
EXTRACT (extract_field FROM datetime_value)
```

Use one of the keywords YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND in place of the extract_field. Format the datetime_value inside the single quotes appropriately, according to the value the extract_field seeks.

Examples

```
EXTRACT(YEAR FROM 1998-04-01) ----> 1998
EXTRACT(MONTH FROM 1998-04-01) ----> 04
EXTRACT(DAY FROM 1998-04-01 09:00:00) ----> 01
EXTRACT(HOUR FROM 1998-04-01 09:00:00) ----> 09
EXTRACT(MINUTE FROM 09:00:00) ----> 00
EXTRACT(SECOND FROM 09:00:00) ----> 00
```

SQL Scalar CAST Function

The SQL Scalar CAST Function explicitly converts a value from one PointBase data type to another. To perform an explicit data conversion, use the following syntax for the SQL Scalar CAST Function.

```
CAST (value AS datatype)
```

Table 1 lists the data types that can be CAST into other data types. If there is a Y in the intersection of two data types, the CAST Function can perform an explicit conversion from the data type in the vertical axis to the data type on the horizontal axis.

Table 1: Converting Data Types With the CAST Function

	C	**VC**	**B**	**I**	**SI**	**DEC**	**N**	**R**	**F**	**DB**	**D**	**T**	**TS**	BB	CB
CHARACTER (C)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y
VARCHAR (VC)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y
BOOLEAN (B)	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N
INTEGER (I)	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
SMALLINT (SI)	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
DECIMAL (DEC)	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
NUMERIC (N)	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
REAL (R)	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
FLOAT (F)	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
DOUBLE (DB)	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N
DATE (D)	Y	Y	N	N	N	N	N	N	N	N	Y	N	Y	N	N
TIME (T)	Y	Y	N	N	N	N	N	N	N	N	N	Y	Y	N	N
TIMESTAMP (TS)	Y	Y	N	N	N	N	N	N	N	N	Y	Y	Y	N	N
BLOB (BB)	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N
CLOB (CB)	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	Y

NOTE: A VARCHAR(10) cast to CHAR(5) will be truncated at the 5th character. The system will display a warning if the truncated characters are nonwhite spaces.

The CAST function throws an exception if the data is not convertible, for example: CAST('a' AS INT) --------> Exception

SQL Scalar Routine Invocation

Using SQL Scalar Routine Invocation, you can call a pre-defined SQL Routine that returns a scalar value. The Routine Invocation can be used anywhere you use a scalar expression. The following syntax is for the Routine Invocation Function. For more information about creating SQL routines (functions and procedures) refer to "Appendix A: SQL Reference."

```
routine_name( [ SQL_argument_list ] )
```

Routine_name is the name of the routine (SQL Function or Procedure). SQL_argument_list consists of expressions separated by commas. Each expression will result in a SQL data type dependent on the routine called.

NOTE: If you use a Routine Invocation Function as a scalar expression, it must only return a single value, otherwise an error is raised.

Routine Determination

Routine determination is the process that determines the routine to invoke, based on the routine name, SQL argument list, and the current path of schemas. The routine name and SQL arguments make up the signature of the routine. It is possible that more than one routine could have the same signature. If more than one possible routine has the same signature, then PointBase uses a precedence list to match each argument of each routine, to determine which one is the best match.

Examples

```
DateConvert('01-02-1993')
```

SQL Aggregate Functions

SQL Aggregate Functions operate on complete sets of data and return a single result. PointBase supports five Aggregate Functions: AVG, COUNT, MAX, MIN, and SUM.

AVG

The AVG Function returns the average value for the column when applied to a column containing numeric data. The following is the syntax for the AVG Function.

```
AVG (column_name)
```

Example

```
SELECT AVG(commission_rate) FROM sales_rep_tbl
```

COUNT

The COUNT Function returns the number of rows in a specified result set. The following syntax is one form of the COUNT Function:

```
COUNT(*)
```

Example

```
SELECT COUNT(*) FROM sales_rep_tbl
```

The second form of the COUNT Function returns the number of rows in a result set where the specified column has a distinct, non-NULL value. The following syntax is the second form of the COUNT Function.

```
COUNT(DISTINCT column_name)
```

MAX

The MAX Function returns the data item with the highest value for a column when applied to a column containing numeric data. If you apply the MAX Function to a CHARACTER value, it returns the last value in the sorted values for that column. The following syntax is for the MAX Function.

```
MAX(column_name)
```

Example

```
SELECT MAX(commission_rate) FROM sales_rep_tbl
```

MIN

The MIN Function returns the data item with the lowest value for a column when applied to a column containing numeric data. If you apply the MIN Function to a CHARACTER value, it returns the first value in the sorted values for that column. The following syntax is for the MIN Function.

```
MIN(column_name)
```

Example

```
SELECT MIN(commission_rate) FROM sales_rep_tbl
```

SUM

The SUM Function returns the sum of all values in the specified column. The result of the SUM Function has the same precision as the column on which it is operating. The following syntax is for the SUM Function.

```
SUM(column_name)
```

Example

```
SELECT SUM(ytd_sales) FROM sales_rep_tbl
```

SQL Special Registers

The PointBase RDBMS supports the following list as special registers. These can be used anywhere a scalar/value expression is allowed.

- CURRENT_USER: is the current user on the system and is an SQL varchar data type of maximal length 128.
- CURRENT_SCHEMA: is the name of the current schema in use and is an SQL varchar data type of maximal length 128.
- CURRENT_DATABASE: is the name of the database in use and is an SQL varchar data type of maximal length 128.
- CURRENT_SESSION: gives the current session ID.
- CURRENT_PATH: is the list of schemas in the path of the current user. The return data type is an SQL varchar of undetermined length. The length depends upon the number of schema names in the path.

Indexes and Constraints

This chapter gives a brief outline of indexes and constraints in the PointBase RDBMS. Indexes and constraints are used to reinforce data integrity and increase database performance. Using indexes and constraints, you can access information from the database quicker and guarantee the referential integrity of information. The following sections describe indexes, keys, and constraints.

Indexes

An index is a set of ordered references to rows of a table. It can contain data from one or more columns of a table. An index improves the performance of data retrieval by reducing the number of physical pages that the database must access in order to read a row in the database. Because indexes store data in order, they also eliminate the need to create temporary storage for the ORDER BY clause if the relevant column is included in an index. Every index has a header, which contains the following information:

- the depth of the index
- number of leaf pages
- the selectivity factor

The PointBase RDBMS builds and maintains indexes without user intervention and provides current information to the query optimizer.

NOTE: Whenever you specify a unique constraint, PointBase creates a unique index automatically.

You can also create and drop an index using the CREATE INDEX and DROP INDEX statements. For information on the query optimizer refer to “Optimizer Usage in PointBase,” in this guide. For CREATE INDEX and DROP INDEX syntax refer to “SQL Reference” of this guide.

Although index management can improve performance, it may add overhead to INSERT and UPDATE operations. PointBase recommends calculating the correct balance so that the overhead does not exceed the benefits. (Consequently, you should create indexes on columns--only when they use the indexes in predicates--where the majority of results return a small number of rows.) Use the following formula to calculate the cost/benefit ratio:

```
# of rows selected / # of rows in table=cost-benefit
```

If the result is less than .10, then PointBase recommends you to create indexes. If the result is greater, then you should not create an index, as the overhead will cost more than its benefits.

Keys

In a database, a key consists of one or more columns of a table that have been granted specific properties. When defining a table or index, you specify the key (primary or foreign). PointBase supports the following types of keys:

Primary Key

The primary key is used as a master reference for columns defined as foreign keys in other tables. Foreign keys can only contain values defined in the Primary key to which they refer. A table can only have one primary key, and the key must contain only unique values without any NULL values. The table containing the primary key is referred to as the parent table.

Foreign Key

A foreign key associates values contained in one or more columns of a table to primary keys of other tables. The table containing the foreign key is referred to as the child table.

The child table references a parent table, which must contain a primary key. A column defined as a foreign key cannot contain NULL values. The values in a foreign key column must match either all the values, or a subset of the values in the referenced Primary Key. A foreign key cannot contain values that are not in the primary key to which it refers.

Constraints

Constraints are rules that the database enforces to improve data integrity. You can specify all of the following constraints at either the column level or at the table level in the PointBase RDBMS

Unique Constraint

A unique constraint forces a column to contain only unique values. PointBase allows NULL values in unique columns, unless you specify NOT NULL when creating or altering a table. When creating or altering a table, you must define unique constraints. However, you can also create a unique constraint automatically when you create a primary key. Although a table can contain any number of unique columns, only one can be the primary key.

NOTE: Whenever you specify a unique constraint, PointBase creates a unique index automatically.

Referential Constraint

You can use a referential constraint to link foreign key columns with primary key columns. You can define referential constraints as you create or alter a table.

Check Constraint

The body of a check constraint is a search condition. You can use a check constraint to make sure that a value going into a column meets the criteria of the search condition. Similar to the other constraints, you can define a check constraint when creating or altering a table. However, you can also use this constraint when updating a column(s) of a table. The value being inserted or modified (through an UPDATE) must cause the search condition to evaluate to TRUE, in order for the data to be inserted or updated.

Search Conditions and Predicates

This chapter describes search conditions and predicates in PointBase. Search conditions and predicates help return specific values from the database. To use a search condition, you must use it with an SQL statement. To use a predicate, you must use it with a search condition. You can specify certain criteria in a search condition and predicate for an SQL statement to perform to the database. The following sections describe search conditions and predicates and their behaviors in PointBase.

Search Conditions

A *search condition* specifies a condition of “TRUE”, “FALSE”, or “UNKNOWN” about a specific row. It is comprised of predicates associated with the logical operators: AND, OR, and NOT. The syntax for a search condition is as follows:

```
[NOT] {predicate | (search_condition)}
    [{AND | OR} [NOT] {predicate | (search_condition)}...]
```

Search conditions contained within parentheses first reads the values from left to right. The precedence order for the logical operators are: NOT, AND, and then OR. If more than one operator of the same precedence is used in a search condition, the optimizer will determine which one to execute before the other. If a search condition does not comprise any logical operators, then the result is the result of the predicate specified.

Simple search conditions

A search condition—in its simplest form—is a logical test that can be applied to each row. It takes the format of two value expressions and an operator and tests the relationship between the two values, for example:

```
value 1 > value 2
x > 2
```

Values

Any one of the values in a search condition may be one of the following:

- a constant
- the value in a column name that is used in the place of one of the value expressions
- a value derived from either one of these two values, using standard operators and non-aggregate functions, such as BALANCE + 10.

Operators

PointBase SQL supports all standard relational operators:

- equals (=)
- greater than (>)
- less than (<)
- not equal to (<>)
- less than or equal to (<=)
- greater than or equal to (>=)

Notice in a combined relation, for example, less than or equal to or greater than or equal to, the equal sign must be the last sign in the relation.

Complex search conditions

A complex search condition can contain multiple boolean expressions, linked by the keywords AND or OR. A boolean expression uses all the same syntax and operators as a boolean condition.

The AND keyword returns TRUE if the search conditions on both sides of the AND keyword return TRUE. If either one of the conditions return FALSE, the joined condition returns FALSE.

The OR keyword returns TRUE if the expressions on either side of the OR keyword return TRUE. If both conditions return FALSE, the joined condition returns FALSE.

The search conditions that make up a complex search condition return according to four rules of precedence:

1. Conditions within parentheses
2. Conditions joined by an AND keyword
3. Conditions joined by an OR keyword
4. Conditions prefixed by a NOT keyword

Order of Evaluation

Any set of expressions within parentheses return first. If there are more than one set of conditions within parentheses in a boolean expression, the sets evaluate from right to left. If sets of conditions within parentheses contain other sets of conditions within parentheses, the innermost sets evaluate first. Although it is not required that complex search conditions, which contain multiple sets of search conditions, use parentheses to separate the conditions, it is highly recommended to improve the readability of the conditions.

The AND, OR, and NOT keywords are reflexive, which means that the ordering of the expressions in a boolean expression does not matter. Regardless of the order, you receive the same result. A code optimizing program may execute the AND, OR, and NOT keywords differently than they appear in a boolean expression, but the boolean expression returns the same result.

Examples

In the first example below, the statement executes from left to right, because AND has a higher precedence than OR. In the second example, the search condition in parenthesis executes first.

1. emp_id > 201 AND d_name = ‘engineering’ OR d_name = ‘research’
2. emp_id > 201 AND (d_name = ‘engineering’ OR d_name = ‘research’)

Predicates

A predicate is an SQL expression that evaluates a search condition that is either TRUE, FALSE or UNKNOWN. TRUE means the expression is correct. FALSE means the expression is incorrect. UNKNOWN means the expression is neither TRUE nor FALSE. All SQL values used in a predicate must be of a compatible data type (family) for comparison.

PointBase supports the following types of predicates:

- comparison (=, <>, <, >, <=, >=, **!=**)
- BETWEEN
- LIKE
- EXITS | NOT EXISTS
- IN | NOT IN
- NULL

NOTE: PointBase does not support multi-valued predicates.

COMPARISON

The COMPARISON predicates compare two values. If either value is NULL, then the result of the predicate is UNKNOWN.

NOTE: When comparing two string values, PointBase ignores any spaces that trail after the string. PointBase ignores trailing spaces in queries and in the table. This behavior supports the ANSI standard, however, it may vary with other database vendors.

Table 1: Comparison Predicate Symbols

Comparison Symbol	*Symbol Description*	*Result Description*
=	equal to	This symbol results to TRUE if both values are the same.
<> or !=	not equal to	This symbol results to TRUE if the first value is equal to the second value.
<	less than	This symbol results to TRUE if the first value is less than the second value.
>	greater than	This symbol results to TRUE if the first value is greater than the second value.
<=	less than or equal to	This symbol results to TRUE if the first value is less than or equal to the second value.
>=	greater than or equal to	This symbol results to TRUE if the first value is greater than or equal to the second value.

Examples

The following are examples of using the comparison predicates. The results (TRUE, FALSE, or UNKNOWN) of the predicates are based on the values of the column.

- `emp_id = 200 ---> TRUE if emp_id is 200`
- `emp_manager <> 'Jones' ----> TRUE if the manager is not JONES`
- `salary > 50000 ----> TRUE if salary is greater than $50,000`

BETWEEN

The BETWEEN predicate determines if a value is between a range of values. The BETWEEN predicate is a short hand notation. It is equivalent to saying the value is greater than or equal to the beginning range and less than or equal to the ending range. For example, value1 BETWEEN value2 AND value3 is equivalent to the following search_condition:

```
value1 >= value2 AND value1 <= value3
```

The following is the syntax for a between predicate:

```
expression [NOT] BETWEEN literal AND literal
```

Examples

In the first example below, the system returns TRUE if the emp_deptid is between 200 and 1000. In the second example, the system returns TRUE if emp_managerid is less than 100 or greater than 400.

1. `emp_deptid BETWEEN 200 AND 1000`
2. `emp_managerid NOT BETWEEN 100 AND 400`

LIKE

The LIKE predicate searches a string to determine if the string has a particular pattern. The pattern is a string with a combination of the following special characters: underscore character, **_** and percent sign, **%**. If the value of any of the arguments is NULL, then the result is UNKNOWN. The following is the syntax for the LIKE predicate:

```
match_expression [NOT] LIKE pattern
```

match_expression

The match_expression is a string that will be searched to determine if the pattern specified can be found.

NOTE: The LIKE predicate is case-sensitive.

Examples

In the first example, the LIKE predicate looks for any row where the column contains a pattern of “engineer” as eight characters contained within the column. The percent sign represents any string of zero or more characters. In the second example, the LIKE predicate looks for all rows that do not contain a pattern of some character followed by ‘bc’ value for a column. The underscore character represents a single character. All other characters in both examples represent themselves.

1. `emp_description LIKE ‘%engineer%’`
2. `dept_description NOT LIKE ‘_bc’`

EXISTS | NOT EXISTS

These quantified operators verifies the existence of rows. The boolean result of an EXISTS or NOT EXISTS predicate is determined by the number of rows returned by the subquery. For EXISTS, the boolean result is TRUE if the subquery returns at least one row and FALSE if the subquery does not return any rows. For NOT EXISTS, the boolean result is TRUE if the subquery does not return any rows and FALSE if the subquery returns at least one row.

Notes

- PointBase supports any level of nested subqueries.
- PointBase allows a subquery to return multiple values only using EXITS, NOT EXISTS, IN, or NOT IN.
- Currently, PointBase does not support any form of the quantified operators, ANY or ALL, for example: =ANY, <=ANY, >=ALL, <>ALL,... etc.

Example

This example retrieves all cities, in which at least one sales representative works.

```
SELECT a.city
FROM office_tbl a
WHERE EXISTS
( SELECT *
FROM sales_rep_tbl b
WHERE a.office_num = b.office_num);
```

Results:

CITY
Miami
Atlanta
San Mateo
San Francisco
San Diego
Oakland
Detroit
New York

IN | NOT IN

You can use these predicate keywords to return a value list or a subquery.

Value List

The IN predicate determines if a value is TRUE for a list of values. The following is the syntax for an IN predicate. The NOT IN predicate also follows the same format as the IN predicate.

```
SELECT|UPDATE|DELETE FROM table
    WHERE expression [NOT] IN (list_of_values)
```

The *list_of_values* can be represented **only by literals with the IN predicate**. The NOT IN predicate returns a TRUE value only when it does not find the *list_of_values* specified.

Example

In the following example, the IN predicate returns TRUE if the “emp_deptid” is any of the values 10, 100, or 1000.

```
emp_deptid IN (10,100,1000)
```

Subquery

IN or NOT IN can compare a single value of each row of a table to a value from potentially multiple result rows from a subquery. IN returns TRUE, if at least one of the resultant subquery row values is equal to the expression; it returns FALSE otherwise. NOT IN returns TRUE if all of the resultant subquery row values are not equal to the expression.

Example

This example retrieves the names of all sales reps working in the western region.

```
SELECT a.first_name, a.last_name
FROM sales_rep_tbl a
WHERE a.office_num IN
( SELECT b.office_num
FROM office_tbl b
WHERE b.region = 'Western');
```

Results:

FIRST_NAME	LAST_NAME
Heather	Smith
George	Valentine
Raymond	Brown
Jack	Smith

NULL

The NULL predicate determines if a column in a selected row contains the SQL value: NULL. If the column value is NULL, then PointBase returns TRUE. The following is the syntax for the NULL predicate:

```
column_name IS [NOT] NULL
```

Examples

In the first example, the NULL predicate looks for any row where the column contains a NULL value. In the second example, the NULL predicate looks for all rows that do not contain a NULL value for a column.

1. `emp_dept IS NULL`
2. `emp_manager IS NOT NULL`

Transactions and Locks

This chapter describes the behavior and usage of transactions and locks in the PointBase RDBMS. By understanding how transactions and locks work in PointBase, you can maximize concurrent database utilization while maintaining appropriate data integrity for your application. The following sections describe transactions, locking concepts, and the different isolation levels that PointBase supports.

Transactions

A transaction is the primary mechanism used by PointBase to protect the integrity of data that can be accessed from the database. All of the changes (INSERT, UPDATE, DELETE) made to a database during a transaction are added to the database when the transaction commits.

A transaction implicitly starts if any Data Manipulation Language (DML) statement is executed, such as SELECT, INSERT, UPDATE, and DELETE, or if any Data Definition Language (DDL) statement is executed, such as CREATE TABLE, CREATE INDEX, etc. A transaction can be explicitly started by executing a `START TRANSACTION ISOLATION LEVEL` statement.

A transaction commits, when you issue a COMMIT statement. An application can also cancel all the changes made within a transaction by rolling back the transaction. A transaction rolls back when you issue a ROLLBACK statement or when an exception occurs.

If you set AUTOCOMMIT to on, a transaction will automatically commit after each statement (INSERT, UPDATE, DELETE) is completed. For example, a statement is completed when all result sets and/or update counts have been retrieved. To bound transactions explicitly, AUTOCOMMIT must be set to off.

A transaction is associated with a connection to the database. If multiple statements or threads use the same connection, they are part of the same transaction. If you decide to allow multiple threads to share one connection, you must synchronize all threads in order to commit the transaction.

For example, if one thread in a transaction issues commit, all the threads within the same transaction will be committed, invalidating threads that have not finished executing. PointBase recommends that you use one connection per thread.

Row Level Locking

When multiple connections or threads access the database concurrently, PointBase ensures the integrity of the data using row level locking. PointBase locks only the rows affected by an SQL statement rather than pages or tables, to ensure maximum concurrent activity. For example, when transaction T1 is updating row 10 in page 100, transaction T2 is able to update row 20 in the same page (100) or to read other rows in page 100.

Locks and Memory

PointBase stores all locks in memory. For efficient use of memory, you can limit the number of locks a single transaction can hold. The default limit is 2000, but you can change this using the locks.maxCount property in the `pointbase.ini` file. (Refer to the *PointBase System Guide* for more information about the pointbase.ini file, which you can use to configure the system properties.)

When a transaction reaches the specified limit of locks, PointBase automatically converts all of the row-level locks, to a table-level lock, reducing concurrency as a consequence. If the system cannot convert the row level locks to the table level lock within a reasonable time, the transaction is aborted. This may happen, if other transactions hold row-level locks on the same table.

Transaction Isolation Levels

The following section describes the transaction-isolation levels that PointBase supports. The transaction-isolation level defines the rules for releasing locks, allowing other users access to the row or table. By understanding PointBase isolation levels, you can understand how the system locking mechanism behaves.

NOTE: For all isolation levels, PointBase holds locks on rows that are modified until the end of the transaction.

READ_COMMITTED

When the transaction-isolation level is set to READ_COMMITTED, PointBase releases the lock on a row as soon it returns the row data to the user. For example, if a query returns 100 rows, the system locks the first row, reads the data and returns it to the user. Before locking and reading from the second row, PointBase releases the lock on the first row to minimize resource usage and maximize concurrency. After all the reads are complete, no locks are held.

SERIALIZABLE and REPEATABLE_READ

When the isolation level is set to SERIALIZABLE or REPEATABLE_READ, PointBase does not release locks on rows read until the end of the transaction. For example, if a query returns 100 rows, the system applies the lock on each row as it reads them. The system releases the locks only when it returns the data from all 100 rows to the user and the transaction is complete.

Recommended Isolation Level

The READ_COMMITTED isolation level gives maximum concurrency and minimum resource usage while providing the required data integrity for most applications. The default isolation level is READ_COMMITTED.

Distributed Transactions

This chapter summarizes distributed transaction processing (DTP) environments and how to use PointBase Embedded or Server in a DTP environment. Following the section, “PointBase’s Role in a DTP Environment,” this chapter briefly describes Sun’s Java Transaction API (JTA), the Java mapping for X/Open’s XA Specification, and also the JDBC API Extensions for distributed transactions. Finally, this chapter describes how to use PointBase Embedded or Server in a DTP environment by providing code snippets, explaining important restrictions, and supplying specific java classes that PointBase Embedded and Server implements for distributed transactions.

Although this chapter summarizes DTP concepts, it is only a summary, and it pertains specifically to PointBase Embedded and Server. For more information about the topics discussed in this chapter, PointBase recommends reading the following books or documents:

- X/Open’s *Distributed Transaction Processing: The XA Specification*
- Sun Microsystem’s JDBC API 2.0
- Sun Microsystem’s Java Transaction API (JTA) 1.0.1

Important Note

To successfully run your XA application with PointBase, you must obtain the following two JAR files from the Sun Microsystem’s website, “jta.jar” and “jdbc2_0-stdext.jar” and, include them in your classpath with the PointBase JAR’s.

- Download the “jta.jar “at http://java.sun.com/products/jta/index.html
- Download the “jdbc2_0-stdext.jar” at http://java.sun.com/products/jdbc/download.html#corespec21.

PointBase’s Role in a DTP Environment

According to the X/Open’s Distributed Transaction Processing (DTP) Model, a DTP environment specifies that application programs can use *Resource Managers* and a *Transaction Manager* to access multiple data sources through one *global transaction.* **PointBase RDBMS acts as a resource manager (RM) in a DTP environment**.

You can use PointBase in a DTP environment to write Enterprise JavaBeans that are transactional across multiple PointBase Servers. Workgroup environments, such as J2EE and J2SE where the data extends across multiple databases can benefit using PointBase, because the PointBase JDBC driver supports the 2-phase commit protocol used by the Java Transactional API (JTA).

Transaction Managers, Resource Managers, and Global Transactions

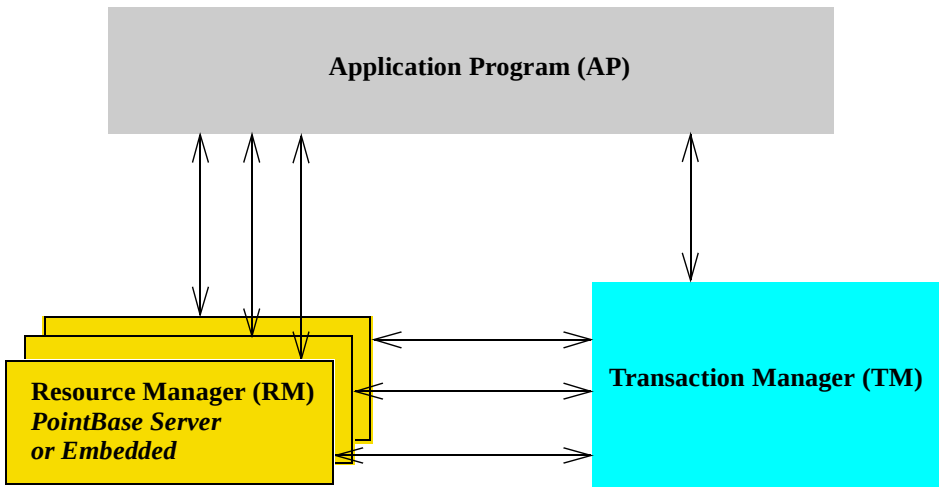
A transaction manager (TM) manages global transactions by ultimately deciding to commit, to rollback, or to recover global transactions. A global transaction is known as a unit of work. For example, an application can group multiple updates to several different data sources into one *unit of work*—a global transaction. A TM also associates resource managers with global transactions.

Each resource manager (RM) involved in a global transaction is unaware of any other RMs involved besides itself. For this reason, each RM requests and receives “permission” from the TM before it performs any work requested by an application. The RM also communicates all work it completes for a global transaction to the TM—whether it successfully completes or fails. With this information, the TM decides how to handle the global transaction.

NOTE: If any RM fails to successfully complete its part of a global transaction, all RMs involved in that global transaction must rollback the work for that particular global transaction.

Interaction Among DTP Components

The following illustration shows PointBase interacting with the application program and the transaction manager. Notice that the application program also interacts with the transaction manager. In this interaction, the application program defines the transaction boundaries or rules with the transaction manager. This guide, however, does not discuss this interaction. For more information about this topic, please refer to the relevant application program documentation. The following list describes the interaction flow among the application program (AP), the resource manager (RM), and the transaction manager (TM).



Java Transaction API (JTA)

The Java Transaction API (JTA) is part of the Sun J2EE standard which deals with distributed transactions. JTA defines a high-level transaction management interface intended for resource managers and transactional applications in DTP environments. PointBase implements the **XAResource** and **Xid** Interface of JTA, which maps the industry standard, X/Open XA Interface, to Java. The interface, X/Open XA Interface allows a transaction manager to manage operations performed by multiple resource managers, using the two-phase commit X/Open XA protocol.

JDBC 2.0 Optional Package API

Sun Microsystems created the JDBC API 2.0 Extensions, `java.sql.XAConnection` and `javax.sql.XADataSource`, so that JDBC drivers can support distributed transactions using the Java Transaction API's XAResource Interface. Refer to the JDBC 2.0 Standard Extension Specification for more details on JDBC API 2.0 Extensions (http://java.sun.com/products/ jdbc).

The PointBase JDBC driver supports distributed transactions by implementing the following interfaces. For unsupported methods, you can view both, "Appendix B: Unsupported JDBC 2.0 Methods in PointBase" and the section, "Unsupported in PointBase" at the end of this chapter.

API	*Description*
javax.transaction.xa.XAResource	This interface maps the industry standard X/Open XA Interface to Java. It defines APIs between the transaction manager and the resource manager. PointBase implements the JDBC standard for this interface. For more information about this interface, refer to http://java.sun.com/products/jta/javadocs-1.0.1/javax/transaction/xa/XAResource.html.
javax.transaction.xa.Xid	This interface defines the global transaction identification structure of the X/Open XA Interface. PointBase implements the JDBC standard for this interface. For more information about this interface, refer to http://java.sun.com/products/jta/javadocs-1.0.1/javax/transaction/xa/Xid.html.

API	Description
javax.sql.XADataSource	This is the JDBC Extension DataSource Interface for JTA's XAResource Interface. PointBase implements the class, `com.pointbase.xa.xaDataSource` for this interface. In addition to the JDBC standard methods, PointBase implements some of its own methods. For more information about PointBase's implementation of this interface, see the section, "Implementing javax.sql.XADataSource" on page 82. For more information about the standard JDBC interface, `javax.sql.XADataSource`, refer to http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/.
javax.sql.XAConnection	This interface is the JDBC Extension Connection Interface for JTA's XAResource Interface. PointBase uses the JDBC standard for this interface. For more information about this interface, refer to http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/.

Implementing javax.sql.XADataSource

The class, `com.pointbase.xa.xaDataSource` is the PointBase implementation of the JDBC Extension Interface, `javax.sql.XADataSource`. It is normally used with the Java Naming and Directory Interface (JNDI) for defining data sources in a DTP environment.

Because database vendors may support different data source properties, this section describes what PointBase supports. And, in addition to the standard JDBC Extension methods of `javax.sql.XADataSource`, PointBase has created its own methods, which this section also describes.

XADataSource and JNDI

Using `com.pointbase.xa.xaDataSource` to initialize an XADataSource object, is the first step to distributed transactions with PointBase. To initialize an XADataSource object, for example, you provide the database URL information, password, user name, etc., to get a connection with a database. However, you can also use JNDI.

Using JNDI, an application can find and access remote services, such as a database service across a network. After registering an XADataSource object with a JNDI naming service, an application can access that object to connect to the data source it represents.

With PointBase, you can use a JNDI naming service to manage data sources and connections. JNDI adds portability to the application code, for example, you do not have to include data source properties in the application code, such as the database name or the password. Also, you do not have to change the application code if you want to change a data source property. For example, instead of changing the application code to reflect a new user name, you can change the user name with the JNDI naming service.

Support for XADataSource Properties

Database vendors may vary when it comes to supporting XADataSource properties. For every supported XADataSource property, the database vendor must provide `set` and `get` methods. PointBase supports the following XADataSource methods for their respective XADataSource properties:

XADataSource Method	*Description*
`setDatabaseName( String url )`	Sets the databaseName property, defining the name of a particular database on a server. In PointBase, this defines the URL.
`String getDatabaseName()`	Returns the URL of a particular database on a server
`setDescription( String description )`	Sets the description property, defining a description of this data source
`String getDescription()`	Returns a description of this data source
`setPassword( String password )`	Sets the password property, defining the user's database password
`String getPassword()`	Returns the user's database password
`setUser( String user )`	Sets the user property, defining the user name
`String getUser()`	Returns the user name

Additional PointBase Methods

In addition to the standard methods of the javax.sql.XADataSource interface and javax.sql.DataSource for that matter, PointBase provides the following methods.

XADataSource Method	*Description*
`setCreateDatabase( boolean p_Create )`	Sets TRUE or FALSE. If set to TRUE, it creates a new database. Default is FALSE.
`Boolean getCreateDatabase()`	Returns TRUE if database exists and FALSE if it does not exist.

Using PointBase in a DTP Environment

This section describes how to use PointBase in a DTP environment. PointBase acts as the resource manager (RM) in a DTP environment, which reads or writes the data requested by an application in a global transaction. The following sections describe step-by-step how to use PointBase in a DTP environment.

Getting the XAResource Object

First, the transaction manager (TM) must get an XAResource object to start and end the association between an XAConnection object and a global transaction. To get an XAResource object, you must do the following:

Initialize XADataSource

Create a DataSource object to produce an XAConnection object. An XAConnection object is similar to a typical Connection object; however, an XAConnection object can obtain an XAResource object, which you need to perform a global transaction.

```
xaDataSource xads = new com.pointbase.xa.xaDataSource();
xads.setDatabaseName( url );
```

NOTE: Initializing a JNDI XADataSource compared to a JDBC XADataSource is similar. The following example initializes a JNDI XADataSource—assuming the XADataSource object has been stored with a JNDI naming service previously.

```
xaDataSource xads = (xaDataSource)ctx.lookup(“pointbase/datasource1”);
```

Get XAConnection Object

Get an XAConnection to “datasource1,” using the getXAConnection method. You need an XAConnection object to obtain an XAResource object.

```
XAConnection conxa = xads.getXAConnection();
```

Get Connection Object

Get a connection to the data source that "datsource1" represents, using the getConnection method. The application involved with the global transaction uses this connection to perform necessary work with the data source.

```
Connection con = conxa.getConnection();
```

Get XAResource Object

Get an XAResource object from the XAConnection object, using the getXAResource method. The TM uses the XAResource object to manage a global transaction and its association with an XAConnection object.

```
xaResource xrs = conxa.getXAResource();
```

NOTE: Only one XAResource object may exist for each XAConnection object. For example, if you call a second getXAResource method on the same XAConnection object, you obtain the same XAResource object.

Using the XAResource Object

Obtaining an XAResource object prepares you for starting and ending the association between a global transaction and an XAConnection object. The following examples describe the syntax that starts and ends the association between an XAConnection and a global transaction; **"xrs" is the XAResource object**:

- Start

```
xrs.start( Xid, Flag );
```

- End

```
xrs.end( Xid, Flag );
```

Xid

The TM assigns Xids to identify a global transaction. Xid consists of two parts, GTRID (transaction ID) and BQUAL (branch ID); both can be a maximum of 64 bytes. PointBase uses a constructor that requires the following parameters:

Parameter Name	*Parameter Type*
formatId	int
trId	byte[ ]
brId	byte[ ]

The following example describes the syntax that a TM can use to define an Xid using the PointBase class, `com.pointbase.xa.xaXid`:

```
Xid xid1 = new com.pointbase.xa.xaXid ( formatId, trId , brId );
```

Flags

The following “Flags” help start and end the association between a global transaction and an XAConnection object.

- TMNOFLAGS: indicates the start of a new global transaction. If you try to start a global transaction with an Xid that is currently in use, you receive the error, XAER_DUPID.

```
xrs.start( xid1, TMNOFLAGS );
```

- TMJOIN: indicates the joining of another existing global transaction branch. If you try to start a global transaction with an Xid that is currently in use, you receive the error, XAER_PROTO.

```
xrs.start( xid1, TMJOIN );
```

- TMRESUME: indicates resuming a suspended global transaction, which must have been previously suspended using the TMSUSPEND flag. You can use the TMRESUME flag in a different thread than the thread that suspended the global transaction, but it must use the same XAConnection.

```
xrs.start( xid1, TMRESUME );
```

- TMSUCCESS: indicates that a global transaction has completed successfully.

```
xrs.end( xid1, TMSUCCESS );
```

- TMFAIL: indicates that a global transaction failed. **You must rollback this global transaction.**

```
xrs.end( xid1, TMFAIL );
```

- TMSUSPEND: indicates suspending a global transaction. You must continue this global transaction with the flag, TMRESUME, within the same XAConnection.

```
xrs.end( xid1, TMSUSPEND );
```

Committing Global Transactions

Starting and ending a global transaction is similar to committing one, because you must commit a global transaction, using the XAResource object. After calling the `XAResource.end(Xid, TMSUCCESS)` method, you may commit the global transaction. The beginning of this chapter mentioned that TMs ultimately decide to commit a global transaction. TMs have the choice to use a “Two Phase Commit” or a “One Phase Commit” protocol. **PointBase (the RM) supports both protocols.**

One Phase Commit

A TM can use the one phase commit protocol, if it knows that only one RM in the DTP environment made changes to the shared data sources.

The following example describes the syntax for committing a global transaction using the one phase commit protocol; **“xrs” is the XAResource object**:

```
xrs.commit( xid1, true);
```

Two Phase Commit

A TM uses the two phase commit protocol, if multiple RMs made changes to shared data sources. In the first phase, (absent in the one phase commit protocol), the TM must confirm that all RMs involved in the global transaction have completed the necessary work successfully. If one RM does not complete its work successfully, the TM must rollback the global transaction. If the TM received a successful response from all RMs, however, the TM proceeds to phase two, committing the global transaction.

The following example describes the syntax for committing a global transaction using the two phase commit protocol; **"xrs" is the XAResource object**:

- Phase One

```
xrs.prepare( xid1 );
```

- Phase Two

```
xrs.commit( xid1, false);
```

Rolling Back Global Transactions

The TM must rollback a global transaction if any RM does not complete its work successfully or if the application requests that the TM rollback the global transaction. The following example describes the syntax for rolling back a global transaction; **"xrs" is the XAResource object**:

```
xrs.rollback( xid1 );
```

Recovering Global Transactions

A DTP environment or system may need to recover after a storage, connection path, or program failure. PointBase (the RM) provides the TM a list of Xids that it has prepared for commitment by the two phase commit protocol. The TM must recover the Xids by either committing them or rolling them back. The following example describes the syntax for recovering a global transaction; **"xrs" is the XAResource object**:

```
Xid[] xids = xrs.recover( Flags );
```

Flags

- TMSTARTSCAN: indicates the start of a new recovery process.

```
Xid[] xids = xrs.recover( TMSTARTSCAN );
```

- TMENDSCAN: indicates the end of a recovery process.

```
Xid[] xids = xrs.recover( TMENDSCAN );
```

- TMNOFLAGS: indicates that no other flags are specified. Use this flag only after you started the recovery scan.

```
Xid[] xids = xrs.recover( TMNOFLAGS );
```

- TMSTARTSCAN | TMENDSCAN: indicates the retrieval of all pending Xids.

```
Xid[] xids = xrs.recover( TMSTARTSCAN|TMENDSCAN );
```

Example

The following example describes a global transaction using a single thread and a single resource manager.

```
// initialize DataSource
com.pointbase.xa.xaDatasource xads = new com.pointbase.xa.xaDataSource()
xads.setDatabaseName( “jdbc:pointbase:embedded:xyz” );
xads.setCreateDatabase( true );

// get a connection object from DataSource
XAConnection conxa = xads.getXAConnection ( );
Connection con = conxa.getConnection();

// get a resource object from Connection
XAResource xrs = conxa.getXAResource ( );


// define an Xid
Xid xid = new com.pointbase.xa.xaXid ( “tr001” , “br001”);

// start a new transaction
xrs.start ( xid, XAResource.TMNOFLAG );

//  do something
Statement stmt = con.createStatement ( );
stmt.execute ( “ create table xxx ( c1 int )“ );
stmt.execute ( “ insert into xxx values ( 1 ) “ );

…

// end an Xid
xrs.end ( xid, XAResource.TMSUCCESS );

// commit the transaction
xrs.prepare ( xid );
xrs.commit ( xid, false );

//close the connection
con.close();
conxa.close();
```

Mixing Global and Local Transactions

Using PointBase, you can mix global and local transactions in the same XAConnection. If you execute an SQL statement and have not started a global transaction, (for example, getting an XAResource object) PointBase starts a local transaction automatically.

If you execute a local transaction, you must commit or rollback the transaction before you can start a global transaction.

NOTE: If autocommit is ON, local transactions commit automatically.

Unsupported in PointBase

PointBase does not support the following for distributed transactions:

- setTransactionTimeout: this method sets the transaction time-out value for this XAResource instance.
- getTransactionTimeout: this method gets the transaction time-out value set for this XAResource instance.

SQL Security and Privileges

This chapter describes PointBase security and privileges. Schemas are an integral part of security in PointBase. When creating a PointBase user, they do not have any access privileges to schemas or other data objects within the database. The PointBase RDBMS only permits the schema or database owner, PBSYSADMIN, or the PBDBA role to grant privileges to the schema and data objects within the schema. These users can grant privileges to the following data objects in the schema:

- Tables
- Columns
- Roles
- SQL Procedures and Functions

Table 1 describes the privileges that the previously mentioned users can grant to other users for tables and columns:

Table 1: User Privileges for Tables and Columns

Privilege Statements	***Privilege Description***
DELETE	Allows a user to delete rows from tables within the schema
INSERT	Allows a user to insert rows of data into tables within the schema
REFERENCES	Allows a user to set up references to primary keys within the schema
SELECT	Allows a user to select rows from tables within the schema
TRIGGER	Allows a user to create triggers on tables within the schema
UPDATE	Allows a user to update rows in tables within the schema
EXECUTE	Allows users to execute functions or stored procedures within the schema

Predefined Users

PointBase provides you with two predefined users. They each have their own purposes for the database. For example, anyone connected to the database using the predefined user, PBPUBLIC, has the capability to perform the following:

- connect to the database
- access the PBPUBLIC schema
- alter any objects within the default schema

In addition, PointBase provides one more type of predefined user. It has complete authority and privileges over all databases in the system. However, it does not have the privilege to modify or drop the system catalog tables.

Internal_System_Administrator (ISA)

This type of predefined user is for PointBase internal use only.

PBSYSADMIN

This type of predefined user has complete authority and privileges over all objects in the database, for example, it can create new users in the database. **However, it does not have the privilege to modify or drop the system catalog tables.** You may not grant additional privileges to the predefined user, PBSYSADMIN. To connect using PBSYSADMIN, you will initially have to use the password, “PBSYSADMIN.” After using it to connect, PointBase encourages you to change the password immediately.

PBPUBLIC

Another PointBase predefined user is PBPUBLIC. To connect using this type of user, you must use the default password, PBPUBLIC. With this type of user, you may access objects in the default schema, PBPUBLIC.

Previous User PUBLIC

In versions 4.1 and earlier, PointBase used the default user, PUBLIC. By default, it also has the password and schema, PUBLIC. These names will still remain effective in versions 4.2 and later; however, PointBase will now use PUBLIC for superficial purposes only. That is, you may still connect to the database using PUBLIC. But internally, PointBase converts the user and the password, PUBLIC, to PBPUBLIC every time you connect, and PointBase recognizes the schema, PUBLIC as if it were the schema, PBPUBLIC. Please note that the passwords, PBPUBLIC and PUBLIC act as the same password, so if you alter either password, it affects the other.

Granting and Revoking Privileges to Users

When you initially create a PointBase database, it automatically creates the user, PBPUBLIC with a password of PBPUBLIC. The PBPUBLIC user owns the default PBPUBLIC schema. For security reasons, PointBase does not recommend using this schema to store sensitive data. Like any PointBase user, PBPUBLIC must be granted the appropriate privileges to access data objects in schemas owned by other users.

PBPUBLIC users will own any new schema that they create unless otherwise specified while creating the schema. New users are then able to create their own new schema and grant appropriate privileges on objects in the schema that they own. All new users must be granted privileges to access the objects in the PBPUBLIC schema if this is required.

To grant the ability for a user to pass a privilege on to other users, you must specify the optional WITH GRANT OPTION qualifier when granting the privilege.

GRANT Syntax

