# Table of Contents

1. **Introduction** .................................................................................................................. 5  
1.1 Typographical Conventions Used in this Guide .......................................................... 6  
2. **Advanced Server JDBC Connector Overview** ............................................................ 7  
2.1 JDBC Driver Types ....................................................................................................... 7  
2.2 The JDBC Interface ...................................................................................................... 8  
2.3 JDBC Classes and Interfaces ....................................................................................... 9  
2.4 The JDBC DriverManager ............................................................................................ 10  
3. **JDBC Installation** ........................................................................................................ 11  
3.1 Configuring the Advanced Server JDBC Connector ................................................... 12  
4. **Using the Advanced Server JDBC Connector with Java applications** .................... 13  
4.1 Loading the Advanced Server JDBC Connector ......................................................... 14  
4.2 Connecting to the Database ......................................................................................... 15  
4.2.1 Additional Connection Properties ....................................................................... 16  
4.3 Executing SQL Statements through Statement Objects ............................................. 17  
4.4 Retrieving Results from a ResultSet Object ............................................................. 18  
4.5 Freeing Resources ....................................................................................................... 19  
4.6 Handling Errors ........................................................................................................... 19  
5. **Executing SQL Commands with executeUpdate()** .................................................. 20  
5.1 Using executeUpdate() to INSERT Data .................................................................... 20  
5.1.1 executeUpdate() Syntax Examples ...................................................................... 22  
6. **Adding a Graphical Interface to a Java Program** ....................................................... 23  
7. **Advanced JDBC Connector Functionality** ............................................................... 26  
7.1 Reducing Client-side Resource Requirements ........................................................... 26  
7.1.1 Modifying the Batch Size of a Statement Object ................................................... 27  
7.2 Using PreparedStatements to Send SQL Commands ................................................ 28  
7.3 Executing Stored Procedures ...................................................................................... 29  
7.3.1 Invoking Stored Procedures ................................................................................... 30  
7.3.1.1 Executing a Simple Stored Procedure ............................................................... 30  
7.3.1.2 Executing Stored Procedures with IN parameters ......................................... 31  
7.3.1.3 Executing Stored Procedures with OUT parameters ....................................... 33  
7.3.1.4 Executing Stored Procedures with IN OUT parameters ................................. 34
7.4 Using REF CURSORS with Java ................................................................. 37
  7.4.1 Using a REF CURSOR to retrieve a ResultSet ................................. 38
7.5 Using BYTEA Data with Java ................................................................. 40
  7.5.1 Inserting BYTEA Data into an Advanced Server database .............. 41
  7.5.2 Retrieving BYTEA Data from an Advanced Server database .......... 43
7.6 Using Object Types and Collections with Java .................................. 45
  7.6.1 Using an Object Type ........................................................................ 46
  7.6.2 Using a Collection ........................................................................... 49
7.7 Asynchronous Notification Handling with NoticeListener ............... 52
8 Reference - JDBC Data Types ................................................................. 55
1 Introduction

The JDBC connector (distributed with the Postgres Plus Advanced Server installer) provides connectivity between a Java application and an Advanced Server database. This guide provides installation instructions, usage instructions, and examples that demonstrate the Advanced Server specific functionality of the JDBC Connector.

Throughout the document, examples demonstrate JDBC connector usage:

- Section 4.3 shows how to query an Advanced Server database.
- Section 5 describes using the `executeUpdate()` method to execute SQL commands.
- Section 7.3 explains different methods to invoke stored procedures.
- Section 7.4 demonstrates declaring and using a `REF CURSOR` with Java.
- Section 7.5 demonstrates using `BYTEA` data with Java.
- Section 7.6 demonstrates using object types and collections with Java.

The JDBC connector is written in Java and conforms to Sun's JDK architecture, as described in section 2.1.

The JDBC connector is built on and supports all of the functionality of the PostgreSQL community driver. For more information about the features and functionality of the driver, please refer to the community documentation at:

http://jdbc.postgresql.org/documentation/index.html
1.1 Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various commands, statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions a term refers to any word or group of words which may be language keywords, user-supplied values, literals, etc. A term’s exact meaning depends upon the context in which it is used.

- *Italic font* introduces a new term, typically, in the sentence that defines it for the first time.
- *Fixed-width (mono-spaced) font* is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, etc. For example, `SELECT * FROM emp;`
- *Italic fixed-width font* is used for terms for which the user must substitute values in actual usage. For example, `DELETE FROM table_name;`
- A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).
- Square brackets [] denote that one or none of the enclosed term(s) may be substituted. For example, `[ a | b ]`, means choose one of “a” or “b” or neither of the two.
- Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, `{ a | b }`, means exactly one of “a” or “b” must be specified.
- Ellipses ... denote that the proceeding term may be repeated. For example, `[ a | b ] ...` means that you may have the sequence, “b a a b a”.


2 Advanced Server JDBC Connector Overview

Sun Microsystems created a standardized interface for connecting Java applications to databases known as Java Database Connectivity (JDBC). The Postgres Plus Advanced Server JDBC Connector connects a Java application to a Postgres database.

2.1 JDBC Driver Types

There are currently four different types of JDBC drivers, each with its own specific implementation, use and limitations. The Advanced Server JDBC Connector is a Type 4 driver.

Type 1 Driver
- This driver type is the JDBC-ODBC bridge.
- It is limited to running locally.
- Must have ODBC installed on computer.
- Must have ODBC driver for specific database installed on computer.
- Generally can’t run inside an applet because of Native Method calls.

Type 2 Driver
- This is the native database library driver.
- Uses Native Database library on computer to access database.
- Generally can’t run inside an applet because of Native Method calls.
- Must have database library installed on client.

Type 3 Driver
- 100% Java Driver, no native methods.
- Does not require pre-installation on client.
- Can be downloaded and configured on-the-fly just like any Java class file.
- Uses a proprietary protocol for talking with a middleware server.
- Middleware server converts from proprietary calls to DBMS specific calls.

Type 4 Driver
- 100% Java Driver, no native methods.
- Does not require pre-installation on client.
- Can be downloaded and configured on-the-fly just like any Java class file.
- Unlike Type III driver, talks directly with the DBMS server.
- Converts JDBC calls directly to database specific calls.
2.2 The JDBC Interface

Figure 2.1 shows the core API interfaces in the JDBC specification and how they relate to each other. These interfaces are implemented in the `java.sql` package.

![Figure 2.1 - JDBC Class Relationships](image-url)
2.3 **JDBC Classes and Interfaces**

The core API is composed of classes and interfaces; these classes and interfaces work together as shown in Figure 2.2:

![Diagram showing core classes and interfaces](image)

*Figure 2.2 - Core Classes and Interfaces*
2.4 The JDBC DriverManager

Figure 2.3 depicts the role of the DriverManager class in a typical JDBC application. The DriverManager acts as the bridge between a Java application and the backend database and determines which JDBC driver to use for the target database.
3 JDBC Installation

The JDBC Connector is distributed with and installed the Postgres Plus Advanced Server graphical or RPM installer.

To use the graphical installer to install Advanced Server and the JDBC connector, launch the graphical installer; when the Advanced Server Select Components window opens (shown in Figure 3.1), confirm that the box next to Connectors is checked to include the JDBC Connector in the installation.

![Figure 3.1 - The Advanced Server Select Components Window](image)

You can also install the Advanced Server Connectors without installing a complete version of Advanced Server. Invoke the Advanced Server installation wizard; when the Features dialog opens, uncheck the box to the left of Database Server (and other unwanted components) before choosing Next.

On Linux, you can also use an RPM package to install the `ppas-jdbc` package. For more information about using a package to install the JDBC connector, please see the Postgres Plus Advanced Server Installation Guide, available at:

To use the JDBC Connector, you must have Java installed on your system (version 1.6.0 or later). You can download a Java installer that matches your environment from the Sun Developer Network website at http://java.sun.com/javase/downloads/index.jsp. Detailed installation instructions are available through the associated Docs link on the same page.

### 3.1 Configuring the Advanced Server JDBC Connector

Advanced Server 9.5 ships with the following JDBC drivers:

- `edb-jdbc15.jar` supports JDBC version 3
- `edb-jdbc16.jar` supports JDBC version 4
- `edb-jdbc17.jar` supports JDBC version 4.1

To make the JDBC driver available to Java, you must either copy the appropriate `java.jar` file for the JDBC version that you are using to your `$java_home/jre/lib/ext` directory or append the location of the `.jar` file to the `CLASSPATH` environment variable.

Note that if you choose to append the location of the `.jar` file to the `CLASSPATH` environment variable, you must include the complete pathname:

```
/opt/PostgresPlus/connectors/jdbc/edb-jdbcxx.jar
```
4 Using the Advanced Server JDBC Connector with Java applications

With Java and the Advanced Server JDBC Connector in place, a Java application can access an Advanced Server database. Listing 1.1 creates an application that executes a query and prints the result set.

Listing 1.1

```java
import java.sql.*;
public class ListEmployees {
    public static void main(String[] args) {
        try {
            Class.forName("com.edb.Driver");
            String url = "jdbc:edb://localhost:5444/edb";
            String user = "enterprisedb";
            String password = "enterprisedb";
            Connection con = DriverManager.getConnection(url, user, password);
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
            while (rs.next())
                System.out.println(rs.getString(1));
            rs.close();
            stmt.close();
            con.close();
            System.out.println("Command successfully executed");
        } catch (ClassNotFoundException e) {
            System.out.println("Class Not Found : " + e.getMessage());
        }
        catch (SQLException exp) {
            System.out.println("SQL Exception: " + exp.getMessage());
            System.out.println("SQL State: " + exp.getSQLState());
            System.out.println("Vendor Error: " + exp.getErrorCode());
        }
    }
}
```

This example is simple, but it demonstrates the fundamental steps required to interact with an Advanced Server database from a Java application:

- Load the JDBC driver
- Build connection properties
- Connect to the database server
- Execute an SQL statement
- Process the result set
4.1 Loading the Advanced Server JDBC Connector

The Advanced Server JDBC driver is written in Java and is distributed in the form of a compiled JAR (Java Archive) file. Use the `Class.forName()` method to load the driver. The `forName()` method dynamically loads a Java class at runtime. When an application calls the `forName()` method, the JVM (Java Virtual Machine) attempts to find the compiled form (the bytecode) that implements the requested class.

The Advanced Server JDBC driver is named `com.edb.Driver`:

```java
Class.forName("com.edb.Driver");
```

After loading the bytecode for the driver, the driver registers itself with another JDBC class (named `DriverManager`) that is responsible for managing all the JDBC drivers installed on the current system.

If the JVM is unable to locate the named driver, it throws a `ClassNotFoundException` (which is intercepted with a `catch` block near the end of the program). The `DriverManager` is designed to handle multiple JDBC driver objects. You can write a Java application that connects to more than one database system via JDBC. The next section explains how to select a specific driver.

- Clean up
- Handle any errors that may occur
4.2 Connecting to the Database

After the driver has loaded and registered itself with the DriverManager, the ListEmployees class can attempt to connect to the database server, as shown in the following code fragment:

```java
String url      = "jdbc:edb://localhost:5444/edb";
String user     = "enterprisedb";
String password = "enterprisedb";
Connection con = DriverManager.getConnection(url, user, password);
```

All JDBC connections start with the DriverManager. The DriverManager class offers a static method called getConnection() that is responsible for creating a connection to the database. When you call the getConnection() method, the DriverManager must decide which JDBC driver to use to connect to the database; that decision is based on a URL (Uniform Resource Locator) that you pass to getConnection().

A JDBC URL takes the following general format:

```
jdbc:<driver>:<connection parameters>
```

The first component in a JDBC URL is always jdbc. When using the Advanced Server JDBC Connector, the second component (the driver) is edb.

The Advanced Server JDBC URL takes one of the following forms:

```
jdbc:edb:<database>
jdbc:edb://<host>/<database>
jdbc:edb://<host>:<port>/<database>
```

Table 1.1 shows the various connection parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>The host name of the server. Defaults to localhost.</td>
</tr>
<tr>
<td>port</td>
<td>The port number the server is listening on. Defaults to the Advanced Server standard port number (5444).</td>
</tr>
<tr>
<td>database</td>
<td>The database name.</td>
</tr>
</tbody>
</table>
4.2.1 Additional Connection Properties

In addition to the standard connection parameters, the Advanced Server JDBC driver supports connection properties that control behavior specific to EnterpriseDB. You can specify these properties in the connection URL or as a Properties object parameter passed to DriverManager.getConnection(). Listing 1.2 demonstrates how to use a Properties object to specify additional connection properties:

Listing 1.2

```java
String url = "jdbc:edb://localhost/edb";
Properties props = new Properties();
props.setProperty("user", "enterprisedb");
props.setProperty("password", "enterprisedb");
props.setProperty("sslfactory", "org.postgresql.ssl.NonValidatingFactory");
props.setProperty("ssl", "true");
Connection con = DriverManager.getConnection(url, props);
```

To specify additional connection properties in the URL, add a question mark and an ampersand-separated list of keyword-value pairs:

```java
String url = "jdbc:edb://localhost/edb?user=enterprisedb&ssl=true";
```

Some of the additional connection properties are shown in Table 1.2:

Table 1.2 - Additional Connection Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>String</td>
<td>The database user on whose behalf the connection is being made.</td>
</tr>
<tr>
<td>password</td>
<td>String</td>
<td>The database user’s password.</td>
</tr>
<tr>
<td>Ssl</td>
<td>Boolean</td>
<td>Requests an authenticated, encrypted SSL connection</td>
</tr>
<tr>
<td>loglevel</td>
<td>Integer</td>
<td>The value of loglevel determines the amount of detail printed to the DriverManager’s current value for LogStream or LogWriter. It currently supports values of: com.edb.Driver.DEBUG com.edb.Driver.INFO Set the value of loglevel to INFO to include sparse log information or to DEBUG to produce significant detail.</td>
</tr>
<tr>
<td>charSet</td>
<td>String</td>
<td>The value of charSet determines the character set used for data sent to or received from the database.</td>
</tr>
<tr>
<td>prepareThreshold</td>
<td>Integer</td>
<td>The value of prepareThreshold determines the number of PreparedStatement executions required before switching to server side prepared statements. The default is five.</td>
</tr>
</tbody>
</table>
4.3 Executing SQL Statements through Statement Objects

After loading the Advanced Server JDBC Connector driver and connecting to the server, the code in the sample application builds a JDBC Statement object, executes an SQL query, and displays the results.

A Statement object sends SQL statements to a database. There are three kinds of Statement objects. Each is specialized to send a particular type of SQL statement:

- A Statement object is used to execute a simple SQL statement with no parameters.
- A PreparedStatement object is used to execute a pre-compiled SQL statement with or without IN parameters.
- A CallableStatement object is used to execute a call to a database stored procedure.

You must construct a Statement object before executing an SQL statement. The Statement object offers a way to send a SQL statement to the server (and gain access to the result set). Each Statement object belongs to a Connection; use the createStatement() method to ask the Connection to create the Statement object.

A Statement object defines several methods to execute different types of SQL statements. In the sample application, the executeQuery() method executes a SELECT statement:

```java
Statement stmt = con.createStatement();
ResultSet rs   = stmt.executeQuery("SELECT * FROM emp");
```

The executeQuery() method expects a single argument: the SQL statement that you want to execute. executeQuery() returns data from the query in a ResultSet object. If the server encounters an error while executing the SQL statement provided, it throws an SQLException (and does not return a ResultSet).
4.4 Retrieving Results from a ResultSet Object

A ResultSet object is the primary storage mechanism for the data returned by an SQL statement. Each ResultSet object contains both data and metadata (in the form of a ResultSetMetaData object). ResultSetMetaData includes useful information about results returned by the SQL command: column names, column count, row count, column length, and so on.

To access the row data stored in a ResultSet object, an application calls one or more getter methods. A getter method retrieves the value in particular column of the current row. There are many different getter methods; each method returns a value of a particular type. For example, the getString() method returns a STRING type; the getDate() method returns a Date, and the getInt() method returns an INT type. When an application calls a getter method, JDBC tries to convert the value into the requested type.

Each ResultSet keeps an internal pointer that points to the current row. When the executeQuery() method returns a ResultSet, the pointer is positioned before the first row; if an application calls a getter method before moving the pointer, the getter method will fail. To advance to the next (or first) row, call the ResultSet’s next() method. ResultSet.next() is a boolean method; it returns TRUE if there is another row in the ResultSet or FALSE if you have moved past the last row.