```
GRANT <privilege-list>
ON <object>
TO <user> [ WITH GRANT OPTION ] | PUBLIC ]
[GRANTED BY <grantor>]
```

Use the GRANT statement to grant privileges on a data object. The following describes the GRANT statement syntax.

GRANT <Privilege-list> Syntax

```
privilege [ , privilege [ , privilege ]…] | ALL PRIVILEGES
```

<Privilege> Syntax

```
SELECT [ ( column-name [ , column-name ]…)]
| DELETE
| INSERT [ ( column-name [ , column-name ]…)]
| UPDATE [ ( column-name [ , column-name ]…)]
| REFERENCES [ ( column-name [ , column-name ]…)]
| TRIGGER [ ( column-name [ , column-name ]…)]
| EXECUTE
```

Usage Notes

- If you do not include one or more of these privileges in the GRANT statement, an error will be raised.
- If the optional “column-names” are not specified for the SELECT, INSERT, UPDATE, REFERENCES and TRIGGER privileges, the GRANT is a table-level grant that allows access to all present and *future* columns of the table.
- If you execute a GRANT statement that contains privileges that you don’t have or for which you do not have the right to grant, then PointBase raises an error.

ON <Object> Syntax

```
[ TABLE ] table-name
|SPECIFIC routine_type specific_routine-name
|routine_type routine_name (parameter_types_list)
[ TRIGGER ] trigger-name
```

Usage Notes

- You may only grant the EXECUTE privilege on an SQL Function or Procedure. The user cannot access tables that the SQL Function or Procedure uses.

TO <user/role-list> | [WITH GRANT OPTION] | PUBLIC Syntax

```
user [ , user ]... [WITH GRANT OPTION] | PUBLIC
```

Usage Notes

- If you do not specify WITH GRANT OPTION, the user cannot pass the same privilege on to other users. However, if you do specify WITH GRANT OPTION, you have given the user permission to pass on the privilege to other users.
- Granting a privilege to PUBLIC grants the privilege to **all** present and future users. PUBLIC is a keyword, representing all users in the database.
- If you grant a privilege twice, and one of the times—either first or second—you granted the optional WITH GRANT OPTION and the other time you granted it without the grant option, the user will retain the grant option.

[GRANTED BY <grantor>] Syntax

```
[GRANTED BY CURRENT_USER | user_name]
```

Usage Notes

- Use this option to indicate whether you want the grant to be from the CURRENT_USER or the CURRENT_ROLE, or whether you want to revoke authorization records that were granted from the CURRENT_USER or from the CURRENT_ROLE.
- If GRANTED BY <grantor> is not specified, then the grantor is the CURRENT_USER.
- If GRANTED BY CURRENT_ROLE is specified, then the CURRENT_ROLE must not be NULL.
- A <*grantor*> of *user_name* is not ANSI standard. Only the users, PBSYSADMIN, database owner, or someone with the PBDBA role can specify a <*grantor*> of *user_name*.

Examples

- The following statement grants the SELECT privilege on the CUSTOMER_TBL table to the user MARKETING_MGR.

```
GRANT SELECT
ON customer_tbl
TO marketing_mgr;
```

- The following GRANT statement allows the user FINANCIAL_MGR to delete, insert and update rows from the DISCOUNT_CODE_TBL table; it also allows this user to grant the same privileges to others.

```
GRANT DELETE,INSERT,UPDATE
ON discount_code_tbl
TO financial_mgr
WITH GRANT OPTION;
```

- The following GRANT statement allows the user HR_MGR to have ALL PRIVILEGES on the table SALES_REP_DATA_TBL. However, the user HR_MGR will only be granted privileges that the user granting the privileges has the right to grant. For example, if the user granting the privileges does not have the right to grant DELETE privileges, the HR_MGR will not have the delete privilege.

```
GRANT ALL PRIVILEGES
ON sales_rep_data_tbl
TO hr_mgr
```

REVOKE Syntax

```
REVOKE [ GRANT OPTION FOR ] <privilege_list>
ON <object>
FROM <user_name> [ RESTRICT | CASCADE ]
[GRANTED BY <grantor>]
```

To revoke a role from a user, use the SQL command, REVOKE. This command revokes *only* the privileges that the specified <*grantor*> granted to the <*user_name*>. If another <*grantor*> granted the same privileges to the <*user_name*>, then the <*user_name*> will still have those privileges.

Please note that the syntax rules for the REVOKE syntax is similar to the GRANT statement. The major difference is the additional RESTRICT or CASCADE keyword and the GRANT OPTION FOR clause. The following describes the optional clauses GRANT OPTION FOR and RESTRICT or CASCADE.

NOTE: You may only revoke privileges, which *you* have granted.

GRANT OPTION FOR

If the optional GRANT OPTION FOR clause is used, the WITH GRANT OPTION right is revoked, but the actual privilege itself is not revoked. CASCADE and RESTRICT may be used in the same way as a normal REVOKE statement.

RESTRICT | CASCADE

If you specify the RESTRICT keyword, only the privilege granted by you, will be revoked from the specified user. If the specified user had the grant option and granted the same privilege to other users, then PointBase will raise an error.

If you specify CASCADE, only the privilege granted by you, will be revoked from the specified user and any other privileges dependent on your grant.

If the optional RESTRICT or CASCADE keywords are not used, PointBase uses CASCADE by default.

Predefined Roles

This section describes predefined roles in PointBase. Predefined roles and roles in general can save you time granting commonly-used privileges to a user, a group of users, or another role. Predefined roles can provide you some type of authority over databases. Predefined roles and roles in general are multiple privileges bundled into one object. You can typically use a predefined role to apply commonly-used privileges to one user or a group of users or another role. For example, one predefined role gives specified users all the privileges that a database owner has. The other predefined role gives specified users read authority on all objects in the database. **You may not grant additional privileges to predefined roles.** PointBase provides the following predefined roles:

PBDBA Role

You have complete authority, including all privileges over the database using the PBDBA role. Please note that it cannot be granted to other roles.

READALL Role

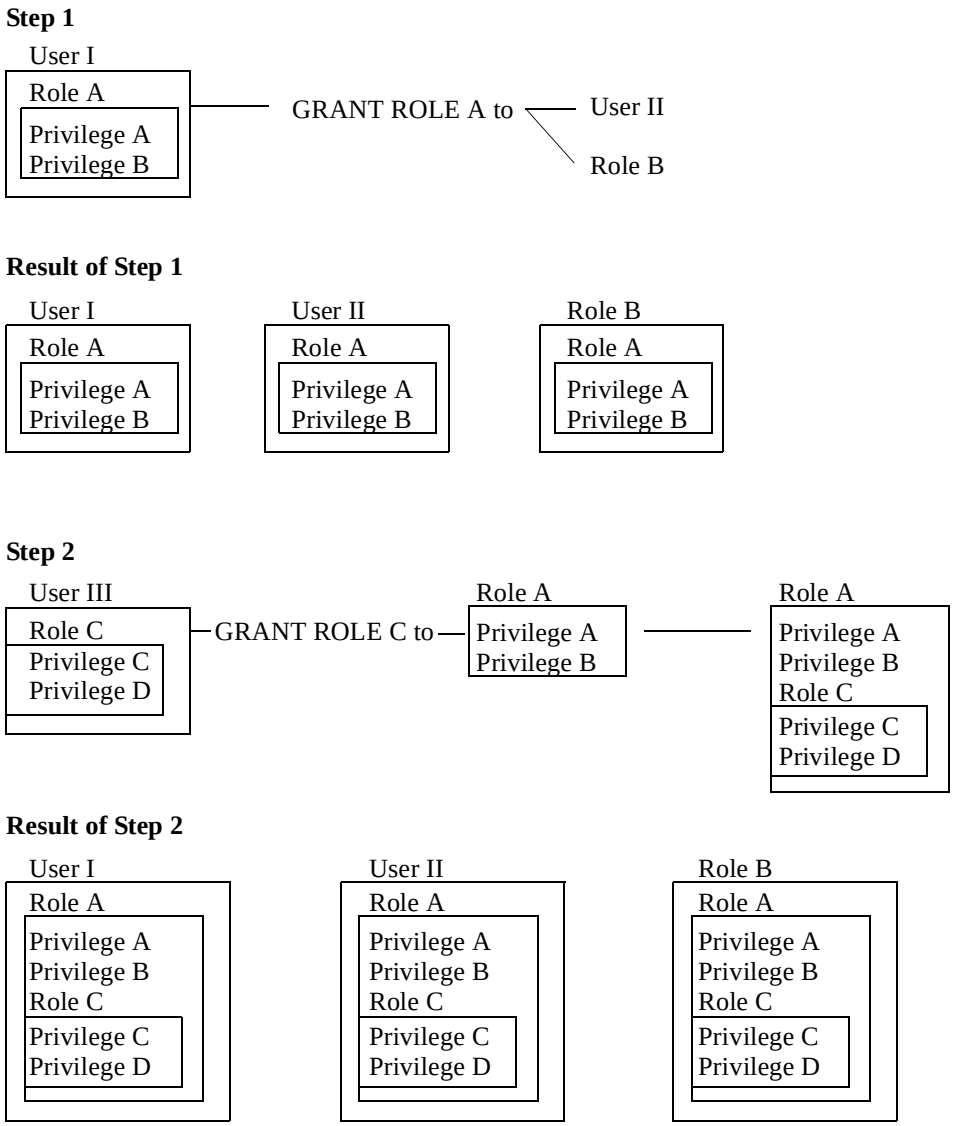
You can grant other users the read or SELECT authority on all objects in the database using the READALL role. With it, any user can unload the entire database—regardless of who owns the objects or what privileges have been granted on them.

Granting and Revoking Privileges to Roles

With PointBase, you have the capability to grant or revoke roles. They may contain multiple privileges, which you can apply towards multiple users, without having to apply each privilege one user at a time. Any user can grant roles to other users or to other roles if they have the authority. Any user with the authority may grant additional privileges to roles.

To enable your current role, you must use the SQL command, SET ROLE. PointBase allows you to enable or set your current role if your current user has been granted that role. A user may only have one enabled role—one current role, at any given time—though a user may have been granted several different roles. Please note that at any given time, users' total privileges are the sum of all privileges directly granted to them and any privileges or roles granted to their current role.

The following diagram briefly characterizes roles by illustrating User I granting Role A to User II and Role B. It also displays User III granting Role C to Role A and how User II and Role B are affected by this change.



CREATE ROLE Syntax

```
CREATE ROLE <role_name> [WITH ADMIN <grantor>]
```

To create a role that can have privileges granted to it, use the SQL command CREATE ROLE. The following explains the CREATE ROLE syntax.

<role_name>

It is the name of the role you are creating. For *<role_name>*, you may use any valid user name, except PUBLIC, NONE, or the same name as an existing user.

```
<grantor> = CURRENT_USER | CURRENT_ROLE | user_name
```

- If WITH ADMIN <*grantor*> is not specified, then the grantor is the CURRENT_USER.
- IF WITH ADMIN CURRENT_ROLE is specified, then the CURRENT_ROLE must not be NULL.
- A <*grantor*> of *user_name* is not ANSI standard. Only the PBSYSADMIN, database owner, or someone in the PBDBA role can specify a <*grantor*> of *user_name*.

Examples

```
<need examples>
```

GRANT ROLE Syntax

```
GRANT <role_name> [ { , <role_name> } …]
TO <grantee> [{ , <grantee>} … ]
[WITH ADMIN OPTION]
[GRANTED BY <grantor>]
```

To grant users a role, use the SQL command, GRANT ROLE. The following explains its syntax.

```
<role_name>
```

It is the name of the role you are granting. You may grant more than one role.

```
<grantee> = PUBLIC | <role_name>
```

- A role can be granted to users or other roles.
- You cannot grant a role to itself.
- You cannot grant one role to a second role, and then attempt to grant the second role back to the first. For example, you can grant Role (A) to Role (B) or Role (B) to Role (A), but not both. Such a series of grants would result in a role grant cycle, which is not allowed.
- Granting to PUBLIC grants the role to **all** present and future roles.

```
[WITH ADMIN OPTION]
```

If WITH ADMIN OPTION is specified, then the <*grantee*> can grant the role to other users or roles. It also gives the <*grantee*> the right to drop the role.

```
<grantor> = CURRENT_USER | CURRENT_ROLE | user_name
```

- If you do not specify GRANTED BY <*grantor*>, then the grantor is the CURRENT_USER.
- If you specify GRANTED BY CURRENT_ROLE, then the current role must not be NULL.
- To successfully execute this command, current users must either be the PBSYSADMIN or the database owner. Or, current users must either have the PBDBA role, or the <*grantor*>*s* must have admin option for every role that they grant.
- A <*grantor*> of *user_name* is not ANSI standard. Only the PBSYSADMIN, database owner, or someone in the PBDBA role can specify a <*grantor*> of *user name*.

REVOKE Syntax

```
REVOKE [ADMIN OPTION FOR] <role_name> [ { , <role_name> } …]
FROM <grantee> [{ , <grantee>} … ]
[GRANTED BY <grantor>]
<drop_behavior>
```

To revoke a role from a user or another role, use the SQL command, REVOKE. This command revokes *only* the roles that the specified <*grantor*> granted to the <*grantee*>. If another <*grantor*> granted the same role the <*grantee*>, then the <*grantee*> will still have privileges to that role.

Please note that the syntax rules for the REVOKE syntax is similar to GRANT ROLE, except for the following.

NOTE: You may only revoke roles, which *you* have granted.

```
[ADMIN OPTION FOR]
```

If ADMIN OPTION FOR is specified, then only the admin option for the role is revoked.

```
<drop_behavior> = CASCADE | RESTRICT
```

- If you specify the RESTRICT keyword, only the role granted by you, will be revoked from the specified <*grantee*>. If the specified <*grantee*> had the ADMIN OPTION and granted the same privilege to other users, they will retain the privilege.
- If you specify CASCADE, only the role granted by you, will be revoked from the specified <*grantee*> and any other roles dependent on your grant.
- If the optional RESTRICT or CASCADE keywords are not used, PointBase uses CASCADE by default.

DROP ROLE Syntax

```
DROP ROLE <role_name> [<drop_behavior>]
```

To successfully execute this command, the current user must be the PBSYSADMIN or the database owner, or the current role must be PBDBA. If your current user or role has been granted admin option on the role being dropped, you may also use this command.

```
<drop_behavior> = CASCADE | RESTRICT
```

- If the drop behavior is CASCADE, then all schemas owned by this role will be dropped. Also, all privilege entries in the catalog tables where this role is the <*grantor*>, the <*grantee*>, or the object being granted will be dropped.
- If the drop behavior is RESTRICT, then an error will be raised if there are any schemas owned by this role or if there are any privilege entries, where this role is the <*grantor*>, the <*grantee*>, or the object being granted.
- If drop behavior is not specified, then CASCADE is the default.
- You cannot drop the predefined roles: PBDBA and READALL.

SET ROLE Syntax

```
SET ROLE <role_name> | NONE
```

Usage Notes

- To successfully execute this command, the current user must be the PBSYSADMIN, the database owner, or a user granted to use this role. Or, the current role must be PBDBA.
- This statement will set the current role for the current user to either the role specified or to the null value if NONE is specified.
- If this statement is executed and an SQL transaction is currently active, then an error will be raised: dbexcpITSActiveSQLX : "Invalid transaction state - active SQL-transaction".

Optimizer Usage in PointBase

Query optimization consists of analyzing an SQL query expression used within SELECT, DELETE, (where clause), INSERT (query expression), or UPDATE (where clause) statements to determine the best way to execute the query expression. The description of which resources to use and how to use them is known as a query plan.

Some of the major influences on the optimizers choice of a query plan are the following:

- The SQL table(s) in the database to be accessed.
- The indexes on the specified SQL table or tables and how they relate to the query expression.
- The size of each SQL table (number of rows, columns) which impacts the number of reads from physical storage.
- The size of the cache that can contain the rows of the SQL table or tables.

Indexes are very crucial to the optimizer. If a column or columns of the index are used in the where clause or as a joining column, then the optimizer can consider using the index in the costing of a given query plan. The optimizer measures the cost of using any index that exists. Indexes allow the database system to only access those rows that meet the criteria of the query expression. For an index, the optimizer determines the number of leaf pages (bottom tier), the depth of the index and the selectivity factor. The selectivity factor is a ratio of finding a given index key value within the index.

Additionally, the optimizer determines the cost of scanning each row of the table.

The optimizer in the PointBase RDMS is a cost-based optimizer. This means that it will look at the numerous possible query plans and determines which one is most optimal and least expensive for a given query expression.

One of the difficulties that an optimizer encounters is that the index information can become quickly outdated. This occurs through deletions, insertions, and modifications to the actual key values of the index. The number of leaf pages, the depth and the selectivity factor all become invalid with just one or more deletions, insertions, or modifications. To solve this problem, most database systems require a database administrator to run a statistics tool to update the information on an index so that the optimizer always has current information.

The optimizer in the PointBase RDMS does not require any such tool or the intervention of a database administrator. The PointBase system automatically keeps the number of leaf pages, index depth, and selectivity factor for each index consistently current. By keeping pertinent information current, the optimizer will always have correct information to determine what query plan should be employed to execute the query expression.

Execution Plan

Whenever a query is compiled, the optimizer figures out various ways the query can be executed and picks the one with the lowest cost. The cost is determined in terms of the number of I/Os needed to perform the query in addition to the CPU cost associated with evaluating the portion of the query under consideration. The important elements of an execution plan are:

- The access methods: if there are any indexes, are they used? If there are multiple indexes, which ones are used? If an index is used, does the base table need to be accessed, or is all the information needed available in the index? If only the index is accessed and not the base table, this is known as an index-only access and can improve performance dramatically.
- The join order: this determines the order in which tables are accessed. At each step, the execution plan estimates costs and the number of rows produced.
- The join type: this element could be a NESTED LOOP or an OUTER JOIN NESTED LOOP.
- The predicates: where in the execution plan are the predicates put to use? The objective is to push the computation as close to the data and filter it out as quickly as possible.

The PointBase PLAN facility provides you with all of these elements. This information is kept in two SQL tables: PLAN_QUERIES and PLAN_TABLE. These SQL tables can be accessed and modified with common SQL commands. These tables have the following characteristics:

- The system automatically creates these tables.
- The owner of each table is the current user.
- As the user, you must truncate the tables to remove data that it no longer needs. To remove data from these tables, use a DELETE statement.

NOTE: For a detailed description of these tables, please refer to the *Database Log Flushing* section of your *PointBase™ System Guide*.

Commands for PointBase Commander

To generate the execution plan, you can do the following:

- From PointBase Commander, use the SET PLANONLY ON command to retrieve the plan.
- Compile and execute the SET PLANONLY ON command. Next, compile the query of interest. Finally, you should set PLANONLY to OFF. Once this is done, you can view the PLAN_TABLE and PLAN_QUERIES tables with a SELECT statement.

SET TIMING ON | OFF

This statement returns the time for the compile step in addition to the total time taken for any command.

SET PLANONLY ON | OFF

Once PLANONLY is turned on, plans will be generated for all SQL queries until PLANONLY is turned off.

SET SHOWPLAN ON | OFF

This statement displays the plans generated by all SQL queries. For example, plans for the following queries are shown below for these statements:

```
select * from t1,t2 where t1.c1 = t2.c1 and t1.c1>5 and t2.c1<100;
select * from t1 LEFT OUTER JOIN t2 on t1.c1=t2.c1;
select max(t1.c1) from t1,t2 group by t2.c1, t1.c1;
```

The PLAN_QUERIES table would be:

Query	***Value***
1	`select * from t1,t2 where t1.c1 = t2.c1 and t1.c1>5 and t2.c1<100`
2	`select * from t1 LEFT OUTER JOIN t2 on t1.c1=t2.c1`
3	`select max(t1.c1) from t1,t2 group by t2.c1, t1.c1`

The PLAN_TABLE table would return:

Query	*Block*	*Step*	*Join*	*Access*	*TableId*	*IndexId*	*Cost*	*OutputRows*	*Expression*
1	1	1	none	table scan	190	Null	1	0	<(T2.C1,constant)
1	1	2	nested loop join	table scan	189	Null	0	0	=(T1.C1,T2.C1), >(T1.C1,constant)
2	1	1	none	table scan	189	Null	1	20	Null
2	1	2	outer nested loop join	table scan	190	Null	20	16	=(T1.C1,T2.C1)
3	1	1	group by	Null	Null	Null	Null	Null	(T2.C1,T1.C1)
3	1	2	none	table scan	189	Null	1	4	Null
3	1	3	nested loop join	table scan	190	Null	4	80	Null

Application Programming Interface Tools

This chapter describes what application programming interface (API) tools PointBase offers and how to use them. Unlike other PointBase tools, for example, Commander and Console, you can integrate the API tools explained in this chapter with a Java application. This chapter will divide each API tool or combination of tools into sections, beginning with the main purpose for using the tool(s), followed by a description of the Java classes and other components, accompanied with a brief summary of how the different parts can work together (if needed), and finally, ending with examples of how to implement the tool(s). After reading or browsing this chapter, you may find a useful tool(s) that an application can integrate.

Load and Unload API's

PointBase provides tools that you can use to either load or unload a database, or unload a table using the load and unload API's. Using it, you can write your application once and call methods to unload or load a database without having to write anything on a command line. However, you can also create a stand-alone tool or a command-line tool using the load and unload API's. Either way you choose, PointBase gives you the needed tools to load or unload a database, or unload a table.

Unload API

To unload a database or table using the unload API, you must use the PointBase class, "com.pointbase.tools.toolsUnload." It contains two static methods, "unloadDatabase(Connection p_conn, String p_filename, boolean p_preserve)" and "unloadTable(Connection p_conn, String p_filename, String p_tableName)."

unloadDatabase(Connection p_con, String p_filename, boolean p_preserve)

To unload a complete database into directory as a specific .sql file, you must use the static method, "unloadDatabase(Connection p_conn, String p_filename, boolean p_preserve)." You need to create the connection and then pass the connection reference to the API. You also need to provide the file name with the complete path; if you do not provide it, the API will unload the database into a .sql file located in the directory, where you launched the application.

The third parameter preserves ownership when unloading. TRUE preserves the ownership of schemas, grantors in GRANT statements, and create ROLE owners. But, it does not preserve the DATABASE OWNER. Whoever creates the new database becomes the database owner. See the example after the unload table method.

unloadTable(Connection p_conn, String p_filename, String p_tableName)

To unload an entire table into a specific .sql file and directory, you must use the static method, "unloadTable(Connection p_conn, String p_filename, String p_tableName)." You need to create the connection and then pass the connection reference to the API. You also need to provide the file name with the complete path; if you do not provide it, the API will unload the table into a .sql file located in the directory where you launched the application. If you unload a table, you must provide the complete-qualified name of the table; that is, "<schema_name>.<tableName>"; if you do not provide it, the API will search for the table name in the current schema path. *For mixed-case-table names, the example describes the supported syntax.*

```
import  com.pointbase.tools.toolsUnload  ;
    public  class  test
    {
        Connection  m_con;
        public  test()  throws  SQLException
        {
            Class.forName("com.pointbase.jdbc.jdbcUniversalDriver");
m_con  =  DriverManager.getConnection("jdbc:pointbase:embedded:sample",  "pbpublic",  "p
ublic");
        }
        public  void  unloadDatabase()
        {
            toolsUnload.database(  m_con,  "e:\\pointbase\\database.sql", true);
            toolsUnload.table(  m_con,  "e:\\pointbase\\table.sql"  "public.t1");
            //table  names are case-sensitive, see the following:
            toolsUnload.table(  m_con,  "e:\\pointbase\\table1.sql"  "public.ajay");

        }
        public  static  void  main(  String[]  args)
        {

            try
{
                test  t  =  new  test();
                test.loadDatabase();
}
catch(SQLException  ex){}
        }
    }
```

Stand-Alone or Command Line Tool

To use the unload tool on the command line, you can use the following example, which unloads a **complete database** into the file, "database.sql" in the directory, "e:\." It also preserves the ownership of schemas, grantors in GRANT statements, and create ROLE owners. But, it does not preserve the DATABASE OWNER. Whoever creates the new database becomes the database owner. You must provide the file name with the complete path; if you do not provide it, the API will unload the table into a .sql file located in the directory where you launched the application. If you unload a table, you must provide the complete-qualified name of the table; that is, "<schema_name>.<tableName>"; if you do not provide it, the API will search for the table name in the current schema path. *For mixed-case-table names, the example describes the supported syntax.* It uses the following default options:

- -driver com.pointbase.jdbc.jdbcUniversalDriver
- -url jdbc:pointbase:embedded:sample
- -user PBPUBLIC

- -password PBPUBLIC

```
java  com.pointbase.tools.toolsUnload
–driver  com.pointbase.jdbc.jdbcUniversalDriver –url jdbc:pointbase:embedded:sample –
file e:\database.sql -preserve true –user pbpublic –password pbpublic -table null
```

To unload a **table**, you can refer to the following example:

```
java  com.pointbase.tools.toolsUnload
–driver  com.pointbase.jdbc.jdbcUniversalDriver –url jdbc:pointbase:embedded:sample –
file e:\table.sql –user pbpublic –password pbpublic -table pbpublic.table1
```

Load API

To load a database using the load API, you must use the PointBase class, “com.pointbase.tools.toolsLoad.” It contains one static method, “load(Connection p_conn, String p_filename).”

load(Connection p_conn, String p_filename)

Using this method, you must first create the connection and then pass the connection reference to the API. You must also provide the file name with the complete path; if you do not provide it, the API will try to load the file from the current location of the application. The following example describes the connection, “m_con” and the complete path and file name, “e:\pointbase\database.sql.”

```
import  com.pointbase.tools.toolsLoad  ;
    public  class  test
    {
        Connection  m_con;
        public  test()  throws  SQLException
        {
            Class.forName(“com.pointbase.jdbc.jdbcUniversalDriver”);
m_con  =  DriverManager.getConnection(“jdbc:pointbase:embedded:sample”,  “pbpublic”,  “p
bpublic”);
        }
        public  void  loadDatabase()
        {
            toolsLoad.load(  m_con,  “e:\pointbase\database.sql”);
        }
        public  static  void  main(  String[]  args)
        {
            try
{
                test  t  =  new  test();
                test.loadDatabase();
}
catch(SQLException  ex){}

        }
    }
```

Stand-Alone or Command Line Tool

To use the load tool on the command line, you can use the following example, which loads a **complete database** into the file, “database.sql” in the directory, “e:\.” You must provide the file name with the complete path; if you do not provide it, the API will try to load the file from the current location of the application. It uses the following default options:

- -driver com.pointbase.jdbc.jdbcUniversalDriver
- -url jdbc:pointbase:embedded:sample
- -user PBPUBLIC
- -password PBPUBLIC

```
java com.pointbase.tools.toolsLoad –driver com.pointbase.jdbc.jdbcUniversalDriver –
url jdbc:pointbase:embedded:sample –file e:\database.sql –user pbpublic –
password pbpublic –log true
```

Appendix A: SQL Reference

Conventions

This section describes documentation conventions. There are two basic conventions:

1. *Page format* conventions provide a structure for the organization of individual pages in the documentation.
2. *Syntax* conventions convey specific information about keywords and clauses in the SQL statements described in this document.

Page Format Conventions

Each SQL statement in the data manipulation language, data definition language, and transaction control sections of the PointBase SQL documentation uses a specific page format.

- Each statement page starts with the primary keyword of the statement, which displays at the heading of the page; for example, SELECT.
- The statement keyword(s) is followed by the syntax of the statement. The statement syntax follows the conventions described in “Syntax Conventions,” below.
- Immediately following the statement syntax is a brief description of the overall statement.
- Detailed explanations are then described for each keyword and clause in the statement. Some clauses may include a more detailed explanation of their own syntax or links to other documents that describe clauses that are common to more than one SQL statement.

Syntax Conventions

Each SQL statement uses certain types of capitalization, formatting, and punctuation that describe the attributes of different portions of the statement.

- If a portion of an SQL statement displays in **UPPERCASE**, the capitalized words are keywords, which are generally required in the SQL statement or clause. Keywords are not case sensitive, and they must be spelled exactly as they display in this document.
- Portions of SQL statements that display in ***lowercase italic*** are SQL values. SQL values used in PointBase SQL can be constants, column names, values formed from combinations of column values and constants, or the result of any function that returns a single value. The values for variables in conditional expressions are case sensitive.
- The clauses in an SQL statement that display between **[**brackets**]** are optional. If an optional clause has several components or keywords, they display within the brackets.
- Curly braces **{}** in SQL statements indicate that one or more clauses are used together.
- Ellipses are sets of periods (such as "..."). Ellipses in an SQL statement have the same meaning as "etc."; they denote that the series of keywords, clauses, or variables that precede the ellipses go on indefinitely.

Data Definition Language

The following section describes the syntax for creating and managing logical data objects. The Data Definition Language (DDL) is essential to creating a database. Use the following DDL statements and operations to begin building your PointBase database.

- "CREATE SCHEMA" on page 108
- "CREATE TABLE" on page 109
- "CREATE VIEW" on page 119
- "CREATE USER" on page 121
- "CREATE ROLE" on page 121
- "CREATE INDEX" on page 122
- "CREATE FUNCTION" on page 123
- "CREATE PROCEDURE" on page 126
- "CREATE TRIGGER" on page 128
- "ALTER USER" on page 134
- "ALTER TABLE" on page 133

CREATE SCHEMA

```
CREATE SCHEMA schema_name
    [ AUTHORIZATION user_name ]
    [COUNTRY country_code [LANGUAGE language_code]]
```

The CREATE SCHEMA statement creates a schema in a PointBase database.

Syntax

CREATE SCHEMA	The CREATE SCHEMA keyword is required as the first words in a CREATE SCHEMA statement.
schema_name[a]	The name of the schema.
user_name	The schema owner name or the role name. If you specify a role name, any user who enables the specified role can have full schema ownership privileges. The schema owner name or the role name must exist in the database or an error is raised. If *user_name* is not specified the current *user_name* is the owner of the schema.
country_code[b]	Specifies the country code. The default country code is US English. When this option is used, char data is stored as Unicode. If this option is NOT used, char, varchar, and CLOB columns use US ASCII values.
language_code	Specifies the language code. The default language code is US English. When this option is used, char data is stored as Unicode. If this option is NOT used, char, varchar, and CLOB columns use US ASCII values.

a. PointBase recommends to use the same name for both the schema_name and the user_name. Once you log in, PointBase creates new objects in the schema that has the same name as your user_name. If no schema has the same name as your user_name, PointBase creates the new objects in the PBPUBLIC schema.

b. Please refer to *Country and Language Codes* of the *PointBase System Guide* for a list of valid country codes and languages.

Examples

```
CREATE SCHEMA ORDERS
AUTHORIZATION Orders_Mgr
COUNTRY     FR
LANGUAGE    FR;
```

CREATE TABLE

```
CREATE TABLE table_name (column_definition | table_constraint_definition
[{, column_definition | table_constraint_definition}...]) [TABLE PAGESIZE size, LOB
PAGESIZE size]
[COUNTRY country_code [LANGUAGE language_code]]
```

The CREATE TABLE statement creates the table structures for the PointBase database. The CREATE TABLE statement allows you to define the table by name, to define the columns, default values, keys, and constraints on the table.

Syntax

CREATE TABLE	The CREATE TABLE keywords are required as the first words in a CREATE TABLE statement.
`table_name`	The *table_name* is the name of the table structure. The table name cannot be the same as a PointBase keyword. Table names in the PointBase database are not case sensitive and can be up to 128 characters long.
column_definition	The *column_definition* contains all the information needed to define the columns that are a part of a table. See the following pages for the section on *column_definition* syntax.
table_constraint_definition	The *table_constraint_definition* allows you to define a constraint that is applicable to the table. Usually this type of constraint is used when you specify multiple columns for any type of constraint. See the following pages for the section on *table_constraint_definitions*.
`TABLE PAGESIZE size`	Use the TABLE PAGESIZE keywords after all the column definitions and table constraint definitions to define the page size of the table. If this specification is omitted, the table uses the default pagesize as set in the database properties file (`pointbase.ini`). The table pagesize identifies the number of digits, KB, or MB reserved for the index. *Size* can be one of the following: • a number, such as 1024 • KiloBytes, such as 1K • MegaBytes, such as 1M

`LOB PAGESIZE` *`size`*	Use the LOB PAGESIZE keywords after all the column definitions and table constraint definitions to define the page size of the BLOB and CLOB columns. If this specification is omitted, the LOB uses the default pagesize. If both table and LOB pagesizes are specified, either the table or the LOB pagesize can be defined before the other. The LOB PAGESIZE identifies the number of digits, KB, or MB reserved for the CLOB or BLOB index. *Size* can be: • a number, such as 1024 • KiloBytes, such as 1K • MegaBytes, such as 1M It is required only if one or more columns in the table contain LOB characters. You should specify this only once, even if the table has multiple LOB columns. All LOBs will use pages of this size for storing LOBs, unless the LOB fits into the data page. If this specification is omitted, the LOB pages use the default page size.
COUNTRY *country_code*	Specifies the country code. The default country code is US English[a]. When this option is used, char data is stored as Unicode. If this option is NOT used, char, varchar, and CLOB columns use US ASCII values.
LANGUAGE *language_code*	Specifies the language code. The default language code is US English. When this option is used, char data is stored as Unicode. If this option is NOT used, char, varchar, and CLOB columns use US ASCII values.

a. Please refer to *Country and Language Codes* of the *PointBase™ System Guide* for a list of valid country codes and languages

A table has a given locale property if the following items are fulfilled:

- the country code or language code is explicitly specified in the CREATE TABLE statement.
- the country code or language code is explicitly specified in the CREATE SCHEMA statement.
- language and country settings are specified in the `pointbase.ini` file.

Column_Definition Syntax

```
column_name data_type [identity_property | default_clause] [column_constraint]
```

column_name	The *column_name* is the name of the column structure within the table created with the CREATE TABLE statement. The column name must be composed of alphanumeric characters or the equivalent in another language, for example, a word in Japanese characters and cannot be the same as a PointBase keyword. The column name must be unique within the table that contains it. Column names in the PointBase database are not case sensitive and can be up to 128 characters in length.
data_type	The data type describes the type of data that can be stored in the column.
identity_property	`IDENTITY [(start_value, increment_value )]` IDENTITY keyword is used to recognize the definition of IDENTITY property. *start_value* is the value of the first row in the table. The value must be a value greater than zero. If you do not specify this value, the default is 1 (one). *increment_value* is an incremental value based on the *start_value*. The value must be a value greater than zero. If you do not specify this value, the default is 1. The maximum value for either *start_value* or *increment_value* is equal to the maximum value possible for the data type. For example, The maximum value possible for NUMERIC (4,0) is 9999. You can have IDENTITY columns with exact numeric data types and a 0 (zero) scale only. The exact numeric data types include INTEGER, SMALLINT, NUMERIC, or DECIMAL. You cannot update IDENTITY columns nor can you specify NULL for them. Also, you can only have one IDENTITY column in a table. (See "IDENTITY Property for Autoincrement" on page 42.)

default_clause	The *default_clause* allows one to specify default values for a given column. Possible default values and an example are: • character string literal: ‘abc’ • numeric literal: 123 • datetime literal: time ‘22:45:21’ • binary string literal: X’104dc2’ • boolean literal: TRUE • NULL value • datetime value functions: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP • special registers • SQL routine The default value can be used with SQL Insert and Update statements. Specify either DEFAULT or DEFAULT VALUES or specify nothing at all and the default value is inserted.
column_constraint	The *column_constraint* is one or more keywords that restricts the data that can be written to a particular column. The PointBase database currently supports the following column constraints. All column constraints are optional. • `NOT NULL` • PRIMARY KEY • UNIQUE • FOREIGN KEY • CHECK

Column_Constraints

NOT NULL	The optional NOT NULL keyword indicates that a particular column must have a non-NULL value associated with it. If one performs any action to a table that results in a NOT NULL column having a NULL value, the PointBase database returns a runtime error. The syntax for the NOT NULL column constraint is: `NOT NULL`
PRIMARY KEY	The optional PRIMARY KEY keyword creates an index for a column. The syntax for the PRIMARY KEY column constraint is: `PRIMARY KEY` The PRIMARY KEY column constraint can specify only one column. To specify a PRIMARY KEY constraint with multiple columns, use a *table_constraint*.
UNIQUE	The optional UNIQUE constraint defines a unique key on the column. All values for this column must be unique. The syntax for the UNIQUE column constraint is: `UNIQUE` The UNIQUE column constraint can specify only one column. To specify a UNIQUE constraint with multiple columns, use a *table_constraint*.

FOREIGN KEY	The optional FOREIGN KEY keyword indicates that a relationship exists between the column value of this table (known as the child table) and the primary key of the parent table referenced in the REFERENCES clause. The syntax for the FOREIGN KEY constraint is:
	`FOREIGN KEY`
	`REFERENCES table_name (column_name )`
	`[ON DELETE {NO ACTION | RESTRICT | CASCADE | SET DEFAULT | SET NULL}]`
	`[ON UPDATE {NO ACTION | RESTRICT |CASCADE | SET DEFAULT | SET NULL}]`
	The ON DELETE clause defines the rules for deleting specific columns on the specified table. To do this, specify either:
	NO ACTION, CASCADE, RESTRICT, SET DEFAULT or SET NULL.
	You must specify at least one identifier. NO ACTION omits the ON DELETE clause. RESTRICT looks to see what objects are dependent on the object being dropped and if there are dependent objects, then the dropping of the object does not occur. CASCADE has the effect of dropping all SQL objects that are dependent on that object. SET DEFAULT assigns default values to all components of the target column. SET NULL assigns null values to all components of the target column.

	The ON UPDATE clause defines the rules for updating specific columns on the specified table. To do this, specify either: NO ACTION, RESTRICT, CASCADE, SET DEFAULT or SET NULL. If the ON DELETE or ON UPDATE clauses are omitted, the default is NO ACTION. FOREIGN KEY REFERENCES are required keywords, *table_name* is the name of a table that already exists in the PointBase database, and the *column_name*s are the names of the columns that define the primary key of the referenced table. This column and the column in the referenced table must have exactly the same data type. The referenced table must have a unique or primary index on the specified column. A foreign key relationship means that any values written to a column with an INSERT or UPDATE statement must already exist as a value in the primary key of the referenced table and columns.
CHECK	The optional CHECK keyword indicates that the value of a column to be inserted or updated must meet the criteria of the check constraint. The syntax for the CHECK constraint is: `CHECK ( search_condition )` where the search_condition follows the rules of search conditions.

Table_Constraint_Definition

The *table_constraint_definition* allows you to define a constraint that is applicable to the table. Usually this type of constraint is used when you specify multiple columns for any type of constraint. There can only be a single *column_constraint* per column. The *table_constraint_definition* uses the syntax of:

```
[CONSTRAINT constraint_name]
       {unique_constraint | referential_constraint | check_constraint}
```

constraint_name	The name that one supplied to identify a constraint. Each *constraint_name* must be unique for a table. The *constraint_name* is optional but if two constraints have the same definition, then they will each need a name for uniqueness.
unique_constraint	The *unique_constraint* defines an explicitly named primary key or unique constraint of one or more columns. The syntax for the *unique_constraint* is: `{UNIQUE | PRIMARY KEY} (column_name [{, column_name}...])`

referential_constraint	The *referential_constraint* defines an explicitly named foreign key constraint of one or more columns.
	
	The syntax for the *referential_constraint* is:
	
	`FOREIGN KEY (column_name [{, column_name}...])`
	`REFERENCES table_name [{column_name,`
	`column_name,...}]`
	`[ON DELETE {NO ACTION | CASCADE | RESTRICT | SET`
	`DEFAULT | SET NULL}]`
	`[ON UPDATE {NO ACTION | CASCADE | RESTRICT | SET`
	`DEFAULT | SET NULL}]`
	`[INDEX PAGESIZE <size>]`
	
	A given foreign key and its matching candidate key must contain the same number of columns, *N*, such as: the *I*th column of the foreign key corresponds to the *I*th column of the matching key (*I* = 1 to *N*), and corresponding columns must have exactly the same data type.
	
	The referenced table must have a unique or primary index on the specified columns. **Not allowed on a view**. PointBase raises an error if you attempt this on a view.
	
	If the column_name for the referenced table is omitted, it defaults to the columns in the primary key of the referenced table.

ON DELETE	The ON DELETE clause defines the rules of behavior when an attempt is made to delete a row in the parent table that has a corresponding row in the referencing table that is dependent on the row in the parent table. The dependency is based on the columns of the FOREIGN KEY in the parent table and corresponding columns in the referencing table. The purpose of this clause is to avoid dangling references. If the behavior rule is CASCADE, then all dependent or matching rows in the referencing table are deleted when the row in the parent table is deleted. If the behavior rule is NO ACTION, then if an attempt is made to delete a row in the parent table that has a dependent row in the referencing table, the row in the parent table will not be deleted. If the behavior rule is RESTRICT, then if an attempt is made to delete a row in the parent table that has a dependent row in the referencing table, the row in the parent table will not be deleted. The database checks before attempting to delete the row in the parent table. If the behavior rule is SET DEFAULT, then the columns of the rows in the referencing table are set to default values for their respective columns when the row in the parent table is deleted. Each column of the referencing table that corresponds to the FOREIGN KEY in the parent table must have a default value or an exception will be raised. If the behavior rule is SET NULL, then the columns of the rows in the referencing table are set to the SQL NULL value for their respective columns when the row in the parent table is deleted. Each column of the referencing table that corresponds to the Foreign Key in the parent table must allow SQL NULL values or an exception will be raised.

ON UPDATE	The ON UPDATE clauses defines the rules of behavior when an attempt is made to update the FOREIGN KEY columns in the parent table that has a corresponding row(s) in the referencing table that is dependent on the values of the FOREIGN KEY columns in the parent table. If the behavior rule is CASCADE, then all dependent or matching columns of rows in the referencing table are updated with the new values in the FOREIGN KEY columns of the parent table row. If the behavior rule is NO ACTION, then if an attempt is made to update columns of the FOREIGN KEY in the parent table and there are columns of rows in the referencing table that are dependent on the pre-updated values, then the update of the FOREIGN KEY columns in the parent table do not occur. If the behavior rule is RESTRICT, then if an attempt is made to update columns of the FOREIGN KEY in the parent table and there are columns of rows in the referencing table that are dependent on the pre-updated values, then the update of the FOREIGN KEY columns in the parent table does not occur. The database checks before attempting to update the row in the parent table. If the behavior rule is SET DEFAULT, then all dependent or matching columns of rows in the referencing table are updated with the default values of the referencing table. Each column of the referencing table that corresponds to the FOREIGN KEY in the parent table must have a default value or an exception will be raised. If the behavior rule is SET NULL, then the columns of the rows in the referencing table are set to the SQL NULL value for their respective columns when the row in the parent table is updated. Each column of the referencing table that corresponds to the Foreign Key in the parent table must allow SQL NULL values or an exception will be raised.
check_constraint	The *check_constraint* defines an explicitly named check constraint of one or more columns. The syntax for the *check_constraint* is: `CHECK ( column_name search_condition )`

NOTE: Creating a table with the CREATE TABLE statement creates the table structures, but does not add any data to the table. An INSERT statement for a table, or a LOAD via an IMPORT statement in PointBase Console, or a RUN in PointBase Commander, must follow the creation of the table.

Example 1

```
CREATE TABLE ORDER_TBL
    (ORDER_NUM           INT,
    CUSTOMER_NUM         INT,
    REP_NUM              INT,
    PRODUCT_NUM          INT,
    SALES_TAX_ST_CD      CHAR (2),
    QUANTITY             SMALLINT,
    SHIPPING_COST        DECIMAL(12,2),
    SALES_DATE           DATE,
    DELIVERY_DATETIME    TIMESTAMP,
    FREIGHT_COMPANY      VARCHAR (30))
    COUNTRY              FR
    LANGUAGE             FR;
```

Example 2

This creates a table with a 5k page size:

```
CREATE TABLE TM5 (C1 INT PRIMARY KEY) TABLE PAGESIZE 5K;
```

This creates a table with a default page size, but the primary key constraint specifies a page size of 2K for the index:

```
CREATE TABLE TM (C1 INT NOT NULL, C2 CHAR (10),
CONSTRAINT PKCONSTRAINT PRIMARY KEY (C1)
INDEX PAGESIZE 2K);
```

In this example, each index has a different pagesize:

```
CREATE TABLE TMF (C1 INT, C2 CHAR (10), C3 INT NOT NULL,
CONSTRAINT PK_TMF PRIMARY KEY (C3)INDEX PAGESIZE 5K
CONSTRAINT FK_TMF FOREIGN KEY (C1) REFERENCES TM (C1) INDEX PAGESIZE 3K );
```

In this example, all LOBs in the table have pagesize and the LOBs automatically create 5K pagesize file for the LOB index:

```
CREATE TABLE TMBLOB (C1 INT NOT NULL, C2 BLOB (10K), C3 BLOB (5K))
LOB PAGESIZE 5K;
CREATE INDEX TMIX ON TMBLOB (C1) INDEX PAGESIZE 6K;
```

Example 3

This creates a table with a column having the IDENTITY property. This column will have the ability to autoincrement the values for each row.

```
CREATE TABLE TAB1(ID INT IDENTITY, NAME VARCHAR(30));
```

CREATE VIEW

```
CREATE VIEW <view name> [ ( view_column_list )]
           AS query_expression
       [ WITH [ levels_clause ] CHECK OPTION ]
```

The CREATE VIEW statement creates a view or derived table in the PointBase database.

Notes

- To create a view, you must own the schema, in which you are creating the view.
- You must have SELECT permission on all referenced columns of all referenced tables in the query expression.
- You can have "nested views," which are views that reference other views.
- To grant privileges on a view, you must have SELECT grant privileges on any referenced table or column in that view.

Syntax

CREATE VIEW	The CREATE VIEW keywords are required as the first words in a CREATE VIEW statement
view_name	The name of the view. The name is not case sensitive and can be up to 128 characters long.
view_column_list	Specify a view column list if the query expression includes two columns with the same name. The view column list and the query expression must specify the same amount of column names. If no view column list is specified, then the view column names are derived from the query expression (select column list).
query_expression	This is a SELECT statement. If the query expression does not include a column, it must have an AS clause correlation name defined. If it includes a column, the view column name is the column name without any table correlation name. The query expression is not allowed to contain any parameters and is limited to 3958 characters.

WITH CHECK OPTION	This option uses the WHERE clause in the view's query_expression like a table constraint: all resultant rows from an INSERT or UPDATE on the view must satisfy the WHERE clause. If no levels_clause is specified, CASCADED is implicit. **However, PointBase currently does not support Updateable Views.** That is, PointBase supports the syntax for WITH CHECK OPTION, but currently not the semantics.
levels_clause	CASCADED indicates that all resultant rows from an INSERT or UPDATE on the view must satisfy the its own WHERE clause and the WHERE clause of any views that are referenced. LOCAL indicates that all resultant rows from an INSERT or UPDATE on the view must only satisfy its own WHERE clause. If no levels_clause is specified, CASCADED is implicit.

Examples

```
CREATE VIEW customer_order
AS select order_num,order_tbl.customer_num,customer_tbl.name
FROM order_tbl,customer_tbl
WHERE product_num = 10;

CREATE VIEW customer_order1
AS select order_num,order_tbl.customer_num
FROM order_tbl,customer_tbl
WHERE order_tbl.customer_num = customer_tbl.customer_num;

CREATE VIEW customer_order2
AS select order_num,order_tbl.customer_num
FROM order_tbl,customer_tbl
WHERE order_tbl.customer_num = customer_tbl.customer_num and product_num=10;

CREATE VIEW namereps
AS select first_name,last_name
FROM sales_rep_tbl
WHERE last_yr_sales in (4000,6000,10000);

CREATE VIEW orde_by_rep (who,how_many,total,low,high,average)
AS select rep_num,count(*), sum(quantity),min(quantity),max(quantity),avg(quantity)
FROM order_tbl group by rep_num;

CREATE VIEW customer_order3
AS select order_num,first_name
FROM customer_order,namereps;

CREATE VIEW exceed_quotas
AS select office_num, sum(quota) as sum_quota, sum(ytd_sales) as sum_ytd
FROM sales_rep_tbl
GROUP BY office_num
HAVING sum(ytd_sales) > sum(quota);
```

CREATE USER

```
CREATE USER user_name PASSWORD password
[DEFAULT ROLE role_specification]
```

The CREATE USER statement creates a user in a given PointBase database and can assign a default role to that user. To successfully execute this command, the current user must be the PBSYSADMIN or the database owner. [See "Predefined Users" on page 89.] Or, the current role must be PBDBA. [See "Predefined Roles" on page 93.]

Syntax

CREATE USER	The CREATE USER keyword are required as the first words in a CREATE USER statement.
user_name	The name of the new user. **You cannot use keyword PUBLIC or an existing role name for the user name.**
password	The password associated with the user.
role_specification	The default *role_specification* is NONE.

Example

```
CREATE USER ENGINEERING_MGR PASSWORD ABCD;
```

CREATE ROLE

PointBase supports this statement. Please refer to the section, "CREATE ROLE Syntax" on page 96.

CREATE INDEX

```
CREATE [UNIQUE] INDEX index_name
ON table_name
column_name [sort_order] {, column_name [sort_order]...})
[INDEX PAGESIZE size]
```

The CREATE INDEX statement creates the index structures.

Syntax

CREATE INDEX	The CREATE INDEX keywords are required as the first words in a CREATE INDEX statement.
[UNIQUE]	If the UNIQUE keyword is specified, then the index will be defined as a unique index where duplicate values of the keys are not allowed.
index_name	The *index_name* is the name of the index. Compose the index name of alphanumeric characters or the equivalent in another language, for example, a word in Japanese characters, which are not the same as a PointBase keyword, unless the name is a delimited identifier. The index name must be unique for its table. Index names in the PointBase database are not case sensitive and can be up to 128 characters in length.
ON	Use the ON keyword between the *index_name* and the *table_name*.
table_name	The *table_name* refers to a table in the PointBase database. The *table_name* must refer to a table that has already been created at the time the CREATE INDEX statement executes. **Not allowed on a view.** PointBase raises an error if you attempt to use a view.
column_name	The *column_name* identifies a column in the table named in the *table_name* of the CREATE TABLE statement. There can be any number of columns. Total maximum length of all columns in an index must not exceed the pagesize.
[sort-order]	This optional clause specifies the sorting order of the column or columns in the index. The acceptable values for the ordering keyword are ASC or ASCENDING for columns that sort from the lowest value to the highest value in the column, and DESC or DESCENDING for columns that sort from the highest value to the lowest value in the column. Each *column-name* can only have one ordering keyword. If you do not include an ordering keyword, the sort order is ASCENDING.

`INDEX PAGESIZE`	Use the INDEX PAGESIZE keywords between the *sort_order* and the *size*.
size	The index size identifies the number of digits, KB, or MB reserved for the index. Size can be: • a number, such as 1024 • KiloBytes, such as 1K The index page size should be less than or equal to 32KB and the minimum is 1 KB. The default pagesize is 4KB unless a specific size has been set in the `pointbase.ini` file.

Example1

This creates an index:

```
CREATE INDEX ORDER_IND
     ON ORDER_TBL (ORDER_NUM DESC, CUSTOMER_NUM ASC);
```

Example2

This creates an index with a pagesize of 2K:

```
CREATE UNIQUE INDEX INDEX1
  ON SALES_TBL
 (CUSTOMER_NUM, SALES_DATE, PRODUCT_NUM)
 INDEX PAGESIZE 2K;
```

CREATE FUNCTION

```
CREATE FUNCTION function_name([parameter_definition [{,parameter_definition}...]])
       RETURNS return_clause
       LANGUAGE JAVA
       SPECIFIC specific_name
       sql_data_access
       EXTERNAL NAME external_function
       PARAMETER STYLE SQL
```

Using a stored function, you can automatically convert data to be stored in a PointBase database, without ever seeing the underlying conversion.

Syntax

Function_name	*Function_name* defines a stored function in a schema. The following are usage rules. • Including a schema name is optional. The following syntax is for *function_name*: `[schema_name.]function_name` • It must be composed of alphanumeric characters or the equivalent in another language, for example, Japanese characters. • It has a maximum limit of 128 characters long. • It is not case sensitive. • It cannot have the same name as a PointBase keyword. • It must be unique in the schema specified.
Parameter_definition	The *parameter_definition* specifies the *parameter_mode*, *parameter_name*, and *parameter_data_type*. (The *parameter_name* is optional.) The *parameter_mode* must be the value, IN. The *parameter_data_type* must be one of the PointBase data types. Separate multiple *parameter_definitions* with a comma. The following syntax is for *parameter_definitions.* `IN [parameter_name] PointBase_data_type`
RETURNS return_clause	This clause specifies the return data type in a stored function. The *data_type* must be one of the PointBase data types. The *return_clause* allows the following values: <PointBase data types> or TABLE (*pointbase_data_type* [{,*pointbase_data_type*}...]) **NOTE:** RETURNS return_clause is used with stored functions only. Stored procedures do not use it.
LANGUAGE JAVA	The clause specifies the language that the stored function uses to call the external function. It can take the following value: JAVA.
SPECIFIC specific_name	The SPECIFIC *specific_name* clause specifies a name that you can use instead of the *function_name* when invoking a stored function. The *specific_name* must be unique within its schema. If a *specific_name* is specified, then routine determination is not used. Routine determination is the process that determines which function to invoke based on the *function_name*, SQL argument list, and the current path of schemas. Refer to the “Search Conditions and Predicates” chapter for more information on routine determination.
Sql_data_access	This clause indicates the usage of SQL statements within the external function of a stored function. Table 1 describes the values that the sql_data_access clause allows.

EXTERNAL NAME external function	The EXTERNAL NAME specifies an external function.
PARAMETER STYLE SQL	This clause represents the parameters being passed according to SQL rules rather than a host language.

Table 1: Sql_data_access Values

Value	*Description*
NO SQL	It signifies that the external function cannot contain any SQL statements.
CONTAINS SQL	It specifies that the external function can contain SQL statements but none that read or modify data.
READS SQL DATA	It specifies that the external function can contain any SQL statement that does not modify SQL data.
MODIFIES SQL DATA	It specifies that the external function can contain any SQL statement that is not a DDL or Transaction Control statement

Example

```
CREATE FUNCTION dateConvert( IN P1 VARCHAR(20) )
RETURNS Date
LANGUAGE Java
NO SQL
EXTERNAL NAME \"SampleExternalMethods::dateConvert\"
PARAMETER STYLE SQL;
```

NOTE: See the “PointBase JDBC Advanced Tutorial” chapter in this guide for more information about functions in PointBase.

CREATE PROCEDURE

```
CREATE PROCEDURE procedure_name([parameter_definition [{,parameter_definition}...]])
       LANGUAGE JAVA
       SPECIFIC specific_name
       sql_data_access
       EXTERNAL NAME external_procedure
       PARAMETER STYLE SQL
```

Using a stored procedure you can return data from a database to a user interface. When the database system returns the data, it is automatically converted from the original value into a user-defined data type value.

Syntax

Procedure_name	*Procedure_name* defines a stored procedure in a schema. The following are usage rules. • Including a schema name is optional. The following syntax is for *procedure_name*: `[schema_name.]procedure_name` • It must be composed of alphanumeric characters or the equivalent in another language, for example, Japanese characters. • It has a maximum limit of 128 characters long. • It is not case sensitive. • It cannot have the same name as a PointBase keyword. • It must be unique in the schema specified.
Parameter_definition	The *parameter_definition* specifies the *parameter_mode*, *parameter_name*, and *parameter_data_type*. (The *parameter_name* is optional.) The *parameter_mode* must be the value, IN. The *parameter_data_type* must be one of the PointBase data types. Separate multiple *parameter_definitions* with a comma. The following syntax is for *parameter_definitions*. `IN [parameter_name] PointBase_data_type`
LANGUAGE JAVA	The clause specifies the language that the stored procedure uses to call the external procedure. It can take the following value: JAVA.

SPECIFIC specific_name	The SPECIFIC *specific_name* clause specifies a name that you can use instead of the *procedure_name* when invoking a stored procedure. The *specific_name* must be unique within its schema. If a *specific_name* is specified, then routine determination is not used. Routine determination is the process that determines which procedure to invoke based on the *procedure_name*, SQL argument list, and the current path of schemas. Refer to the "Search Conditions and Predicates" chapter for more information on routine determination.
Sql_data_access	This clause indicates the usage of SQL statements within the external procedure of a stored procedure. Table 2 describes the values that the *sql_data_access* clause allows.
EXTERNAL NAME external procedure	The EXTERNAL NAME specifies an external procedure.
PARAMETER STYLE SQL	This clause represents the parameters being passed according to SQL rules rather than a host language.

Table 2: Sql_data_access Values

Value	***Description***
NO SQL	It signifies that the external procedure cannot contain any SQL statements.
CONTAINS SQL	It specifies that the external procedure can contain SQL statements but none that read or modify data.
READS SQL DATA	It specifies that the external procedure can contain any SQL statement that does not modify SQL data.
MODIFIES SQL DATA	It specifies that the external procedure can contain any SQL statement that is not a DDL or Transaction Control statement

Example

```
CREATE PROCEDURE getCost( IN P1 VARCHAR(20), IN P2 VARCHAR(2), INOUT P3 FLOAT )
LANGUAGE JAVA
SPECIFIC getCost
DETERMINISTIC
NO SQL
EXTERNAL NAME \"SampleExternalMethods::getCost\"
PARAMETER STYLE SQL;
```

NOTE: See the "PointBase JDBC Advanced Tutorial" chapter in this guide for more information about stored procedures in PointBase.

CREATE TRIGGER

```
CREATE TRIGGER trigger_name
       trigger_action_time triggering_event
       ON table_name
       [REFERENCING referencing_clause [{referencing_clause }...]]
       FOR EACH granularity
       [WHEN (search_condition)]
       trigger_body
```

A trigger can specify additional constraints and business rules within the database to manage the various executions of an application. A trigger operates automatically by executing or firing a DELETE, INSERT, or UPDATE SQL statement on a table with which the trigger is associated. The trigger definitions are saved in the SYSTRIGGERS and SYSTRIGGERCOLUMNS system catalogs.

Syntax

CREATE TRIGGER Trigger_Name	The CREATE TRIGGER keywords are required when creating a trigger. *Trigger_name* defines a unique trigger in a schema. To drop a trigger from a table, you must use the *trigger_name.* Usage Rules for *Trigger_Names* • Including a schema name is optional. The following syntax is for *trigger_name*: `[schema_name.]trigger_name` • It must be composed of alphanumeric characters or the equivalent in another language, for example, Japanese characters. • It has a maximum limit of 128 characters long. • It is not case sensitive. • It cannot have the same name as a PointBase keyword. • It must be unique in the schema specified.
Trigger_Action_Time	*Trigger_action_time* signifies when the trigger can be fired or executed relative to the *triggering_event*. It takes one of the following values: BEFORE or AFTER. If you specify BEFORE as the *trigger_action_time*, the SQL statements in the *trigger_body* cannot directly or indirectly modify SQL data by invoking a stored function or procedure. PointBase does not support cascading BEFORE triggers.
Triggering_Event	*Triggering_event* specifies what type of SQL statement fires or executes the trigger. It can take one of the following SQL statements: DELETE, INSERT, or UPDATE.

Table_Name	*Table_name* specifies the name of the table to which the trigger belongs. A table is allowed to have multiple triggers. If more than one trigger is associated with a table, the triggers are executed in ascending order of their creation timestamps. **Not allowed on a view.** PointBase raises an error if you attempt this on a view.

REFERENCING Referencing_clause	This clause defines correlation or alias names for old and new values of a row. You can use the correlation or alias names in the WHEN (*search_condition*) clause or in SQL statements of the *trigger_body*. The following is the *referencing_clause* syntax for old values of a row.
	`OLD AS correlation_name`
	The following is the *referencing_clause* syntax for new values of a row.
	`NEW AS correlation_name`
	NOTE: Refer toTable 3.

FOR EACH Granularity	This clause defines the granularity of a trigger. The granularity determines the number of times that a trigger will be fired for each *triggering_event*. *Granularity* can take the value of either ROW or STATEMENT. • STATEMENT specifies the *trigger_body* to execute only once regardless of the number of rows being modified (deleted, inserted, or updated) by the *triggering_event* statement. • ROW specifies the *trigger_body* to execute once for each row that is being modified by the *triggering_event* statement. If you do not specify the granularity, the default granularity is STATEMENT.

WHEN (search_condition)	WHEN (*search_condition*) specifies the conditions to execute the *trigger_body* or not. All supported SQL search conditions are allowed in this clause. New row values can be referred to in this section.
Trigger_body	*Trigger_body* specifies an SQL statement that the trigger executes. You can only use one SQL statement in the *trigger_body*. If the *granularity* of the trigger is ROW or STATEMENT, the *trigger_body* can only take the values of the following SQL statements. • VALUES • CALL (only constants can be used. See note.) • SET assignment (for BEFORE Triggers only. See note.) • SIGNAL However, if you use the BEGIN ATOMIC...END keywords, the *trigger_body* can use any applicable SQL statement. Only one statement can be used within the BEGIN ATOMIC...END keywords and must be terminated by a semi colon. **NOTE:** Old and new row values are not supported in the *trigger_body*, except when using the SET assignment statement. Refer to the “Data Control Language” section of this appendix for more information about these statements.

The following table describes the supported combinations of row values, *trigger_action_time*s, *triggering_events*, and *granularity*, when using REFERENCING *referencing_clause*.

Table 3

Row Values	*Trigger_action_time and Triggering_event*	*Granularity*
OLD	BEFORE DELETE BEFORE UPDATE AFTER DELETE AFTER UPDATE	ROW
NEW	BEFORE INSERT BEFORE UPDATE AFTER INSERT AFTER UPDATE	ROW

Examples

To use all of the following trigger examples, you must complete the following:

- Include the SampleExternalMethods.class file in your CLASSPATH when you connect to PointBase.
- Follow the prompts to create a new database called “sample.”
- Type `run sample.sql;` You must type the complete path to the “sample.sql” file located in the directory “<install directory>\pointbase\samples\server_embedded,” for example,
  `run c:/pointbase/samples/server_embedded/sample.sql;`

Example 1

```
CREATE TRIGGER trigger2
BEFORE UPDATE ON product_tbl
REFERENCING NEW AS NEWROW
FOR EACH ROW
WHEN (NEWROW.qty_on_hand < 0)
SET NEWROW.qty_on_hand = 0;

CREATE TRIGGER trigger3
BEFORE UPDATE ON product_tbl
REFERENCING NEW AS NEWROW
FOR EACH ROW
WHEN (NEWROW.purchase_cost < 0)
SIGNAL 'Products prices cannot be negative'

CREATE TRIGGER trigger4
AFTER UPDATE ON product_tbl
REFERENCING NEW AS NEWROW
FOR EACH ROW
WHEN (NEWROW.qty_on_hand > 100)
VALUES(showQuantity('You have increased the quantity above', 100));
```

Example 2

Step 1.