After moving the pointer to the first row, the sample application uses the getString() getter method to retrieve the value in the first column and then prints that value. Since ListEmployees calls rs.next() and rs.getString() in a loop, it processes each row in the result set. ListEmployees exits the loop when rs.next() moves the pointer past the last row and returns FALSE.

```java
while(rs.next())
{
    System.out.println(rs.getString(1));
}
```

When using the ResultSet interface, remember:

- You must call next() before reading any values. next() returns true if another row is available and prepares the row for processing.
- Under the JDBC specification, an application should access each row in the ResultSet only once. It is safest to stick to this rule, although at the current time, the Advanced Server JDBC driver will allow you to access a field as many times as you want.
- When you’ve finished using a ResultSet, call the close() method to free the resources held by that object.
4.5 Freeing Resources

Every JDBC object consumes some number of resources. A ResultSet object, for example, may contain a copy of every row returned by a query; a Statement object may contain the text of the last command executed, and so forth. It’s usually a good idea to free up those resources when the application no longer needs them. The sample application releases the resources consumed by the ResultSet, Statement, and Connection objects by calling each object’s close() method:

```java
rs.close();
stmt.close();
con.close();
```

If you attempt to use a JDBC object after closing it, that object will throw an error.

4.6 Handling Errors

When connecting to an external resource (such as a database server), errors are bound to occur; your code should include a way to handle these errors. Both JDBC and the Advanced Server JDBC Connector provide various types of error handling. The ListEmployees class (Listing 1.1) demonstrates how to handle an error using try/catch blocks.

When a JDBC object throws an error (an object of type SQLException or of a type derived from SQLException), the SQLException object exposes three different pieces of error information:

- The error message.
- The SQL State.
- A vendor-specific error code.

In the example, the following code displays the value of these components should an error occur:

```java
System.out.println("SQL Exception: " + exp.getMessage());
System.out.println("SQL State:    " + exp.getSQLState());
System.out.println("Vendor Error:  " + exp.getErrorCode());
```

For example, if the server tries to connect to a database that does not exist on the specified host, the following error message is displayed:

```
SQL Exception: FATAL: database "acctg" does not exist
SQL State:  3D000
Vendor Error:  0
```
5 Executing SQL Commands with executeUpdate()

In the previous example ListEmployees executed a SELECT statement using the Statement.executeQuery() method. executeQuery() was designed to execute query statements so it returns a ResultSet that contains the data returned by the query. The Statement class offers a second method that you should use to execute other types of commands (UPDATE, INSERT, DELETE, and so forth). Instead of returning a collection of rows, the executeUpdate() method returns the number of rows affected by the SQL command it executes.

The signature of the executeUpdate() method is:

```
int executeUpdate(String sqlStatement)
```

Provide this method a single parameter of type String, containing the SQL command that you wish to execute.

5.1 Using executeUpdate() to INSERT Data

Listing 1.3 demonstrates using the executeUpdate() method to add a row to the emp table.

NOTE: the following example is not a complete application, only a method – the samples in the remainder of this document do not include the code required to set up and tear down a Connection. To experiment with the example, you must provide a class that invokes the sample code.

Listing 1.3

```java
public void updateEmployee(Connection con) {
    try {
        Console console = System.console();
        Statement stmt = con.createStatement();
        String empno = console.readLine("Employee Number :");
        String ename = console.readLine("Employee Name :");
        int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename) VALUES( "+empno+", "+ename+" )");
        System.out.println("");
        System.out.println("Success - "+rowcount+" rows affected.");
    } catch( Exception err) {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```
The `updateEmployee()` method expects a single argument from the caller, a `Connection` object that must be connected to a Postgres Plus Advanced Server database:

```java
public void updateEmployee(Connection con)
```

Next, `updateEmployee()` prompts the user for an employee name and number:

```java
String empno = console.readLine("Employee Number : ");
String ename = console.readLine("Employee Name : ");
```

`updateEmployee()` concatenates the values returned by `console.readLine()` into an `INSERT` statement and pass the result to the `executeUpdate()` method.

```java
int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename)
VALUES("+empno+","+ename+" )");
```

For example, if the user enters an employee number of 6000 and a name of Jones, the `INSERT` statement passed to `executeUpdate()` will look like this:

```
INSERT INTO emp(empno, ename) VALUES(6000, 'Jones');
```

The `executeUpdate()` method returns the number of rows affected by the SQL statement (an `INSERT` typically affects one row, but an `UPDATE` or `DELETE` statement can affect more). If `executeUpdate()` returns without throwing an error, the call to `System.out.println` displays a message to the user that shows the number of rows affected.

```java
System.out.println("\n");
System.out.println("Success - "+rowcount+" rows affected.");
```

The catch block displays an appropriate error message to the user if the program encounters an exception:

```java
{
   System.out.println("An error has occurred.");
   System.out.println("See full details below.");
   err.printStackTrace();
}
```
5.1.1 `executeUpdate()` Syntax Examples

You can use `executeUpdate()` with any SQL command that does not return a result set. Some simple syntax examples using `executeUpdate()` with SQL commands follow:

To use the `UPDATE` command with `executeUpdate()` to update a row:

```java
stmt.executeUpdate("UPDATE emp SET ename='"+ename+"' WHERE empno="+empno);
```

To use the `DELETE` command with `executeUpdate()` to remove a row from a table:

```java
stmt.executeUpdate("DELETE FROM emp WHERE empno="+empno);
```

To use the `DROP TABLE` command with `executeUpdate()` to delete a table from a database:

```java
stmt.executeUpdate("DROP TABLE tablename");
```

To use the `CREATE TABLE` command with `executeUpdate()` to add a new table to a database:

```java
stmt.executeUpdate("CREATE TABLE tablename (fieldname NUMBER(4,2),
fieldname2 VARCHAR2(30))");
```

To use the `ALTER TABLE` command with `executeUpdate()` to change the attributes of a table:

```java
stmt.executeUpdate("ALTER TABLE tablename ADD COLUMN colname BOOLEAN");
```
6 Adding a Graphical Interface to a Java Program

With a little extra work, you can add a graphical user interface to a program – the next example (Listing 1.4) demonstrates how to write a Java application that creates a JTable (a spreadsheet-like graphical object) and copies the data returned by a query into that JTable.

NOTE: The following sample application is a method, not a complete application. To call this method, provide an appropriate main() function and wrapper class.

Listing 1.4

```java
import java.sql.*;
import java.util.Vector;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
...
public void showEmployees(Connection con)
{
    try
    {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
        ResultSetMetaData rsmd = rs.getMetaData();
        Vector labels = new Vector();
        for(int column = 0; column < rsmd.getColumnCount(); column++)
            labels.addElement(rsmd.getColumnLabel(column + 1));

        Vector rows = new Vector();
        while(rs.next())
        {
            Vector rowValues = new Vector();
            for(int column = 0; column < rsmd.getColumnCount(); column++)
                rowValues.addElement(rs.getString(column + 1));
            rows.addElement(rowValues);
        }

        JTable table = new JTable(rows, labels);
        JFrame jf = new JFrame("Browsing table: EMP (from EnterpriseDB)");
        jf.getContentPane().add(new JScrollPane(table));
        jf.setSize(400, 400);
        jf.setVisible(true);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        System.out.println("Command successfully executed");
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}```
Before writing the `showEmployees()` method, you must import the definitions for a few JDK-provided classes:

```java
import java.sql.*;
import java.util.Vector;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
```

The `showEmployees()` method expects a `Connection` object to be provided by the caller; the `Connection` object must be connected to a Postgres Plus Advanced Server:

```java
public void showEmployees(Connection con)
```

`showEmployees()` creates a `Statement` and uses the `executeQuery()` method to execute an SQL query that generates an employee list:

```java
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

As you would expect, `executeQuery()` returns a `ResultSet` object. The `ResultSet` object contains the metadata that describes the `shape` of the result set (that is, the number of rows and columns in the result set, the data type for each column, the name of each column, and so forth). You can extract the metadata from the `ResultSet` by calling the `getMetaData()` method:

```java
ResultSetMetaData rsmd = rs.getMetaData();
```

Next, `showEmployees()` creates a `Vector` (a one dimensional array) to hold the column headers and then copies each header from the `ResultSetMetaData` object into the vector:

```java
Vector labels = new Vector();
for(int column = 0; column < rsmd.getColumnCount(); column++)
{
    labels.addElement(rsmd.getColumnLabel(column + 1));
}
```

With the column headers in place, `showEmployees()` extracts each row from the `ResultSet` and copies it into a new vector (named `rows`). The `rows` vector is actually a vector of vectors: each entry in the `rows` vector contains a vector that contains the data values in that row. This combination forms the two-dimensional array that you will need to build a `JTable`. After creating the `rows` vector, the program reads through each row in the `ResultSet` (by calling `rs.next()`). For each column in each row, a `getter` method extracts the value at that row/column and adds the value to the `rowValues` vector. Finally, `showEmployees()` adds each `rowValues` vector to the `rows` vector:

```java
Vector rows = new Vector();
```
while(rs.next())
{
    Vector rowValues = new Vector();
    for(int column = 0; column < rsmd.getColumnCount(); column++)
    rowValues.addElement(rs.getString(column + 1));
    rows.addElement(rowValues);
}

At this point, the vector (labels) contains the column headers, and a second two-dimensional vector (rows) contains the data for the table. Now you can create a JTable from the vectors and a JFrame to hold the JTable:

JTable table = new JTable(rows, labels);
JFrame jf = new JFrame("Browsing table: EMP (from EnterpriseDB)");
jf.getContentPane().add(new JScrollPane(table));
jf.setSize(400, 400);
jf.setVisible(true);
System.out.println("Command successfully executed");

The showEmployees() method includes a catch block to intercept any errors that may occur and display an appropriate message to the user:

catch(Exception err)
{
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
}

The result of calling the showEmployees() method is shown in Figure 1.4:

![Image of the showEmployees Window]
7 Advanced JDBC Connector Functionality

The previous example created a graphical user interface that displayed a result set in a JTable. Now we will switch gears and show you some of the more advanced features of the Advanced Server JDBC Connector.

To avoid unnecessary clutter, the rest of the code samples in this document will use the console to interact with the user instead of creating a graphical use interface.

7.1 Reducing Client-side Resource Requirements

The Advanced Server JDBC driver retrieves the results of a SQL query as a ResultSet object. If a query returns a large number of rows, using a batched ResultSet will:

- Reduce the amount of time it takes to retrieve the first row.
- Save time by retrieving only the rows that you need.
- Reduce the memory requirement of the client.

When you reduce the fetch size of a ResultSet object, the driver doesn’t copy the entire ResultSet across the network (from the server to the client). Instead, the driver requests a small number of rows at a time; as the client application moves through the result set, the driver fetches the next batch of rows from the server.

Batched result sets cannot be used in all situations. Not adhering to the following restrictions will make the driver silently fall back to fetching the whole ResultSet at once:

- The client application must disable autocommit.
- The Statement object must be created with a ResultSet type of TYPE_FORWARD_ONLY type (which is the default). TYPE_FORWARD_ONLY result sets can only step forward through the ResultSet.
- The query must consist of a single SQL statement.
7.1.1 Modifying the Batch Size of a Statement Object

Limiting the batch size of a ResultSet object can speed the retrieval of data and reduce the resources needed by a client-side application. Listing 1.5 creates a Statement object with a batch size limited to five rows.

Listing 1.5

```java
// Make sure autocommit is off
conn.setAutoCommit(false);

Statement stmt = conn.createStatement();
// Set the Batch Size.
stmt.setFetchSize(5);

ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
while (rs.next())
    System.out.println("a row was returned.");

rs.close();
stmt.close();
```

The call to `conn.setAutoCommit(false)` ensures that the server won’t close the ResultSet before you have a chance to retrieve the first row. After preparing the Connection, you can construct a Statement object:

```java
Statement stmt = db.createStatement();
```

The following code sets the batch size to five (rows) before executing the query:

```java
stmt.setFetchSize(5);
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

For each row in the ResultSet object, the call to `println()` prints a row was returned.

```java
System.out.println("a row was returned.");
```

Remember, while the ResultSet contains all of the rows in the table, they are only fetched from the server five rows at a time. From the client’s point of view, the only difference between a batched result set and an unbatched result set is that a batched result may return the first row in less time.

Next, we will look at another feature (the PreparedStatement) that you can use to increase the performance of certain JDBC applications.
7.2 Using PreparedStatements to Send SQL Commands

Many applications execute the same SQL statement over and over again, changing one or more of the data values in the statement between each iteration. If you use a Statement object to repeatedly execute a SQL statement, the server must parse, plan, and optimize the statement every time. JDBC offers another Statement derivative, the PreparedStatement to reduce the amount of work required in such a scenario.

Listing 1.6 demonstrates invoking a PreparedStatement that accepts an employee ID and employee name and inserts that employee information in the emp table:

Listing 1.6

```java
public void AddEmployee(Connection con)
{
    try
    {
        Console c = System.console();
        String command = "INSERT INTO emp(empno,ename) VALUES(?,?)";
        PreparedStatement stmt = con.prepareStatement(command);
        stmt.setObject(1,new Integer(c.readLine("ID:")));
        stmt.setObject(2,c.readLine("Name:"));
        stmt.execute();
        System.out.println("The procedure successfully executed.");
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

Instead of hard-coding data values in the SQL statement, you insert placeholders to represent the values that will change with each iteration. Listing 1.6 shows an INSERT statement that includes two placeholders (each represented by a question mark):

```java
String command = "INSERT INTO emp(empno,ename) VALUES(?,?)";
```

With the parameterized SQL statement in hand, the AddEmployee() method can ask the Connection object to prepare that statement and return a PreparedStatement object:

```java
PreparedStatement stmt = con.prepareStatement(command);
```

At this point, the PreparedStatement has parsed and planned the INSERT statement, but it does not know what values to add to the table. Before executing the PreparedStatement, you must supply a value for each placeholder by calling a setter method. setObject() expects two arguments:

- A parameter number; parameter number one corresponds to the first question mark, parameter number two corresponds to the second question mark, etc.
The AddEmployee() method prompts the user for an employee ID and name and calls setObject() with the values supplied by the user:

```java
stmt.setObject(1, new Integer(c.readLine("ID:")));
stmt.setObject(2, c.readLine("Name:"));
```

And then asks the PreparedStatement object to execute the statement:

```java
stmt.execute();
```

If the SQL statement executes as expected, AddEmployee() displays a message that confirms the execution. If the server encounters an exception, the error handling code displays an error message.

### 7.3 Executing Stored Procedures

A stored procedure is a module that is written in EnterpriseDB’s SPL and stored in the database. A stored procedure may define input parameters to supply data to the procedure and output parameters to return data from the procedure. Stored procedures execute within the server and consist of database access commands (SQL), control statements, and data structures that manipulate the data obtained from the database.

Stored procedures are especially useful when extensive data manipulation is required before storing data from the client. It is also efficient to use a stored procedure to manipulate data in a batch program.
7.3.1 Invoking Stored Procedures

The `CallableStatement` class provides a way for a Java program to call stored procedures. A `CallableStatement` object can have a variable number of parameters used for input (IN parameters), output (OUT parameters), or both (IN OUT parameters).

The syntax for invoking a stored procedure in JDBC is shown below. Note that the square brackets indicate optional parameters; they are not part of the command syntax.

```java
{call procedure_name([?, ?, ...])}
```

The syntax to invoke a procedure that returns a result parameter is:

```java
{? = call procedure_name([?, ?, ...])}
```

Each question mark serves as a placeholder for a parameter. The stored procedure determines if the placeholders represent IN, OUT, or IN OUT parameters and the Java code must match. We will show you how to supply values for IN (or IN OUT) parameters and how to retrieve values returned in OUT (or IN OUT) parameters in a moment.

7.3.1.1 Executing a Simple Stored Procedure

Listing 1.7-a shows a stored procedure that increases the salary of each employee by 10%. `increaseSalary` expects no arguments from the caller and does not return any information:

**Listing 1.7-a**

```java
CREATE OR REPLACE PROCEDURE increaseSalary
IS
    BEGIN
        UPDATE emp SET sal = sal * 1.10;
    END;
```

Listing 1.7-b demonstrates how to invoke the `increaseSalary` procedure:

**Listing 1.7-b**

```java
public void SimpleCallSample(Connection con)
{
    try
    {
        CallableStatement stmt = con.prepareCall("{call increaseSalary()}");
        stmt.executeUpdate();
        System.out.println("Stored Procedure executed successfully");
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
    }
}
```
To invoke a stored procedure from a Java application, use a `CallableStatement` object. The `CallableStatement` class is derived from the `Statement` class and, like the `Statement` class, you obtain a `CallableStatement` object by asking a `Connection` object to create one for you. To create a `CallableStatement` from a `Connection`, use the `prepareCall()` method:

```java
CallableStatement stmt = con.prepareCall("{call increaseSalary()}");
```

As the name implies, the `prepareCall()` method prepares the statement, but does not execute it. As you will see in the next example, an application typically binds parameter values between the call to `prepareCall()` and the call to `execute()`. To invoke the stored procedure on the server, call the `execute()` method.