```
CREATE PROCEDURE showTime (IN p1 VARCHAR(30), IN P2 TIMESTAMP)
LANGUAGE JAVA
NO SQL
EXTERNAL NAME "SampleExternalMethods::showTime";
```

Step 2.

```
CREATE TRIGGER trigger1
AFTER INSERT ON discount_code_tbl
FOR EACH ROW
CALL showTime('New discount code inserted' , CURRENT_TIMESTAMP);
```

ALTER TABLE

```
ALTER TABLE table_name alter_table_action [{,alter_table_action},...]
```

The ALTER TABLE statement modifies the structure of a table in the PointBase database. With this statement, constraints or columns may be added or dropped.

Syntax

ALTER TABLE	The ALTER TABLE keywords are required as the first words in an ALTER TABLE statement.
table_name	The *table_name* variable must be the name of an existing table in a PointBase database. The ALTER TABLE statement generates an error if the value of the *table_name* does not exist.
alter_table_action	The action allows adding or dropping a constraint or column. See the following section for the *alter_table_action* syntax.

Alter_Table_Action Syntax

```
ADD table_constraint_definition
| DROP CONSTRAINT constraint_name [CASCADE | RESTRICT]
| ADD [COLUMN] column_definition
| DROP [COLUMN] column_name [CASCADE | RESTRICT]
```

ADD *table_constraint_definition*	Adds a table constraint definition to the table. **Not allowed on a view.** PointBase raises an error if you attempt this on a view. If the constraint is a referential constraint that references a view, an error will be raised.
DROP CONSTRAINT *constraint_name*	Drops an existing named constraint from the table. The system automatically provides a name for the constraint if none was specified when it was added. The constraint name can be found in the table SysTableConstraint.
ADD [COLUMN] *column_definition*	Adds a column to the end of the *column_definition* for the table. (See *column_definition* on page 111.) The default value is NULL, unless declared NOT NULL with an assigned default value. This will only affect columns that you create after the default value is assigned. **Not allowed on a view.** PointBase raises an error if you attempt this on a view.

DROP [COLUMN] *column_name*	Drops one or multiple existing named column(s) from the table. **Not allowed on a view.** PointBase raises an error if you attempt this on a view. If the table_name + column_name is in the system catalog, SysViewTables, then either an error will be raised (if RESTRICT) or all dependent views will be dropped (if CASCADE).
[CASCADE/RESTRICT]	The optional RESTRICT qualifier to a DROP statement allows a drop only if no objects are dependent on the column or constraint. The optional CASCADE qualifier to a DROP statement drops all related objects to the column or constraint.

Examples

```
ALTER TABLE T2 ADD UNIQUE (C1);
ALTER TABLE T2 ADD ORDER_NUM INT;
ALTER TABLE T2 ADD CONSTRAINT constraint_0 FOREIGN KEY (C1) REFERENCES T1 (C1);
ALTER TABLE T2 ADD CONSTRAINT constraint_1 PRIMARY KEY (C1,C2);
ALTER TABLE T2 DROP ORDER_NUM CASCADE;
```

ALTER USER

```
ALTER USER user_name {PASSWORD password | DEFAULT ROLE role_name}
```

To change the password or default role of a database user, you must use the non-standard SQL command, ALTER USER. It can only be used by the following types of users:

- DBA
- Any user having the PBDBA role
- Owner of database

You may also use ALTER USER to change your own password or default role.

Syntax

ALTER USER *user_name*	The *user_name* specifies the name of the user, for whom you will change the password or default role.
PASSWORD *password*	The *password* defines the **new** password for the specified user.
DEFAULT ROLE *role_name*	The *role_name* defines the **new** default role for the specified user.

Examples

```
ALTER USER Scott PASSWORD lion;
ALTER USER Scott DEFAULT ROLE CEO;
```

Dropping SQL Objects

The following sections describes how to drop SQL objects in PointBase:

Drop Behavior (Optional)

Side effects can occur when an SQL object is dropped. For example, if a table is dropped, what becomes of an index that is based on that table? SQL allows you to specify the drop behavior. To do this, specify either: CASCADE or RESTRICT. The syntax for *drop_behavior* is as follows:

CASCADE | RESTRICT

You may specify one or the other. CASCADE has the effect of dropping all SQL objects that are dependent on that object. RESTRICT is the default for the drop behavior. RESTRICT looks to see what objects are dependent on the object being dropped. If there are dependent objects, then the dropping of the object does not occur.

DROP INDEX

```
DROP INDEX table_name.index_name
```

The DROP INDEX statement deletes an index structure of a table from the PointBase database.

Syntax

DROP INDEX	The DROP INDEX keyword is required at the beginning of a DROP INDEX statement.
table_name.index_name	The *index_name* must be the name of an existing index in a PointBase database. The *index_name* must be qualified with the name of the table that the index is on, as in *table_name.index_name*. The DROP INDEX statement raises an error if the value of the *index_name* does not exist.

Examples

```
DROP INDEX ORDER_TBL.ORDER;
```

DROP FUNCTION or DROP PROCEDURE

```
DROP { SPECIFIC routine_type specific_routine_name}
    | {routine_type routine_name [parameter_data_type_list])}
    [drop_behavior]
```

The DROP ROUTINE statement destroys a routine in a schema of a PointBase database.

Syntax

DROP	The DROP keyword is required as the first word in a DROP ROUTINE statement. The SPECIFIC clause refers to a specific function that shares the same name with other functions. *specific_routine_name* must be unique in the database.
specific_routine_name	The specific_routine_name that was specified when the function or procedure was defined.
routine_type	FUNCTION | PROCEDURE | ROUTINE
routine_name	The name of the SQL function or procedure.
parameter_data_type_list	The optional *parameter_list* clause specifies selection criteria for a DROP statement. Only SQL data types are specified. No parameter mode or name is allowed.
drop_behavior	If RESTRICT is specified, then if there are any other SQL routines, or constraints, then the routine is not dropped and neither are the other SQL routines, triggers, nor constraints. With CASCADE, all SQL objects (other SQL routines, and constraints) that use the SQL routine are dropped as well as the SQL routine. RESTRICT is the default.

Examples

```
DROP FUNCTION ORDERS_TOTAL (char(10), int) CASCADE;
```

DROP SCHEMA

```
DROP SCHEMA schema_name [drop_behavior]
```

The DROP schema statement destroys a schema in the PointBase database.

Syntax

`DROP SCHEMA`	The DROP SCHEMA keywords are required as the first words in a DROP SCHEMA statement.
schema_name	The name of the schema. If the schema contains any views, than either PointBase raises an error (if RESTRICT) or drops all views (if CASCADE).
drop_behavior	If RESTRICT is specified, then if there are any tables or SQL routines in *schema_name*, then the schema is not dropped and neither are the tables, nor the SQL routines. With CASCADE, all tables, indexes, columns, constraints, triggers, and SQL routines that are associated with *schema_name* are dropped as well as the schema. RESTRICT is the default.

Examples

```
DROP SCHEMA ORDERS CASCADE;
```

DROP TABLE

```
DROP TABLE table_name [drop_behavior]
```

The DROP TABLE statement destroys a table in the PointBase database.

Syntax

`DROP TABLE`	The DROP TABLE keywords are required as the first words in a DROP TABLE statement.
table_name	The *table_name* variable must be the name of an existing table in a PointBase database. The DROP TABLE statement generates an error if the value of the *table_name* does not exist. If the table is in the system catalog, SysViewTables, then either PointBase raises an error (if RESTRICT) or drops all dependent views (if CASCADE).
drop_behavior	If RESTRICT is specified, then if there are any table constraints, or SQL routines that use *table_name*, then the table is not dropped and neither are the table constraints nor the SQL routines. With CASCADE, all indexes, columns, constraints, triggers, and SQL routines that are associated with *table_name* are dropped as well as the table. RESTRICT is the default.

Examples

```
DROP TABLE ORDER_TBL CASCADE;
```

DROP VIEW

```
DROP VIEW <view name> [ RESTRICT | CASCADE ]
```

This statement removes a specified view or viewed table from the PointBase database.

Notes

- The only objects that can be dependent on a view are other views.

Syntax

`DROP VIEW`	The DROP VIEW keywords are required as the first words in a DROP VIEW statement.
view name	The *view name* variable must be the name of an existing view in the PointBase database.
RESTRICT | CASCADE	RESTRICT verifies if there are any dependent views. If other views depend on this view, an error is raised and this view is not dropped. CASCADE does not verify if there are any dependent views. This view is dropped as well as all dependent views.

Examples

```
DROP VIEW customer_order cascade;
DROP VIEW customer_order restrict;
```

DROP TRIGGER

```
DROP TRIGGER trigger_name
```

The DROP TRIGGER statement deletes a trigger structure from the PointBase database.

Syntax

DROP TRIGGER	The DROP TRIGGER keywords are required as the first words in a DROP TRIGGER statement.
trigger_name	The *trigger_name* is a two-part name which includes the name of the schema. The trigger name must be composed of alphanumeric characters or the equivalent in another language, for example, a word in Japanese characters and cannot be the same as a PointBase keyword. Trigger names in the PointBase database are not case sensitive and can be up to 128 characters long. They must be unique in their schema.

Examples

```
DROP TRIGGER TRG1;
```

DROP USER

DROP USER *user_name* [*drop_behavior*]

The DROP USER statement deletes a user object from the PointBase database. To successfully execute this command, the current user must be the PBSYSADMIN or the database owner. [See "Predefined Users" on page 91.] Or, the current role must be PBDBA . [See "Predefined Roles" on page 95.] **You cannot drop the predefined users: PBPUBLIC or PBSYSADMIN.** Additionally, you cannot create nor drop the user PUBLIC.

Syntax

DROP USER	The DROP USER keyword is required at the beginning of a DROP USER statement.
user_name	The *user_name* must be the name of an existing user in PointBase database. The DROP USER statement raises an error if the value of the *user_name* does not exist.
drop_behavior	If RESTRICT is specified and if any schemas have *user_name* specified, the system does not drop the user and the schema. With CASCADE, the system drops all schemas that have *user_name* as the owner, in addition to dropping the user_name. RESTRICT is the default.

Examples

```
DROP USER ENGINEERING_MANAGER CASCADE;
```

DROP ROLE

PointBase supports this statement. Please refer to the section, "DROP ROLE Syntax" on page 98.

Data Manipulation Language and Data Query Language

To retrieve, INSERT, DELETE and modify data in the PointBase RDBMS, use the Data Manipulation Language (DML) and Data Query Language (DQL). DML and DQL allows an application to do the following:

- SELECT: Retrieve rows of data.
- INSERT: Place new rows of data in the database.
- UPDATE: Replace existing values in the database with new values.
- DELETE: Delete rows of data in the database.

SELECT

```
SELECT [ DISTINCT ] column_list [ AS correlation_name ]
FROM table_expression
   [WHERE search conditions]
   [GROUP BY column_list ]
   [HAVING search_condition ]
   [ORDER BY {column_name | value} [sort_order]]
```

The SELECT statement retrieves data from the PointBase database.

Syntax

SELECT	The SELECT keyword is required as the first word in a SELECT statement.
select_expression	The *select_expression* contains all the information needed to specify the columns and/or SQL scalar expressions that return from a SELECT statement. The *column_list* can be a string of comma-separated column names or the wild card character (*) or expressions. If a column name exists in more than one of the tables in the SELECT statement, a table name or *correlation name* must be used to qualify the column name. You can use a function that returns a single value for each row in the column listing of a SELECT statement. The DISTINCT keyword is optional. When specified, the distinct function eliminates duplicate occurrences of the same row (not columns) and returns only distinct values. The DISTINCT keyword can only be associated with column names in the *column_list* and can only be used once in a query.
FROM	The FROM keyword is required in a SELECT statement between the *select-expression* and the *table-expression*.

table_expression	The *table_expression* contains all the information needed to specify the tables in a SELECT statement and the relationship between multiple tables in the statement. The *table_expression* takes the syntax of:
	`table_expression::=`
	`table_name_exp`
	`| table_name joined_table_exp [join_table_exp...]`
	where:
	`table_name_exp::= table_name [AS correlation_name]`
	`joined_table_exp::= join_type table_name_exp [ON_clause | USING_clause]`
	and the ON_clause or USING_clause are known as the join specification:
	`ON_clause::= ON search_condition`
	`USING_clause::= USING (column_name_list)`
	The table_expression can contain any number of table_names. It does not require you to give any specific ordering of the table_names. The optimizer will determine the appropriate ordering of execution. For more on the optimizer, see "Optimizer Usage in PointBase."

table_name	The names represented by *table_name* are the names of the tables that contain the columns listed in *columns*. If you join more than one table in the SELECT statement, separate the table names with commas. **NOTE:** If more than one table is specified in the table list, then it is known as a join. PointBase supports CROSS, INNER, and LEFT and RIGHT OUTER joins.
AS *correlation_name*	A *correlation name* is a means of giving a different name to a table that qualifies the names of columns in the SELECT statement. A correlation name is sometimes used to document the source of columns even when there are not duplicate column names. It is not required to provide a correlation name for every table in a SELECT statement.
ON_clause	With the ON_clause, you can specify a search_condition when joining two tables. The effect of the ON_clause is the Cartesian product of the two tables that meet the search_condition criteria.

USING	The USING_clause can only be used if each joining table has the same column names as the other joining table. For example, if we have:
	`USING (C1, C2)`
	the effect of the USING_clause is an ON_clause of the following (if we are joining tables T1 and T2):
	`ON T1.C1=T2.C1 AND T1.C2=T2.C2`

WHERE *search conditions*	The WHERE clause is an optional clause that specifies selection criteria for a query. The search condition(s) that follow the WHERE keyword evaluates each row that could be included in the result set. [You may use a subquery as part of the search condition. See “Subqueries” in this section for more information.] If the search conditions returns false for a row, the row is not included in the result set; if the search conditions returns true, the row is included in the result set. If a WHERE clause is not specified, then all rows of the table(s) are included in the result set. For more information on search conditions, see the chapter, “Search Conditions and Predicates.”
GROUP BY *column_list*	The format of the Group-By clause is: `GROUP BY grouping column [ , grouping-column ]...` Grouping-column is a column-reference optionally followed by a collate clause (but only if the column - reference identifies a column whose data type is character string). The collate-clause identifies the collation used for comparing the columns. The result of a group-by-clause is a virtual table, but that result is called a grouped table. The input table is partitioned into one or more groups; the number of groups is the minimum such that, for each grouping -column, no two rows of any group have different values for that grouping – column. For any group in the resulting grouped table, every row in the group has the same value for the grouping - column. Otherwise, the group- by - clause produces an output table that is identical to the input table.

HAVING *search_condition*	The having-clause is a filter. The filtering operation is applied to the grouped table resulting from the preceding clause. If there is a group-by-clause, the grouped table resulting from it is the input to the having-clause. If there is no group-by-clause, the entire table resulting from the where-clause is treated as a grouped table with exactly one group. In this case, there is no grouping-column. The format of the having-clause is: `HAVING search-condition` The search-condition is applied to each group of the grouped table. That's because the only columns of the input table that the having-clause can reference are the grouping columns, unless the columns are used in a set function.

ORDER BY	`ORDER BY {column_name | value} [sort_order]`
	`[{, column_name | value} [sort_order]...}]`
	
	The optional ORDER BY clause specifies the ordering of the rows returned from a SELECT statement. An ORDER BY clause can contain one or more column values, separated with commas; functions are not allowed. If a *column_name* is specified in the ORDER BY clause, then that *column_name* must also be specified in the *column_list*.
	Each column or value in the ORDER BY clause can include an optional *sort_order* qualifier. Acceptable sort order qualifiers are ASC, for ascending sort order, and DESC, for descending sort order. If no sort order is specified, the default is ascending. If the ORDER BY clause contains multiple columns, the order of the columns designates the order of the grouping.
	*If a query contains **any** UNION operators, the ORDER BY clause must be specified last after all the unions are specified.*

NOTE: The SELECT statement returns the qualified result set to the calling application. For more information on how PointBase optimizes SELECT statements and the joins they contain, see the chapter, “Optimizer Usage in PointBase.”

Examples

All of the following examples were created using the sample database that comes with every database product.

Example 1

When querying a column that is not unique, the keyword DISTINCT will allow you eliminate duplicate rows. The ORDER BY clause will sort one or more columns based on ascending or descending sequences. By default the sort order is set to ascending sequence.

```
SELECT DISTINCT name FROM manufacture_tbl ORDER BY name DESC;
```

Results:

NAME
Zetsoft
World Savings
Wells Fargo
Toshiba
Sony
SoftClip
Sams Publishing
Rico Enterprises
MicroSoft
Matrox

Example 2

It is possible to use an SQL constant that will help produce results that are easier to interpret. The example below illustrates two variations of SQL constants. The first example 'Shipping Cost' demonstrates a fixed column type and the second example '$' is concatenated to a select list. Also notice the comparison test that finds the all records that were charged over $300 in shipping costs and not shipped to Florida.

```
SELECT order_num, sales_tax_st_cd, 'Shipping Cost', '$' || shipping_cost FROM order_tbl
WHERE shipping_cost > 300 AND UPPER(sales_tax_st_cd) NOT LIKE '%FL' ORDER BY order_num
ASC;
```

Results:

ORDER_NUM	SALES_TAX	Shipping Cost	'$' || shipping
10398002	TX	Shipping Cost	$359.99
10398009	CA	Shipping Cost	$700
20598101	MI	Shipping Cost	$2500
30198001	NY	Shipping Cost	$2000.99
30298004	NY	Shipping Cost	$700

Joins

Relational join operations are implemented through the basic SELECT...WHERE statement. See SELECT for additional information. PointBase supports the following join operations:

- CROSS JOIN
- INNER JOIN
- OUTER JOIN

CROSS JOIN

The cross join operation performs a cross product on the joining tables.

```
SELECT *
FROM t1 CROSS JOIN t2
```

The cross join is the same type of join found in earlier versions of SQL. Those versions of SQL that did not use the JOIN keyword, used a comma instead.

INNER JOIN

In inner joins, columns with the same names have compatible data types and the rows will be selected only when every matching column has the same value as its data type.

```
SELECT *
FROM t1 INNER JOIN t2
ON t1.c1 = t2.c3;
```

INNER JOIN Example:

This example is joining common values from the sales_rep table and sales tax code table based on a common type ‘decimal rate’. As you can see, it is returning all rows that have a common rate and commission values. Also notice that the data is being filtered base on a tax code rate that is over 7.0.

```
SELECT last_name, commission_rate, sales_tax_code_tbl.rate from sales_rep_tbl INNER JOIN
sales_tax_code_tbl ON (sales_rep_tbl.commission_rate = sales_tax_code_tbl.rate) AND
(sales_tax_code_tbl.rate > 7.0);
```

The SELECT statement returns the following:

LAST_NAME	COMMISSION	RATE
Longer	8	8
Hillerger	9	9
Smith	7.75	7.75
Smith	7.75	7.75
Smith	7.75	7.75
Smith	7.75	7.75
Donohue	7.75	7.75
Donohue	7.75	7.75
Donohue	7.75	7.75
Donohue	7.75	7.75

OUTER JOIN

Outer join operations preserve unmatched rows from one or both tables, depending on the keyword used. PointBase supports the following:

- LEFT OUTER JOIN
- RIGHT OUTER JOIN

LEFT OUTER JOIN

The LEFT OUTER JOIN preserves unmatched rows from the left table, the one that precedes the keyword JOIN

```
SELECT *
FROM t1 LEFT OUTER JOIN t2
ON t1.c1=t2.c3;
```

LEFT OUTER JOIN Example:

The example below is performing a Left Outer Join based on where the sales representative commission rate and the sales tax code table's rate are equal. Notice that all of the values in the left table (sales_rep_tbl) are preserved.