```java
stmt.execute();
```

This stored procedure (`increaseSalary`) did not expect any `IN` parameters and did not return any information to the caller (using `OUT` parameters) so invoking the procedure is simply a matter of creating a `CallableStatement` object and then calling that object’s `execute()` method.

The next section demonstrates how to invoke a stored procedure that requires data (`IN` parameters) from the caller.

### 7.3.1.2 Executing Stored Procedures with IN parameters

The code in the next example first creates and then invokes a stored procedure named `empInsert`; `empInsert` requires `IN` parameters that contain employee information: `empno`, `ename`, `job`, `sal`, `comm`, `deptno`, and `mgr`. `empInsert` then inserts that information into the `emp` table.

Listing 1.8-a creates the stored procedure in the Advanced Server database:

```sql
CREATE OR REPLACE PROCEDURE empInsert(
    pEname  IN VARCHAR,
    pJob    IN VARCHAR,
    pSal    IN FLOAT4,
    pComm   IN FLOAT4,
    pDeptno IN INTEGER,
    pMgr    IN INTEGER
) AS
    DECLARE
        CURSOR getMax IS SELECT MAX(empno) FROM emp;
        max_empno INTEGER := 10;
    BEGIN
```
OPEN getMax;
FETCH getMax INTO max_empno;
INSERT INTO emp(empno, ename, job, sal, comm, deptno, mgr)
VALUES(max_empno+1, pEname, pJob, pSal, pComm, pDeptno, pMgr);
CLOSE getMax;
END;

Listing 1.8-b demonstrates how to invoke the stored procedure from Java:

```java
public void CallExample2(Connection con) {
    try {
        Console c = System.console();
        String commandText = "{call empInsert(?,?,?,?,?,?)}";
        CallableStatement stmt = con.prepareCall(commandText);
        stmt.setObject(1, new String(c.readLine("Employee Name :")));
        stmt.setObject(2, new String(c.readLine("Job :")));
        stmt.setObject(3, new Float(c.readLine("Salary :")));
        stmt.setObject(4, new Float(c.readLine("Commission :")));
        stmt.setObject(5, new Integer(c.readLine("Department No :")));
        stmt.setObject(6, new Integer(c.readLine("Manager")));
        stmt.execute();
    } catch(Exception err) {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

Each placeholder (?) in the command (commandText) represents a point in the command that is later replaced with data:

```java
String commandText = "{call EMP_INSERT(?,?,?,?,?,?)}";
CallableStatement stmt = con.prepareCall(commandText);
```

The `setObject()` method binds a value to an IN or IN OUT placeholder. Each call to `setObject()` specifies a parameter number and a value to bind to that parameter:

```java
stmt.setObject(1, new String(c.readLine("Employee Name :")));
stmt.setObject(2, new String(c.readLine("Job :")));
stmt.setObject(3, new Float(c.readLine("Salary :")));
stmt.setObject(4, new Float(c.readLine("Commission :")));
stmt.setObject(5, new Integer(c.readLine("Department No :")));
stmt.setObject(6, new Integer(c.readLine("Manager")));
```

After supplying a value for each placeholder, this method executes the statement by calling the `execute()` method.
7.3.1.3 Executing Stored Procedures with OUT parameters

The next example creates and invokes an SPL stored procedure called `deptSelect`. This procedure requires one IN parameter (department number) and returns two OUT parameters (the department name and location) corresponding to the department number. The code in Listing 1.9-a creates the `deptSelect` procedure:

Listing 1.9-a

```sql
CREATE OR REPLACE PROCEDURE deptSelect
(
  p_deptno IN INTEGER,
  p_dname OUT VARCHAR,
  p_loc OUT VARCHAR
) AS
DECLARE
cursor deptCursor IS SELECT dname, loc FROM dept WHERE deptno=p_deptno;
BEGIN
  OPEN deptCursor;
  FETCH deptCursor INTO p_dname, p_loc;

  CLOSE deptCursor;
END;
```

Listing 1.9-b shows the Java code required to invoke the `deptSelect` stored procedure:

Listing 1.9-b

```java
public void GetDeptInfo(Connection con)
{
  try
  {
    Console c = System.console();
    String commandText = "{call deptSelect(?,?,?)}";
    CallableStatement stmt = con.prepareCall(commandText);
    stmt.setObject(1, new Integer(c.readLine("Dept No :")));
    stmt.registerOutParameter(2, Types.VARCHAR);
    stmt.registerOutParameter(3, Types.VARCHAR);
    stmt.execute();
    System.out.println("Dept Name: " + stmt.getString(2));
    System.out.println("Location : " + stmt.getString(3));
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

Each placeholder (?) in the command (commandText) represents a point in the command that is later replaced with data:

```java
String commandText = "{call deptSelect(?,?,?)}";
CallableStatement stmt = con.prepareCall(commandText);
```
The `setObject()` method binds a value to an `IN` or `IN OUT` placeholder. When calling `setObject()` you must identify a placeholder (by its ordinal number) and provide a value to substitute in place of that placeholder:

```java
stmt.setObject(1, new Integer(c.readLine("Dept No")));
```

The JDBC type of each `OUT` parameter must be registered before the `CallableStatement` object can be executed. Registering the JDBC type is done with the `registerOutParameter()` method.

```java
stmt.registerOutParameter(2, Types.VARCHAR);
stmt.registerOutParameter(3, Types.VARCHAR);
```

After executing the statement, the `CallableStatement`’s getter method retrieves the `OUT` parameter values: to retrieve a `VARCHAR` value, use the `getString()` getter method.

```java
stmt.execute();
System.out.println("Dept Name: " + stmt.getString(2));
System.out.println("Location : " + stmt.getString(3));
```

In the current example `GetDeptInfo()` registers two `OUT` parameters and (after executing the stored procedure) retrieves the values returned in the `OUT` parameters. Since both `OUT` parameters are defined as `VARCHAR` values, `GetDeptInfo()` uses the `getString()` method to retrieve the `OUT` parameters.

### 7.3.1.4 Executing Stored Procedures with IN OUT parameters

The code in the next example creates and invokes a stored procedure named `empQuery` defined with one `IN` parameter (p_deptno), two `IN OUT` parameters (p_empno and p_ename) and three `OUT` parameters (p_job, p_hiredate and p_sal). `empQuery` then returns information about the employee in the two `IN OUT` parameters and three `OUT` parameters.

Listing 1.10-a creates a stored procedure named `empQuery`:

**Listing 1.10-a**

```sql
CREATE OR REPLACE PROCEDURE empQuery
(
    p_deptno        IN     NUMBER,
    p_empno        IN OUT NUMBER,
    p_ename        IN OUT VARCHAR2,
    p_job          OUT    VARCHAR2,
    p_hiredate     OUT    DATE,
    p_sal          OUT     NUMBER
) IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
    INTO p_empno, p_ename, p_job, p_hiredate, p_sal
```
FROM emp
WHERE deptno = p_deptno
AND (empno = p_empno
OR ename = UPPER(p_ename));
END;

Listing 1.10-b demonstrates invoking the empQuery procedure, providing values for the IN parameters, and handling the OUT and IN OUT parameters:

Listing 1.10-b

```java
public void CallSample4(Connection con)
{
    try
    {
        Console c = System.console();
        String commandText = '{call emp_query(?,?,?,?,?,?)}';
        CallableStatement stmt = con.prepareCall(commandText);
        stmt.setInt(1, new Integer(c.readLine("Department No:")));
        stmt.setInt(2, new Integer(c.readLine("Employee No:")));
        stmt.setString(3, new String(c.readLine("Employee Name:")));
        stmt.registerOutParameter(2, Types.INTEGER);
        stmt.registerOutParameter(3, Types.VARCHAR);
        stmt.registerOutParameter(4, Types.VARCHAR);
        stmt.registerOutParameter(5, Types.TIMESTAMP);
        stmt.registerOutParameter(6, Types.NUMERIC);
        stmt.execute();
        System.out.println("Employee No: " + stmt.getInt(2));
        System.out.println("Employee Name: " + stmt.getString(3));
        System.out.println("Job : " + stmt.getString(4));
        System.out.println("Hiredate : " + stmt.getTimestamp(5));
        System.out.println("Salary : " + stmt.getBigDecimal(6));
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