```
SELECT last_name, ytd_sales, commission_rate, sales_tax_code_tbl.rate FROM sales_rep_tbl
LEFT
OUTER JOIN   sales_tax_code_tbl ON (sales_rep_tbl.commission_rate =
sales_tax_code_tbl.rate) AND (sales_tax_code_tbl.rate > 6.0) AND
(sales_rep_tbl.commission_rate >= 8);
```

The SELECT statement returns the following:

LAST_NAME	YTD_SALES	COMMISSION	RATE
Longer	80000	8	8
Hillerger	675000	9.5	9.5
Valentine	857000	9	NULL
Smith	950000	8.75	NULL

RIGHT OUTER JOIN

The RIGHT OUTER JOIN operates similarly to a LEFT OUTER JOIN except the RIGHT or second named table of unmatched rows are preserved.

```
SELECT *
FROM t1 RIGHT OUTER JOIN t2
ON t1.c1=t2.c3;
```

Right Outer Join Example:

This example is using a right outer join to display all distinct unmatched records from the sales tax code table based the sales_rep table.

```
SELECT DISTINCT sales_tax_code_tbl.rate from sales_rep_tbl RIGHT OUTER JOIN
sales_tax_code_tbl ON (sales_rep_tbl.commission_rate = sales_tax_code_tbl.rate) AND
(sales_tax_code_tbl.rate > 8.0);
```

The SELECT statement returns the following:

RATE
8.25
8.5
9.5
9.75
10.25
11.5
13

UNION operator

One of the core SQL operators in conjunction with the SELECT statement is the UNION operator. It is a relational operator that combines the output of two SELECT statements; that is, they combine two or more tables whose respective column data types are of the same family data type. For example, a UNION on a CHARACTER and VARCHAR will work because they are part of the String data type family. A SMALLINT and an INTEGER UNION will also work, because they are part of the exact NUMERIC data type family.

The UNION operator has two forms: the first, UNION DISTINCT, returns only unique rows from a query and discards any duplicate rows; the second, UNION ALL, does not discard duplicate rows; it returns all rows from the first SELECT statement followed by all rows from the second SELECT statement. You may not mix UNION ALL and UNION DISTINCT in the same query scope. However, you may have UNION ALL in the main query and UNION DISTINCT in a subquery, for example. If a query has more than one UNION operator, they must be the same form of UNION operators. You will receive an error if you mix two different forms of the UNION operator in a query.

The output column names resulting from a UNION will have the same column names that the expressions in the very first SELECT statement had. If the UNION query uses the ORDER BY clause, PointBase will order the final results after evaluating all UNIONs. The ORDER BY clause must be last in the query—after specifying all of the UNIONs. Any column names in the ORDER BY clause must refer to the column names in the very first SELECT statement in the query, as the ORDER BY clause sorts the final results by the output column names.

Union Examples:

This example is combining two character columns from the office table and product table. The results will include *all* of the rows of data from each table.

```
SELECT type_code FROM office_tbl UNION ALL SELECT prod_code FROM product_code_tbl;
```

The SELECT statement returns the following:

```
type_code
----------
A
R
R
R
R
R
R
R
W
BK
CB
FW
HW
MS
SW
```

This example uses the columns as in the previous example; however, it uses UNION DISTINCT and orders the results by “type_code.” The result will not return any duplicate rows.

```
SELECT type_code FROM office_tbl UNION DISTINCT SELECT prod_code FROM product_code_tbl
order by type_code;
```

```
type_code
----------
A
BK
CB
FW
HW
MS
R
SW
W
```

Subqueries

Subqueries can be either a SELECT statement or an expression that you can use in any DML statement, for example, SELECT, INSERT, DELETE, UPDATE. The following describes different types of subqueries that PointBase supports.

Subquery Type	*Description*
Scalar Subquery	A subquery that returns at most one row and one column.
Table Subquery (with one column)	A subquery that may return any number of rows within one column. A table subquery may only appear on the right hand side of a quantified comparison predicate. This type of predicate compares a single row value of a table to potentially multiple result row values from a subquery. PointBase supports table subqueries only in a quantified comparison predicate that uses the quantified operators, IN, NOT IN, EXISTS, or NOT EXISTS. Also see "Predicates" on page 70 for more information about these quantified operators.
Non-correlated Subquery	A subquery that does not use a correlated (outer) reference. It references a column, which an enclosing (outer) query block does not define.
Correlated Subquery	A subquery that uses a correlated reference, sometimes referred to as an “outer reference”. It references a column, which an enclosing (outer) query block defines.
Nested Subqueries	A subquery located within another subquery. PointBase supports any level of nested subqueries.

Notes on PointBase Subqueries

- PointBase allows a subquery to return multiple values using the quantified operators, EXITS, NOT EXISTS, IN, or NOT IN only. See "Predicates" on page 70 for more information about IN, NOT IN, EXISTS, or NOT EXISTS.
- Currently, PointBase does not support row subqueries.

Scalar Subquery (Non-correlated) Example

This example retrieves the names of all sales people in the Miami office.

```
SELECT a.first_name, a.last_name
FROM sales_rep_tbl a
WHERE a.office_num  =
( SELECT b.office_num
FROM office_tbl b
WHERE city = 'Miami' );
```

Results:

FIRST_NAME	LAST_NAME
John	Longer

Scalar Subquery (Correlated) Example

This example retrieves the cities of all the offices whose target sales exceed all the sales representative's quotas working in them.

```
SELECT a.city
FROM office_tbl a
WHERE a.target_sales >
( SELECT sum(b.quota)
FROM sales_rep_tbl b
WHERE b.office_num = a.office_num);
```

Results:

CITY
Miami
Atlanta
San Mateo
San Francisco
San Diego
Oakland
Detroit
New York

Table Subquery (Non-correlated) Example

This example retrieves the names of all sales reps working in the western region.

```
SELECT a.first_name, a.last_name
FROM sales_rep_tbl a
WHERE a.office_num IN
( SELECT b.office_num
FROM office_tbl b
WHERE b.region = 'Western');
```

Results:

FIRST_NAME	LAST_NAME
Heather	Smith
George	Valentine
Raymond	Brown
Jack	Smith

Table Subquery (Correlated) Example

This example retrieves all cities, in which at least one sales representative works.

```
SELECT a.city
FROM office_tbl a
WHERE EXISTS
( SELECT *
FROM sales_rep_tbl b
WHERE a.office_num = b.office_num);
```

Results:

CITY
Miami
Atlanta
San Mateo
San Francisco
San Diego
Oakland
Detroit
New York

INSERT

```
INSERT INTO table_name  [(insert_column_list)]
 query_expression
```

The INSERT statement adds new rows to a table in a PointBase database.

NOTE: To insert, you must have privileges on the entire table. Partial privilege on some columns will not work because you have to insert some data (null) into other columns.

Syntax

INSERT INTO	The INSERT INTO keywords are required as the first words in an INSERT statement.
table_name	*table_name* identifies the table that will receive the new data specified in the INSERT statement.
(insert_column_list)	The optional list of columns that receive values in an INSERT statement are indicated between parentheses and separated by commas. The order of the list of columns is important, since the first value following the VALUES clause inserts into the first column in the list of columns. Each subsequent column matches with its counterpart in the *query_expression*. The *insert_column_list* is optional. If it is not specified, then an implicit column list is assumed. **Please note:** when inserting a specific value into an IDENTITY column, every row value that follows in that column will continue to have an incremental value based on the *highest* value assigned for that column—even if the highest value was deleted or rolled back. (See "IDENTITY Property for Autoincrement" on page 42.)]
query_expression	The *query_expression* indicates the values that insert into the table in the INSERT statement.

Query_Expression

The *query_expression* can take one of the following forms:

NOTE: PointBase effectively ignores any spaces that trail after a string when using the INSERT statement. This behavior supports the ANSI standard; however, it may vary with other database vendors.

Form 1: Table_values_constructor

The *table_values_constructor* can be lists of values to be inserted into the columns in the *insert_column_list*. The keyword VALUES, as in VALUES(value1, value2, value3), precede the list of table constructor values.

Another variation of the *table_values_constructor* allows more than one row at a time with a single INSERT statement. Each row of data must contain a value for each column in the list of columns that matches the data type of the column. Enclose each row of data in its own set of parentheses.

Examples

The following INSERT statement inserts a row of data with discrete values:

```
INSERT INTO OFFICE_TYPE_CODE_TBL (TYPE_CODE, DESCRIPTION, MISC)
VALUES ('C', 'Caller', NULL);
```

This example inserts into a table where one of the columns has the IDENTITY property. This column will have the ability to autoincrement the values for each row. Note that you can insert values explicitly for the IDENTITY column or allow values to be automatically generated by not explicitly inserting them. Remember that, PointBase will continue to generate incremental values based on the highest value assigned for the column—even if the highest value was deleted or rolled back.

```
CREATE TABLE TAB1(ID INT IDENTITY, NAME VARCHAR(30));
INSERT INTO TAB1(ID,NAME) VALUES(100, 'Palo Alto');
INSERT INTO TAB1(ID,NAME) VALUES(101, 'Menlo Park');
INSERT INTO TAB1(NAME) VALUES('Cupertino');
```

Unicode data values use the “\u” delimiter for each character with PointBase Commander. For example, unicode representation of the French alphabet is the following:

```
\u05d0 through \u05ea
```

such as:

```
INSERT INTO OFFICE_TYPE_CODE_TBL VALUES ('F', 'French', 'gar_on');
```

From a JAVA program, unicode characters are treated like others and may be expressed through their escape literal representation, such as the following:

```
INSERT INTO OFFICE_TYPE_CODE_TBL VALUES ('X', 'French', '\u00f4');
```

Inserting Multiple Rows

A single INSERT statement can use discrete values to insert more than one row of data by nesting the values for rows enclosed in parentheses, such as the following:

```
INSERT INTO OFFICE_TYPE_CODE_TBL VALUES (’B’, ’ ‘Buyer’, ’Decision Maker’), (’S’,
’Seller’, ’ Sales Rep’), (’T’, ’Talker’, ’ Not a Programmer’);
```

In the PointBase Commander or Console, this example uses dynamic SQL where the value is supplied at runtime.

```
INSERT INTO ORDER_TBL(ORDER_NUM, CUSTOMER_NUM, REP_NUM, PRODUCT_NUM, SALES_TAX_ST_CD,
QUANTITY, SHIPPING_COST, SALES_DATE, SHIPPING_DATE, DELIVERY_DATETIME,
FREIGHT_COMPANY)VALUES(?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
{
010398552, 1, 5001, 980001, 'FL', 000010, 449.00, '1998-01-02', '1998-01-02', '1998-01-15
15:00:00', 'Southern Freight'
010398967, 1, 5001, 980001, 'CA', 000010, 449.00, '1998-01-02', '1998-01-02', '1998-01-15
15:00:00', 'California Freight'
};
```

Form 2: DEFAULT VALUES

Default values can be the list of values that are created to be inserted into the table. It will contain the default values as specified in the CREATE TABLE statement. If the default value of a column is the NULL value and null values are not allowed (NOT NULL), then an error is raised.

DEFAULT and NULL can be used to represent values to be inserted into the table. If DEFAULT is specified, then the default value specified (explicitly or implicitly) is inserted into the column. If NULL is specified, then the NULL value is inserted into the column. Note: If an attempt to insert a NULL value in a column and nulls are not allowed (NOT NULL), then an error is raised.

Examples

```
INSERT INTO T2 VALUES (DEFAULT);
```

or

```
INSERT INTO T2 VALUES (DEFAULT VALUES);
```

Form 3: Query Specification

Query specification is the list of values that you create from an SQL SELECT query. The result set returned from the query must have the same number of column values, with the same data types, as the list of columns in the INSERT statement.

If you duplicate column names between the source table and the target table in a query specification, each table name must have a correlation name and you must qualify the column names with the correlation name.

Example

```
INSERT INTO LOCAL_SALES_TAX_CODE_TBL SELECT * FROM SALES_TAX_CODE_TBL WHERE STATE_CODE =
'FL';
```

UPDATE

```
UPDATE table_name
   SET set_clause_list
   [WHERE search_condition]
```

The UPDATE statement changes the values of data in the table(s) contained in the PointBase database.

Syntax

UPDATE	The UPDATE key word is required as the first word in an UPDATE statement.
table_name	*Table_name* identifies the table that contains the columns to update.
SET	The SET clause is required in an UPDATE statement between the table_name identifier and the list of columns to be updated.

The *set_clause_list* has two possible forms:

```
column_name = value [{, column_name = value}...]
```

or

```
(column_name [{, column_name}...]) = VALUES(value [{,value}...])
```

The list of value expressions sets the value of the columns in the target table. Each value expression includes the name of a column in the table, the equal sign (=), and the new value for the column. The new value for the column can be a constant, a column in the table, DEFAULT keyword, NULL keyword, or a value computed with either one of these value types using an SQL Scalar function.

A single UPDATE statement can update one or more columns in the designated table. If you update more than one column, separate the value expressions with commas.

If DEFAULT is specified, then the default value of the column on the CREATE TABLE is inserted into the column. If NULL is specified, then the NULL value is inserted into the column. If an attempt to insert null value into a column and the column does not allow this (NOT NULL) then an error is raised.

An alternative syntax for the set_clause_list is SET (column list) = VALUES (value list).

WHERE *search_condition*	The WHERE clause specifies selection criteria for an UPDATE statement. The *search_condition* that follows the WHERE keyword evaluates for each row in the indicated table. If the *search_condition* returns true for a row, the columns in the row update with the new values indicated in the UPDATE statement; if the *search_condition* returns false or unknown, the row is ignored by the UPDATE statement.

NOTE: If an UPDATE statement does not contain a WHERE clause, all rows in the target table update with the new values. The UPDATE statement writes new values to rows in a PointBase database, but the changes become permanent only when a COMMIT statement executes following an UPDATE statement, which finalizes changes to the database.

If the UPDATE of a row causes the row to expand past the limits of the page or pages that contained it originally, the PointBase RDBMS will automatically allow the row to span pages. The JDBC calls that execute the UPDATE statement return the number of rows updated.

Examples

```
UPDATE ORDER_TBL SET FREIGHT_COMPANY='Shipping Express',customer_num=25 WHERE
order_num=10398001;
```

DELETE

```
DELETE FROM table_name
  [WHERE search_condition]
```

The DELETE statement deletes a row in a table in a PointBase database.

Syntax

DELETE FROM	The DELETE FROM keyword is required in a DELETE statement.
table_name	The *table_name* is the name of the table from which the selected rows are to be deleted.
`WHERE search_condition`	The optional WHERE clause specifies selection criteria for a DELETE statement. The conditional expression that follows the WHERE keyword is evaluated for each row in the identified table. If the *search_condition* returns true for a row, the row is deleted; if the *search_condition* returns false, the row is not deleted. If no WHERE clause is specified, all rows are deleted from the table. See “Search Conditions and Predicates,” for more information.

The DELETE statement marks rows in the database for deletion. The rows are actually removed when a commit occurs after the statement executes, which completes any changes to the database. For more information on COMMIT, see “Transaction Control.”
The JDBC calls that execute the DELETE statement return the number of rows to be deleted.

Examples

```
DELETE FROM ORDER_TBL
     WHERE SHIPPING_COST <= 275.00;
DELETE FROM ORDER_TBL
     WHERE SHIPPING_COST =?
{
1.00
2.00
3.00
};
```

Data Control Language

To manipulate data, use the Data Control Language (DCL). With DCL, you can perform the following:

- CALL: Execute an SQL procedure.
- RETURN: Return a value from an SQL function.
- SET assignment: Assign a value to an SQL variable.
- SET PATH: Set or change the current path being used to locate the SQL objects in various schemas.
- SIGNAL: Raise an SQLState exception.
- VALUES: Invoke an SQL routine.

CALL

```
CALL procedure_name([argument_list])
```

The CALL statement executes an SQL routine that is a procedure.

Syntax

CALL	The CALL keyword is required in a CALL statement.
procedure_name	The *procedure_name* is the name of the procedure which is executed. No results are returned.
argument_list	The optional *argument_list* clause specifies values for the CALL statement. **NOTE:** Only constants can be used. You cannot use new or old row values.

Examples

```
CALL PROC1();
CALL PROC2('abc');
```

RETURN

```
RETURN SQL_expression
```

The Return statement returns a scalar value from an SQL expression. It can only be used within an SQL function.

Syntax

RETURN	The RETURN keyword is required as the first word in a RETURN statement.
SQL_expression	The *SQL_expression* can be a constant, an SQL routine invocation, one of the SQL Scalar functions, an SQL Cast functions, or an SQL Special Register.

Examples

```
RETURN ‘abc’;
RETURN gestlastcount ( );
RETURN ‘Happy New Year’ | | ‘ ‘ CAST (CURRENT_TIMESTAMP TO VARCHAR(30) );
RETURN NULL;
RETURN;
```

SET assignment

```
SET assignment_target = assignment_source
```

You may use SET assignment statements for BEFORE triggers only. The SET assignment statement assigns a value to an SQL Trigger row correlation variable. The SET assignment statement is much like the *set_clause* of an SQL UPDATE statement.

Syntax

SET	The SET keyword is required as the first word in a SET assignment statement.
assignment_target	The *assignment_target* consists of both, an SQL correlation variable of an SQL Trigger and a column_name. The column_name refers the column of the SQL correlation variable. You may use new or old row values.
assignment_source	The *assignment_source* is one or more SQL expressions that can be a constant, an SQL routine invocation, one of the SQL Scalar functions, an SQL Cast functions, or an SQL Special Register. You may not use an SQL correlation variable, however, you can reference new or old row values in the WHEN *search_condition*. *Assignment_source* values are assigned to the *assignment_target*.

Examples

```
SET newrow.inventory = getnewvalue (  );
SET newrow.selldate = CURRENT_DATE;
SET my_newalias.fruitname = ‘apples’;
```

SET PATH

```
SET PATH schema_name [{,schema_name}...]
```

With the SET PATH statement, you can use it to set or change the current path that you are using to locate the SQL objects in various schemas. This results in the setting of the CURRENT_PATH of a SQL session. To find the correct system tables, the schema POINTBASE must be included in the path.

Syntax

SET PATH	The SET PATH keywords are required as the first words in a SET PATH statement.
schema_name	Required keywords to begin the statement.

Examples

```
SET PATH Employees, Engineering, Sales, PointBase;
```

This sets the CURRENT_PATH to the following schemas in the order specified: Employees, Engineering, Sales, and PointBase. If you wish to append the Marketing schema to the CURRENT_PATH so that the order becomes Employees, Engineering, Sales, PointBase, and Marketing, enter:

```
SET PATH CURRENT_PATH, Marketing;
```

If you never execute a SET PATH statement, then the CURRENT_PATH consists of the schema POINTBASE, followed by your existing schema. When a SET PATH statement is issued, it completely replaces the existing CURRENT_PATH, unless CURRENT_PATH is part of the schemas being set in the path.

The order of the schemas in the path is generally crucial. When the database system is looking for SQL objects, it looks for them in each schema (unless explicitly referenced otherwise), starting with the first schema in the path, then the next, etc...., until an SQL object is found that meets the criteria. One way to override the CURRENT_PATH is to explicitly reference the SQL object. For example, to reference a table, you can specify schema_name.table_name. In the above examples, the SQL object of table_name would be searched in the schema of name schema_name.

SIGNAL

```
SIGNAL 'sqlstate_message'
```

With the SIGNAL statement, you can use it to raise an SQLSTATE exception. This statement can only be used within a *trigger_body* or within the body of an SQL routine, whose language type is SQL. This statement will cause an SQLSTATE exception to be thrown and propagated back to your program. You provide the text of the message.

NOTE: The SIGNAL statement rolls back the specific event that activated its trigger and all the changes caused by the trigger, as well as the original SQL statement of the user, which includes all the triggers and cascading actions that it invoked.

Syntax

`SIGNAL`	The SIGNAL keyword is required as the first word in a SIGNAL statement.
sqlstate_message	The *sqlstate_message* is an SQL string literal value. You can specify any text they would like. The actual SQLSTATE code will be ZG014 and the SQL error code is 25014.

Examples

```
SIGNAL'The oranges inventory is empty';
SIGNAL 'The salary of an employee would have been higher than the salary of his/her Man-
ager';
```

VALUES

```
VALUES ( SQL_expression [ { , SQL_expression } ... ] )
```

The VALUES statement is an SQL stand alone SQL statement. It should not be confused with the *values_clause* of an INSERT statement or with the *from_clause* of an SQL Select statement.

Typically, the VALUES statement is used to invoke SQL routines. The VALUES statement discards all SQL expression values returned by either a constant, an SQL routine invocation, one of the SQL Scalar functions, one of the SQL Cast functions, or an SQL Special Register.

Syntax

VALUES	The VALUES keyword is required as the first word in a VALUES statement.
SQL_expression	The *SQL_expression* can be a constant, an SQL routine invocation, one of the SQL Scalar functions, an SQL Cast functions, or an SQL Special Register.

Examples

```
VALUES (addnewfruit( ‘apple’) );
VALUES (increaseorders(200) );
VALUES (CURENT_DATE );
```

Transaction Control

In this section you can find the following transaction control statements:

SAVEPOINT