Each placeholder (?) in the command (commandText) represents a point in the command that is later replaced with data:

```java
String commandText = '{call emp_query(?,?,?,?,?,?)}';
CallableStatement stmt = con.prepareCall(commandText);
```

The `setInt()` method is a type-specific setter method that binds an `Integer` value to an IN or IN OUT placeholder. The call to `setInt()` specifies a parameter number and provides a value to substitute in place of that placeholder:

```java
stmt.setInt(1, new Integer(c.readLine("Department No:")));
stmt.setInt(2, new Integer(c.readLine("Employee No:")));
```

The `setString()` method binds a `String` value to an IN or IN OUT placeholder:

```java
stmt.setString(3, new String(c.readLine("Employee Name:")));
```
Before executing the `CallableStatement`, you must register the JDBC type of each `OUT` parameter by calling the `registerOutParameter()` method.

```java
stmt.registerOutParameter(2, Types.INTEGER);
stmt.registerOutParameter(3, Types.VARCHAR);
stmt.registerOutParameter(4, Types.VARCHAR);
stmt.registerOutParameter(5, Types.TIMESTAMP);
stmt.registerOutParameter(6, Types.NUMERIC);
```

Remember, before calling a procedure with an `IN` parameter, you must assign a value to that parameter with a setter method. Before calling a procedure with an `OUT` parameter, you register the type of that parameter; then you can retrieve the value returned by calling a getter method. When calling a procedure that defines an `IN` `OUT` parameter, you must perform all three actions:

- Assign a value to the parameter.
- Register the type of the parameter.
- Retrieve the value returned with a getter method.
7.4 Using REF CURSORS with Java

A REF CURSOR is a cursor variable that contains a pointer to a query result set returned by an OPEN statement. Unlike a static cursor, a REF CURSOR is not tied to a particular query. You may open the same REF CURSOR variable any number of times with the OPEN statement containing different queries; each time, a new result set is created for that query and made available via the cursor variable. A REF CURSOR can also pass a result set from one procedure to another.

Advanced Server supports the declaration of both strongly-typed and weakly-typed REF CURSORS. A strongly-typed cursor must declare the shape (the type of each column) of the expected result set. You can only use a strongly-typed cursor with a query that returns the declared columns; opening the cursor with a query that returns a result set with a different shape will cause the server to throw an exception. On the other hand, a weakly-typed cursor can work with a result set of any shape.

To declare a strongly-typed REF CURSOR:

```
TYPE <cursor_type_name> IS REF CURSOR RETURN <return_type>;
```

To declare a weakly-typed REF_CURSOR:

```
name SYS_REFCURSOR;
```
### 7.4.1 Using a REF CURSOR to retrieve a ResultSet

The stored procedure shown in Listing 1.11-a (getEmpNames) builds two REF CURSORs on the server; the first REF CURSOR contains a list of commissioned employees in the emp table, while the second REF CURSOR contains a list of salaried employees in the emp table:

**Listing 1.11-a**

```sql
CREATE OR REPLACE PROCEDURE getEmpNames
(
  commissioned IN OUT SYS_REFCURSOR,
  salaried IN OUT SYS_REFCURSOR
) IS BEGIN
  OPEN commissioned FOR SELECT ename FROM emp WHERE comm is NOT NULL;
  OPEN salaried FOR SELECT ename FROM emp WHERE comm is NULL;
END;
```

The RefCursorSample() method (see Listing 1.11-b) invokes the getEmpName() stored procedure and displays the names returned in each of the two REF CURSOR variables:

**Listing 1.11-b**

```java
public void RefCursorSample(Connection con)
{
  try
  {
    con.setAutoCommit(false);
    String commandText = "(call getEmpNames(?,?))";
    CallableStatement stmt = con.prepareCall(commandText);
    stmt.setNull(1, Types.REF);
    stmt.registerOutParameter(1, Types.REF);
    stmt.setNull(2, Types.REF);
    stmt.registerOutParameter(2, Types.REF);
    stmt.execute();
    ResultSet commissioned = (ResultSet)stmt.getObject(1);
    System.out.println("Commissioned employees:");
    while(commissioned.next())
    {
      System.out.println(commissioned.getString(1));
    }
    ResultSet salaried = (ResultSet)stmt.getObject(2);
    System.out.println("Salaried employees:");
    while(salaried.next())
    {
      System.out.println(salaried.getString(1));
    }
  }
  catch(Exception err)
  {
    
  }
```

A CallableStatement prepares each REF CURSOR (commissioned and salaried). Each cursor is returned as an IN OUT parameter of the stored procedure, getEmpNames():

```java
String commandText = "{call getEmpNames(?,?)}";
CallableStatement stmt = con.prepareCall(commandText);

The call to registerOutParameter() registers the parameter type (Types.REF) of the first REF CURSOR (commissioned):

```java
stmt.setNull(1, Types.REF);
stmt.registerOutParameter(1, Types.REF);
```

Another call to registerOutParameter() registers the second parameter type (Types.REF) of the second REF CURSOR (salaried):

```java
stmt.setNull(2, Types.REF);
stmt.registerOutParameter(2, Types.REF);
```

A call to stmt.execute() executes the statement:

```java
stmt.execute();
```

The getObject() method retrieves the values from the first parameter and casts the result to a ResultSet. Then, RefCursorSample iterates through the cursor and prints the name of each commissioned employee:

```java
ResultSet commissioned = (ResultSet)stmt.getObject(1);
while(commissioned.next())
{
    System.out.println(commissioned.getString(1));
}
```

The same getter method retrieves the ResultSet from the second parameter and RefCursorExample iterates through that cursor, printing the name of each salaried employee:

```java
ResultSet salaried = (ResultSet)stmt.getObject(2);
while(salaried.next())
{
    System.out.println(salaried.getString(1));
}
7.5 Using BYTEA Data with Java

The BYTEA data type stores a binary string in a sequence of bytes; digital images and sound files are often stored as binary data. Postgres Plus Advanced Server can store and retrieve binary data via the BYTEA data type.

The following Java sample stores BYTEA data in an Advanced Server database and then demonstrates how to retrieve that data. The example requires a bit of setup; Listings 1.12-a, 1.12-b, and 1.12-c create the server-side environment for the Java example.

Listing 1.12-a creates a table (emp_detail) that stores BYTEA data. emp_detail contains two columns: the first column stores an employee’s ID number (type INT) and serves as the primary key for the table; the second column stores a photograph of the employee in BYTEA format.

Listing 1.12-a

```
CREATE TABLE emp_detail
(
  empno INT4 PRIMARY KEY,
  pic BYTEA
);
```

Listing 1.12-b creates a procedure (ADD_PIC) that inserts a row into the emp_detail table:

Listing 1.12-b

```
CREATE OR REPLACE PROCEDURE ADD_PIC(p_empno IN int4, p_photo IN bytea) AS
BEGIN
  INSERT INTO emp_detail VALUES(p_empno, p_photo);
END;
```

And finally, Listing 1.12-c creates a function (GET_PIC) that returns the photograph for a given employee:

Listing 1.12-c

```
CREATE OR REPLACE FUNCTION GET_PIC(p_empno IN int4) RETURN BYTEA IS
DECLARE
  photo BYTEA;
BEGIN
  SELECT pic INTO photo from EMP_DETAIL WHERE empno = p_empno;
  RETURN photo;
END;
```
7.5.1 Inserting BYTEA Data into an Advanced Server database

Listing 1.13 shows a Java method that invokes the ADD_PIC procedure (see Listing 1.12-b) to copy a photograph from the client file system to the emp_detail table on the server.