```
SAVEPOINT savepoint_name
```

The PointBase transaction model supports savepoints. Savepoints allow transactions to be partially rolled back by establishing a point within a transaction. Savepoints are destroyed automatically when a transaction commits.

NOTE: Make sure that auto commit is turned off when using savepoint.

Syntax

SAVEPOINT *savepoint_name*	The *savepoint_name* can either be an SQL identifier or a numeric value with a scale of zero.

Examples

```
SAVEPOINT SVP1;
SAVEPOINT 2;
```

COMMIT

```
COMMIT [WORK]
```

The COMMIT statement successfully terminates a PointBase transaction.

Syntax

`COMMIT [WORK]`	The COMMIT statement takes no qualifiers. The keyword WORK is optional.

Issuing a COMMIT statement ends the current PointBase transaction. The COMMIT causes three basic actions in the PointBase database:

1. Writes any and all changes that have occurred to the data during the current transaction to the database.
2. Releases any locks that have been placed on data in the PointBase database.
3. Destroys any result sets that have been returned from a query.

Examples

```
COMMIT WORK;
```

RELEASE SAVEPOINT

```
RELEASE SAVEPOINT savepoint_name
```

The RELEASE SAVEPOINT statement destroys a savepoint within a transaction and all the savepoints created after the specified savepoint. The savepoint is automatically released when a COMMIT or ROLLBACK occurs.

The savepoint name specified in this command should have been created earlier by a savepoint command in the current transaction. If the savepoint name is not found, an exception is raised for the invalid savepoint name.

NOTE: Make sure that autocommit is turned off when using savepoint.

Syntax

RELEASE SAVEPOINT *savepoint_name*	The *savepoint_name* can either be an alphanumeric SQL identifier or an integer number.

Example 1

```
RELEASE SAVEPOINT SVP1;
RELEASE SAVEPOINT 2;
```

Example 2

```
CREATE TABLE T1 (c1 int);
Savepoint sp1;
INSERT INTO T1 values (1);
Savepoint sp2;
INSERT INTO T1 values (2);
Savepoint sp3;
INSERT INTO T3 values (3);
RELEASE savepoint sp2;
```

NOTE: In the last statement of Example 2, the savepoint sp2 is destroyed.

ROLLBACK

```
ROLLBACK [WORK] [TO SAVEPOINT savepoint_name]
```

The ROLLBACK statement rolls back any changes that have taken place in a PointBase transaction to the beginning of the transaction or to a savepoint.

A ROLLBACK TO SAVEPOINT statement allows you to undo all changes to the database back to the savepoint. **This action does not terminate a transaction.** If a ROLLBACK statement references a savepoint, then the transaction rolls back to where the savepoint was specified.

NOTE: Make sure that auto commit is turned off when using savepoint.

Syntax

ROLLBACK TO SAVEPOINT *savepoint_name*	The *savepoint_name* can either be an SQL identifier or a numeric value with a scale of zero.

Examples

```
ROLLBACK WORK;
ROLLBACK WORK TO SAVEPOINT SVP1;
```

Issuing a ROLLBACK statement restores the data changed in a transaction to the values that existed before the PointBase transaction began. If you specify a *savepoint_name*, then all changes made to data in the transaction, after the SAVEPOINT *savepoint_name* statement was executed, rolls back. The specified savepoint and all savepoints issued subsequent to this savepoint are destroyed. The transaction resumes after the savepoint statement.

A ROLLBACK statement **without any qualifier** ends the current transaction, which causes two actions in the PointBase database:

1. Releases any locks that have been placed on data in the PointBase database.
2. Destroys any result sets that have been returned from a query.

SET DATALOG

```
SET DATALOG OFF | ON FOR TABLE table_name
```

The SET DATALOG command allows administrators to turn OFF or ON data logging for a specific table. By default, data logging is set to ON for all tables. When set to OFF, deletions or updates are not allowed on the specified tables. **You should turn DATALOG to OFF for insertions only.** If the specified table has one or more indexes, during insertions its indexes will automatically be updated and the index will be logged.

No transaction should be active while executing a SET DATALOG command. PointBase recommends that you execute this command just after a ROLLBACK or a COMMIT statement and before a START TRANSACTION ISOLATION LEVEL statement (or any statement that starts a transaction.) Any transaction that starts after the SET DATALOG statement will turn OFF logging for the specified table. At the end of the transaction, logging is automatically turned back ON. Optionally, before the end of the transaction, you can turn logging ON by setting the ON option in the SET DATALOG statement.

The main purpose of the SET DATALOG statement is to increase performance by turning off data logging while inserting a lot of data (via bulk loading) into a table. The table is locked exclusively by the first insert into the specified table in this transaction. This exclusive lock is then released at the end of the transaction.

Example 1

In the following example, after the COMMIT statement, the data logging is turned OFF for the table T1. The INSERT statement starts a transaction, turns off the data logging for table T1 and inserts all the data from the file ‘data.tab’ into table T1. The final COMMIT commits all the inserted data and turns data logging ON for table T1.

```
commit work;
set datalog off for table T1;
SET BULK ON;
insert into T1 values (?,?,?) use c:\data.tab delimiter tab;
commit work;
```

Example 2

In this example, data logging is turned OFF and one row is inserted into table T2. Although this is allowed, there is no advantage to turning OFF data logging for only a few row inserts.

```
commit work;
set datalog off for table T2;
SET BULK ON;
insert into T2 values (10,20,30);
commit work;
```

START TRANSACTION ISOLATION LEVEL

```
START TRANSACTION ISOLATION LEVEL
 isolation_level [access_mode], [DIAGNOSTICS SIZE diagnostics_size]
```

The START TRANSACTION ISOLATION LEVEL statement is an explicit way to start a transaction.

Syntax

isolation_level	PointBase supports the following transaction isolation levels: • READ UNCOMMITTED • READ COMMITTED • REPEATABLE READ • SERIALIZABLE
access_mode	PointBase supports READ ONLY and READ WRITE access modes. The default mode is READ WRITE. It can only be specified once. If the *access_mode* is not specified, then it is implicitly READ WRITE. In the READ ONLY mode, no modification to date can be made.
DIAGNOSTICS SIZE number_of_conditions	The *diagnostics_size* represents the maximum *number_of_conditions* or SQL exceptions that are saved for each statement that executes. This number lists the number of conditions that can be held at any given time in the diagnostic area. The value must be greater than 0. A default value is defined at implementation time. The *number_of_conditions* can specified only once.

READ UNCOMMITTED

This mode permits Read and Write. It is also known as a ‘dirty read.’ In this mode, all rows, including uncommitted rows are retrieved. For example, if transaction T1 performs one row insert, transaction T2 retrieves that row before T1 ends.

READ COMMITTED

This mode retrieves committed rows only. However, if the same SELECT statement is executed again, the results may differ due to update from other transaction. For example, a transaction T1 retrieves a row, another transaction T2 then updates that row and commits, and T1 then retrieves the same row again. Transaction T1 has retrieved the same row twice, but produced two different values.

Read and Write are permitted with more concurrency. For most users, this mode may satisfy their needs. If a transaction isolation level is not specified in the `pointbase.ini` file, the default is the transaction isolation level, READ_COMMITTED.

REPEATABLE READ

In this mode, only committed rows are retrieved (as in the READ_COMMITTED) but without the problem seen in the READ_COMMITTED isolation level: if the same row is retrieved again in the same transaction, the exact same value is retrieved. However, if a new row is added by another transaction and commits the insert (also delete or update), a second time retrieval for the same select statement may include the newly inserted (also deleted or updated) row. This phenomenon is know as a phantom read.

SERIALIZABLE

This mode is the highest level possible, superior in functionality to a REPEATABLE_READ as no phantom occurs. If a SELECT statement retrieves a collection of rows to satisfy a condition, and the same SELECT statement is executed again in the same transaction, then it is guaranteed to retrieve the same set of rows with the same values.

In this mode, concurrency is reduced compared to other modes. If the number of rows retrieved or affected by the transaction exceeds the number of locks specified in the `pointbase.ini` file, the row level locks are converted to table level locks, further reducing the concurrency. *The default number of locks is 2000.*

Example

```
START TRANSACTION ISOLATION LEVEL SERIALIZABLE, READ WRITE;
```

PointBase-Specific SQL

This section describes non-standard SQL statements that PointBase supports. PointBase has provided these statements to supply additional functionality for your application. Each section represents its own SQL statement. For each of them, the section will summarize the purpose, describe the syntax, explain the usage, and give examples of the statement. You may browse the PointBase-specific SQL statements to discover useful commands.

SHUTDOWN

```
SHUTDOWN [FORCE]
```

To shut down your PointBase Server or Embedded databases, you can use the SHUTDOWN statement. It can shut down either PointBase Server or PointBase Embedded. However, you must be the database owner or the PBSYSADMIN user, or you must have the PBDBA role for your current role to perform the shut down.

Syntax

FORCE	It shuts down the database regardless of open client connections.

Examples

```
SHUTDOWN;
SHUTDOW FORCE;
```

BACKUP

```
BACKUP [<user class name >] [<user param>]
```

This SQL statement initiates online backup. Online backup functionality facilitates database backup while the database application is running. To use this statement, the application must first implement the PointBase interface, “com.pointbase.tools.toolsBackup.” The example in this section describes the PointBase default implementation of this interface.

Online backup has many uses. You can use online backup, when you do not want to bring down the database while taking a backup or when some critical event is recorded in the database, and you want to backup the database immediately. Additionally, having the online backup facility, an application has the flexibility to copy the database to any type of storage it wants, for example, Flash memory.

Important Notes

- You may initiate this statement using PointBase Embedded or Server
- Only the database owner, PBSYSADMIN user, or users with READALL or PBDBA roles are allowed to backup the database
- During online backup, all transactions, including the one that requests write operations, are active— but the write operation will wait for the return from *copyDatabaseFiles()*— which the application must implement; whereas, the read operations continue without any interruption if they can proceed.
- While online backup is in progress the SQL statements will not get lock time-out even if they exceed the regular lock time-out time.
- If CREATE INDEX is in progress then online backup will wait for it to complete.

Syntax

<user class name>	the name of the class which implements the interface, “com.pointbase.tools.toolsBackup.” If this is not given in the statement then the default implementation will be used. (See Example.)
<user param>	the user parameter. This can be quoted identifier in which case it can have comma separated values. If this is not given in the statement then NULL will be passed to the “copyDatabaseFiles()” method.

Example

To accomplish the online backup functionality, you must first implement the interface “com.pointbase.tools.toolsBackup.” Once the interface is implemented, it must be in the classpath with the server database JAR when you launch the application. After launching the application, you can initiate online backup my executing the BACKUP SQL statement.

TIP: Use online backup when the load the database is light, for example, during night times.

Implement toolsBackup Interface

The application needs to implement the toolsBackup interface and the code for copying the database files. The class that implements this interface needs to have a default constructor, for example:

```
interface toolsBackup
{
          public void
    copyDatabaseFiles(String  databaseFiles[], String userParam)
       throws Exception;

}
```

- *databaseFiles[]* is the absolute filenames of all the files for this database.
- *userParam* is a String which application can specify in the online backup SQL statement that will be passed to this method. This can contain such information, like destination directory.

Default Implementation

The class, “toolsBackupDefault,” is the PointBase default implementation for the interface, “com.pointbase.tools.toolsBackup.” In this default implementation, you must write the code that copies the data files to some destination directory. This implementation does not overwrite any files. If the destination directory contains files with the same name of the backup database file then an Exception is raised. If the *userParam* is NULL, then the destination directory is “<database directory>/backup.” <database directory> is the directory of the original database file. If you specify the *userParam*, then it should be a valid existing directory. The file copy is done in blocks of data and the block size is 4096.

The following code describes the PointBase default implementation, “toolsBackupDefault.”

```
package com.pointbase.tools;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;


public class toolsBackupDefault implements toolsBackup
{
    static int COPY_BLOCK_SIZE = 4*1024;

    public toolsBackupDefault()
    {
    }


    public void copyDatabaseFiles( String[] p_databaseFileNames,
        String p_userParams)
        throws Exception
    {
        File l_databaseFiles[] = new File[p_databaseFileNames.length];
        for (int i=0; i < p_databaseFileNames.length; i++)
            l_databaseFiles[i] = new File(p_databaseFileNames[i]);

        String destinationDir;
        if (p_userParams == null)
        {
            destinationDir = l_databaseFiles[0].getParent()+ "/backup";
        }
        else
            destinationDir = p_userParams;

        File fDirectory = new File(destinationDir);

        if (p_userParams == null)
        {
            if (!fDirectory.exists())
                fDirectory.mkdir();
        }

        if (!fDirectory.exists())
            throw new Exception("The destination directory "+ destinationDir + "
does not exist");

        if (!fDirectory.isDirectory())
            throw new Exception("The destination is not a directory");

        // Check if any of the files with the given database file names exist
        // in the destination
        for( int i=0; i<l_databaseFiles.length; i++ )
        {
            File l_destination = new File( destinationDir,
l_databaseFiles[i].getName());
            if (l_destination.exists())
```

```
            throw new Exception("The destination directory already contains
 file " + l_destination);
        }

        // Copy the database files
        for( int i=0; i<l_databaseFiles.length; i++ )
        {
            File l_destination = new File( destinationDir,
 l_databaseFiles[i].getName());
            l_destination.createNewFile();
            copyFile( l_databaseFiles[i], l_destination );
        }
    }

        private void
    copyFile( File fSource, File fDest )
        throws IOException
    {
        InputStream fis = new BufferedInputStream(new FileInputStream(fSource));
        OutputStream fos = new BufferedOutputStream(new
FileOutputStream(fDest));

        int iLen = (int) fSource.length();

        // read the input byte array...
        byte[] buf = new byte[ COPY_BLOCK_SIZE ];
        int toGo = iLen;
        int dataRead;

        while( toGo > 0 )
        {
            toGo -= (dataRead = fis.read( buf));
            fos.write(buf, 0, dataRead);
        }

        fos.close();
        fis.close();
    }
  }
```

Include Implementation in Classpath

Whatever the user implementation of the toolsBackup interface is, the class must be in the classpath with the PointBase Embedded or Server JAR files, when launching the application, for example:

```
java -classpath c:\pbserver42.jar;c:\pbtools42.jar;c:\<userimplementation.class>;
```

The PointBase default implementation is located in the “pbtools” JAR.

Execute BACKUP Statement

The following example executes the BACKUP statement using the PointBase default implementation of the “toolsBackup” interface and specifies the destination directory, “c:/backup/databases.”

```
BACKUP toolsBackupDefault c:/backup/databases;
```

The next example does not specify a destination directory, so the PointBase default implementation copies the backup database file to, “<database directory>/backup.” <database directory> is the directory of the original database file.

```
BACKUP toolsBackupDefault;
```

The following example does not specify an implementation class of “toolsBackup” nor does it specify a destination directory. If this is the case, the PointBase default implementation, “toolsBackupDefault” is used, and the destination directory is “<database directory>/backup.”

```
BACKUP;
```

Appendix B: Unsupported JDBC 2.0 Methods in PointBase

Table 1 describes the unsupported JDBC 2.0 methods from the java.sql package.

Table 1: Unsupported JDBC 2.0 Methods From Java.sql Package

Java.sql Class	*Unsupported Methods*
CallableStatement	getArray(int p_parameterIndex)
	getObject(int p_parameterIndex,java.util.Map p_map)
	getRef(int p_parameterIndex)
	setArray(int p_parameterIndex,Array p_value)
	setRef(int p_parameterIndex,Ref p_value)
Connection	getTypeMap()
	setTypeMap(java.util.Map p_map)
DatabaseMetaData	getUDTs(String p_catalog,String p_schemaPattern,String p_typeNamePattern,int[] p_types)
PreparedStatement	setArray(int p_parameterIndex,Array p_value)
	setRef(int p_parameterIndex,Ref p_value)
ResultSet	getArray(int p_ColumnIndex)
	getArray(String p_ColumnName)
	getObject(int p_ColumnIndex,java.util.Map p_Map)
	getObject(String p_ColumnName,java.util.Map p_Map)
	getRef(int p_ColumnIndex)

Table 1: Unsupported JDBC 2.0 Methods From Java.sql Package

Java.sql Class	*Unsupported Methods*
	getRef(String p_ColumnName)

Appendix C: Reserved Words

PointBase reserves certain words as keywords. Reserved words cannot be used, by themselves, as an identifier for a table, column, or index, or as a correlation name defined in a SELECT statement, unless you delimit them. A delimited identifier is an identifier in double quotes. Any word, including keywords, can be a delimited identifier. A reserved word can be part of an identifier, such as DEFAULT_TABLE, as long as it is not exactly the same as the keyword by itself.

Although CREATE TABLE (VARCHAR VARCHAR(10)) is not a legal PointBase syntax because of the illegal use of the reserved words, “TABLE” and “VARCHAR.” The same identifiers, however, can be legally used if they are delimited, as in CREATE TABLE "TABLE" ("VARCHAR" VARCHAR(10)).

NOTE: The words listed here are SQL reserved words and should not be used. Some of these keywords may not be supported in this release, but are reserved for future releases of PointBase.

Reserved words in the PointBase database are:

```
ACTION
ADD
AFTER
ALL
ALTER
AND
AS
ASC
ASCENDING
AT
ATOMIC
AUTHORIZATION
AVG
BEFORE
BEGIN
BETWEEN
BINARY
```

```
BIT
BLOB
BOOLEAN
BOTH
BY
CALL
CASCADE
CASE
CAST
CHAR
CHARACTER
CHAR_LENGTH
CHARACTER_LENGTH
CHECK
CLOB
COLUMN
COMMIT
COMMITTED
CONSTRAINT
CONTAINS
COUNT
COUNTRY
CREATE
CROSS
CURRENT
CURRENT_DATABASE
CURRENT_DATE
CURRENT_LSN
CURRENT_PATH
CURRENT_SCHEMA
CURRENT_SESSION
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_USER
DATA
DATABASE
DATALOG
DATE
DAY
DEC
DECIMAL
DEFAULT
DEFERRABLE
```

```
DELETE
DESC
DESCENDING
DETERMINISTIC
DIAGNOSTICS
DISCONNECT
DISTINCT
DOUBLE
DROP
EACH
END
EXCEPT
EXECUTE
EXTERNAL
EXTRACT
FALSE
FILTER_COLUMN
FILTER_ROW
FLOAT
FOR
FOREIGN
FROM
FULL
FUNCTION
G
GETLASTLSN
GRANT
GROUP
K
HAVING
HOUR
IMAGE
IMMEDIATE
IN
INDEX
INDEXONLY
INITIALLY
INNER
INOUT
INSERT
INT
INTEGER
```

```
INTO
IS
ISOLATION
JAVA
JOIN
KEY
LANGUAGE
LARGE
LEADING
LEFT
LENGTH
LEVEL
LIKE
LOB
LONG
LONGRAW
LOWER
LSN_CURRENT_ID
LSN_CURRENT_OFFSET
LSN_SKIP_ID
LSN_SKIP_OFFSET
LSN_START_ID
LSN_START_OFFSET
M
MATCH
MAX
METHOD
MIN
MINUTE
MODIFIES
MONTH
NAME
NATURAL
NEW
NO
NOT
NUMBER
NUMERIC
NULL
OBJECT
OCTET_LENGTH
OF
```

```
OFF
OLD
ON
ONLY
OPTION
OR
ORDER
OUT
OUTER
PAGESIZE
PARAMETER
PASSWORD
PATH
PLANONLY
POSITION
PRECISION
PRIMARY
PRIVILEGES
PROCEDURE
PUBLICATION
RAW
READ
READS
REAL
REFERENCES
REFERENCING
RELEASE
REPEATABLE
RESTRICT
RETURN
RETURNS
REVOKE
RIGHT
ROLLBACK
ROUTINE
ROW
SAVEPOINT
SCALAR
SCHEMA
SECOND
SELECT
SERIALIZABLE
```

SESSION_USER

SET
SIGNAL

SIZE
SMALLINT
SNAPSHOT

SPECIFIC

SQLSTATE

STARTSTATEMENT

STYLE

SUBSCRIPTION

SUBSTRING
SUM

SWITCHLOGFILE

SYSDATE

SYSTIME

SYSTIMESTAMP

TABLE
TEXT

TIME
TIMESTAMP
TINYINT

TO

TRAILING

TRANSACTION

TRIGGER

TRIM

TRUE
UNCOMMITTED

UNION

UNIQUE

UNISYNC

UNKNOWN

UPDATE
UPPER
USER

USING

VALUES
VARBINARY

VARCHAR
VARCHAR2

WHEN

```
WHERE
WITH
WRITE
WORK
YEAR
```

Appendix D: SQL Data Type Code

This section contains a mapping of SQL data types and their corresponding type code. These code values are based on the ANSI and ISO SQL standard.

SQL Data Type	*Type Code*
BLOB	30
BOOLEAN	16
CHARACTER	1
CHARACTER VARYING	12
CLOB	40
DATE	91
TIME	92
TIMESTAMP	93
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2
REAL	7
SMALLINT	5