Listing 1.13

```java
public void InsertPic(Connection con)
{
    try
    {
        Console c = System.console();
        int empno = Integer.parseInt(c.readLine("Employee No :"));
        String fileName = c.readLine("Image filename :");
        File f = new File(fileName);

        if(!f.exists())
        {
            System.out.println("Image file not found. Terminating... ");
            return;
        }

        CallableStatement stmt = con.prepareCall("{call ADD_PIC(?,
        ?)}");
        stmt.setInt(1, empno);
        stmt.setBinaryStream(2, new FileInputStream(f), (int)f.length());
        stmt.execute();
        System.out.println("Added image for Employee "+empno);
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

InsertPic() prompts the user for an employee number and the name of an image file:

```java
int empno = Integer.parseInt(c.readLine("Employee No :"));
String fileName = c.readLine("Image filename :");
```

If the requested file does not exist, InsertPic() displays an error message and terminates:

```java
File f = new File(fileName);
if(!f.exists())
{
    System.out.println("Image file not found. Terminating...");
    return;
}
```
Next, `InsertPic()` prepares a `CallableStatement` object (`stmt`) that calls the `ADD_PIC` procedure. The first placeholder (?) represents the first parameter expected by `ADD_PIC(p_empno)`; the second placeholder represents the second parameter `p_photo`. To provide actual values for those placeholders, `InsertPic()` calls two setter methods. Since the first parameter is of type `INTEGER`, `InsertPic()` calls the `setInt()` method to provide a value for `p_empno`. The second parameter is of type `BYTEA`, so `InsertPic()` uses a binary setter method; in this case, the method is `setBinaryStream()`:

```java
CallableStatement stmt = con.prepareCall("{call ADD_PIC(?, ?)}");
stmt.setInt(1, empno);
stmt.setBinaryStream(2, new FileInputStream(f), f.length());
```

Now that the placeholders are bound to actual values, `InsertPic()` executes the `CallableStatement`:

```java
stmt.execute();
```

If all goes well, `InsertPic()` displays a message verifying that the image has been added to the table. If an error occurs, the `catch` block displays a message to the user:

```java
System.out.println("Added image for Employee \\
"+empno);
catch(Exception err)
{
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
}
```
### 7.5.2 Retrieving BYTEA Data from an Advanced Server database

Now that you know how to insert BYTEA data from a Java application, Listing 1.14 demonstrates how to retrieve BYTEA data from the server.

**Listing 1.14**

```java
    public static void GetPic(Connection con)
    {
        try
        {
            Console c = System.console();
            int empno = Integer.parseInt(c.readLine("Employee No :"));
            CallableStatement stmt = con.prepareCall("(=?=call GET_PIC(?))");
            stmt.setInt(2, empno);
            stmt.registerOutParameter(1, Types.BINARY);
            stmt.execute();
            byte[] b = stmt.getBytes(1);
            String fileName = c.readLine("Destination filename :");
            FileOutputStream fos = new FileOutputStream(new File(fileName));
            fos.write(b);
            fos.close();
            System.out.println("File saved at \"+fileName+\"\")
        }
        catch(Exception err)
        {
            System.out.println("An error has occurred.");
            System.out.println("See full details below.");
            err.printStackTrace();
        }
     }
```

GetPic() starts by prompting the user for an employee ID number:

```java
    int empno = Integer.parseInt(c.readLine("Employee No :"));
```

Next, GetPic() prepares a CallableStatement with one IN parameter and one OUT parameter. The first parameter is the OUT parameter that will contain the photograph retrieved from the database. Since the photograph is BYTEA data, GetPic() registers the parameter as a Type.BINARY. The second parameter is the IN parameter that holds the employee number (an INT), so GetPic() uses the setInt() method to provide a value for the second parameter.

```java
    CallableStatement stmt = con.prepareCall("(=?=call GET_PIC(?))");
    stmt.setInt(2, empno);
    stmt.registerOutParameter(1, Types.BINARY);
```

Next, GetPic() uses the getBytes getter method to retrieve the BYTEA data from the CallableStatement:

```java
    stmt.execute();
```
byte[] b = stmt.getBytes(1);

The program prompts the user for the name of the file where it will store the photograph:

String fileName = c.readLine("Destination filename :");

The FileOutputStream object writes the binary data that contains the photograph to the destination filename:

```java
FileOutputStream fos = new FileOutputStream(new File(fileName));
fos.write(b);
fos.close();
```

Finally, GetPic() displays a message confirming that the file has been saved at the new location:

```java
System.out.println("File saved at "+fileName+"\n");
```
7.6 Using Object Types and Collections with Java

The SQL CREATE TYPE command is used to create a user-defined object type, which is stored in the Advanced Server database. The CREATE TYPE command is also used to create a collection, commonly referred to as an array, which is also stored in the Advanced Server database.

These user-defined types can then be referenced within SPL procedures, SPL functions, and Java programs.

The basic object type is created with the CREATE TYPE AS OBJECT command along with optional usage of the CREATE TYPE BODY command.

A nested table type collection is created using the CREATE TYPE AS TABLE OF command. A varray type collection is created with the CREATE TYPE VARRAY command.

Example usage of an object type and a collection are shown in the following sections.

Listing 1.15 shows a Java method used by both examples to establish the connection to the Advanced Server database.

```
public static Connection getEDBConnection() throws ClassNotFoundException, SQLException {
    String url = "jdbc:edb://localhost:5444/test";
    String user = "enterprisedb";
    String password = "edb";
    Class.forName("com.edb.Driver");
    Connection conn = DriverManager.getConnection(url, user, password);
    return conn;
}
```
### 7.6.1 Using an Object Type

Create the object types in the Advanced Server database. Object type `addr_object_type` defines the attributes of an address:

```sql
CREATE OR REPLACE TYPE addr_object_type AS OBJECT
(
    street VARCHAR2(30),
    city VARCHAR2(20),
    state CHAR(2),
    zip NUMBER(5)
);
```

Object type `emp_obj_typ` defines the attributes of an employee. Note that one of these attributes is object type `ADDR_OBJECT_TYPE` as previously described. The object type body contains a method that displays the employee information:

```sql
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT
(
    empno NUMBER(4),
    ename VARCHAR2(20),
    addr ADDR_OBJECT_TYPE,
    MEMBER PROCEDURE display_emp(SELF IN OUT emp_obj_typ)
);
CREATE OR REPLACE TYPE BODY emp_obj_typ AS
MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Employee No   : ' || SELF.empno);
        DBMS_OUTPUT.PUT_LINE('Name          : ' || SELF.ename);
        DBMS_OUTPUT.PUT_LINE('Street        : ' || SELF.addr.street);
        DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || SELF.addr.city || ', ' || SELF.addr.state || ' ' || LPAD(SELF.addr.zip,5,'0'));
    END;
END;
```

Listing 1.16 is a Java method that includes these user-defined object types:

```java
public static void testUDT() throws SQLException {
    Connection conn = null;
    try {
        conn = getEDBConnection();
        String commandText = "(call emp_obj_typ.display_emp(?))";
        CallableStatement stmt = conn.prepareCall(commandText);
        // initialize emp_obj_typ structure
        // create addr_object_type structure
        Struct address = conn.createStruct("addr_object_type",
            new Object[]{"123 MAIN STREET","EDISON","NJ",8817});
        Struct emp = conn.createStruct("emp_obj_typ",
            new Object[]{9001,"JONES", address});
        // set emp_obj_typ type param
```
stmt.registerOutParameter(1, Types.STRUCT, "emp_obj_typ");
stmt.setObject(1, emp);
stmt.execute();

// extract emp_obj_typ object
emp = (Struct)stmt.getObject(1);
Object[] attrEmp = emp.getAttributes();
System.out.println("empno: " + attrEmp[0]);
System.out.println("ename: " + attrEmp[1]);

// extract addr_object_type attributes
address = (Struct) attrEmp[2];
Object[] attrAddress = address.getAttributes();
System.out.println("street: " + attrAddress[0]);
System.out.println("city: " + attrAddress[1]);
System.out.println("state: " + attrAddress[2]);
System.out.println("zip: " + attrAddress[3]);
}
}

A CallableStatement object is prepared based on the display_emp() method of the emp_obj_typ object type:

String commandText = "{call emp_obj_typ.display_emp(?)}";
CallableStatement stmt = conn.prepareCall(commandText);

createStruct() initializes and creates instances of object types addr_object_type and emp_obj_typ named address and emp, respectively:

Struct address = conn.createStruct("addr_object_type",
        new Object[]{"123 MAIN STREET","EDISON","NJ",8817});
Struct emp     = conn.createStruct("emp_obj_typ",
        new Object[]{9001, "JONES", address});

The call to registerOutParameter() registers the parameter type (Types.STRUCT) of emp_obj_typ:

stmt.registerOutParameter(1, Types.STRUCT, "emp_obj_typ");

The setObject() method binds the object instance emp to the IN OUT placeholder.

stmt.setObject(1, emp);

A call to stmt.execute() executes the call to the display_emp() method:

stmt.execute();

getObject() retrieves the emp_obj_typ object type. The attributes of the emp and address object instances are then retrieved and displayed:
emp = (Struct) stmt.getObject(1);
Object[] attrEmp = emp.getAttributes();
System.out.println("empno: " + attrEmp[0]);
System.out.println("ename: " + attrEmp[1]);

address = (Struct) attrEmp[2];
Object[] attrAddress = address.getAttributes();
System.out.println("street: " + attrAddress[0]);
System.out.println("city: " + attrAddress[1]);
System.out.println("state: " + attrAddress[2]);
System.out.println("zip: " + attrAddress[3]);
7.6.2 Using a Collection

Create collection types NUMBER_ARRAY and CHAR_ARRAY in the Advanced Server database:

```sql
CREATE OR REPLACE TYPE NUMBER_ARRAY AS TABLE OF NUMBER;
CREATE OR REPLACE TYPE CHAR_ARRAY AS TABLE OF VARCHAR(50);
```

Listing 1.17-a is an SPL function that uses collection types NUMBER_ARRAY and CHAR_ARRAY as IN parameters and CHAR_ARRAY as the OUT parameter.

The function concatenates the employee ID from the NUMBER_ARRAY IN parameter with the employee name in the corresponding row from the CHAR_ARRAY IN parameter. The resulting concatenated entries are returned in the CHAR_ARRAY OUT parameter.

**Listing 1.17-a**

```sql
CREATE OR REPLACE FUNCTION concatEmpIdName
(    arrEmpIds  NUMBER_ARRAY,
    arrEmpNames  CHAR_ARRAY
) RETURN CHAR_ARRAY
AS
DECLARE
    i INTEGER := 0;
    arrEmpIdNames  CHAR_ARRAY;
BEGIN
    arrEmpIdNames := CHAR_ARRAY(NULL,NULL);
    FOR i IN arrEmpIds.FIRST..arrEmpIds.LAST LOOP
        arrEmpIdNames(i) := arrEmpIds(i) || '  ' || arrEmpNames(i);
    END LOOP;
    RETURN arrEmpIdNames;
END;
```

Listing 1.17-b is a Java method that calls the Listing 1.17-a function, passing and retrieving the collection types:

**Listing 1.17-b**

```java
public static void testTableOfAsInOutParams() throws SQLException {
    Connection conn = null;
    try {
        conn = getEDBConnection();
        String commandText = "(? = call concatEmpIdName(? ,? ))";
        CallableStatement stmt = conn.prepareCall(commandText);

        // create collections to specify employee id and name values
        Array empIdArray = conn.createArrayOf("integer",
                      new Integer[]{7900, 7902});
        Array empNameArray = conn.createArrayOf("varchar",
                      new String[]{"JAMES", "FORD"});

        // set TABLE OF VARCHAR as OUT param
        stmt.registerOutParameter(1, Types.ARRAY);
```
// set TABLE OF INTEGER as IN param
stmt.setObject(2, empIdArray, Types.OTHER);

// set TABLE OF VARCHAR as IN param
stmt.setObject(3, empNameArray, Types.OTHER);
stmt.execute();
java.sql.Array empIdNameArray = stmt.getArray(1);
String[] emps = (String[]) empIdNameArray.getArray();

System.out.println("items length: " + emps.length);
System.out.println("items[0]: " + emps[0].toString());
System.out.println("items[1]: " + emps[1].toString());
}
}
}

A CallableStatement object is prepared to invoke the concatEmpIdName() function:

String commandText = "(\? = call concatEmpIdName(??,?,?))";
CallableStatement stmt = conn.prepareCall(commandText);

createArrayOf() initializes and creates collections named empIdArray and empNameArray:

Array empIdArray = conn.createArrayOf("integer",
new Integer[]{7900, 7902});
Array empNameArray = conn.createArrayOf("varchar",
new String[]{"JAMES", "FORD"});

The call to registerOutParameter() registers the parameter type (Types.ARRAY) of the OUT parameter:

stmt.registerOutParameter(1, Types.ARRAY);

The setObject() method binds the collections empIdArray and empNameArray to the IN placeholders:

stmt.setObject(2, empIdArray, Types.OTHER);
stmt.setObject(3, empNameArray, Types.OTHER);

A call to stmt.execute() invokes the concatEmpIdName() function:

stmt.execute();

getArray() retrieves the collection returned by the function. The first two rows consisting of the concatenated employee IDs and names are displayed:
java.sql.Array empIdNameArray = stmt.getArray(1);
String[] emps = (String[]) empIdNameArray.getArray();
System.out.println("items length: " + emps.length);
System.out.println("items[0]: " + emps[0].toString());
System.out.println("items[1]: " + emps[1].toString());
7.7 Asynchronous Notification Handling with NoticeListener

The Advanced Server JDBC Connector provides asynchronous notification handling functionality. A notification is a message generated by the server when an SPL (or PL/pgSQL) program executes a RAISE NOTICE statement. Each notification is sent from the server to the client application. To intercept a notification in a JDBC client, an application must create a NoticeListener object (or, more typically, an object derived from NoticeListener).

It is important to understand that a notification is sent to the client as a result of executing an SPL (or PL/pgSQL) program. To generate a notification, you must execute an SQL statement that invokes a stored procedure, function, or trigger: the notification is delivered to the client as the SQL statement executes. Notifications work with any type of statement object; CallableStatement objects, PreparedStatement objects, or simple Statement objects. A JDBC program intercepts a notification by associating a NoticeListener with a Statement object. When the Statement object executes an SQL statement that raises a notice, JDBC invokes the noticeReceived() method in the associated NoticeListener.

Listing 1.18-a shows an SPL procedure that loops through the emp table and gives each employee a 10% raise. As each employee is processed, adjustSalary executes a RAISE NOTICE statement (in this case, the message contained in the notification reports progress to the client application). Listing 1.18-b will demonstrate how to create a NoticeListener that intercepts each notification.

Listing 1.18-a

```
CREATE OR REPLACE PROCEDURE adjustSalary
IS
    v_empno NUMBER(4);
    v_ename VARCHAR2(10);
    CURSOR emp_cur IS SELECT empno, ename FROM emp;
BEGIN
    OPEN emp_cur;
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        UPDATE emp SET sal = sal * 1.10 WHERE empno = v_empno;
        RAISE NOTICE 'Salary increased for %', v_ename;
    END LOOP;
    CLOSE emp_cur;
END;
```
Listing 1.18-b shows how to intercept notifications in a JDBC application.

Listing 1.18-b

```java
public void NoticeExample(Connection con)
{
    CallableStatement stmt;
    try
    {
        stmt = con.prepareCall("{call adjustSalary()}");
        MyNoticeListener listener = new MyNoticeListener();
        ((BaseStatement)stmt).addNoticeListener(listener);
        stmt.execute();
        System.out.println("Finished");
    }
    catch (SQLException e)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        e.printStackTrace();
    }
}
class MyNoticeListener implements NoticeListener
{
    public MyNoticeListener()
    {
    }

    public void noticeReceived(SQLWarning warn)
    { 
        System.out.println("NOTICE: "+ warn.getMessage());
    }
}
```

The `NoticeExample()` method is straightforward; it expects a single argument, a `Connection` object, from the caller:

```java
public void NoticeExample(Connection con)
{

}
```

`NoticeExample()` begins by preparing a call to the `adjustSalary` procedure shown in example 1.10-a. As you would expect, `con.prepareCall()` returns a `CallableStatement` object. Before executing the `CallableStatement`, you must create an object that implements the `NoticeListener` interface and add that object to the list of `NoticeListeners` associated with the `CallableStatement`:

```java
CallableStatement stmt = con.prepareCall("{call adjustSalary()}");
MyNoticeListener listener = new MyNoticeListener();
((BaseStatement)stmt).addNoticeListener(listener);
```

Once the `NoticeListener` is in place, `NoticeExample` method executes the `CallableStatement` (invoking the `adjustSalary` procedure on the server) and displays a message to the user:

```java
stmt.execute();
```
System.out.println("Finished");

Each time the adjustSalary procedure executes a RAISE NOTICE statement, the server sends the text of the message ("Salary increased for ") to the Statement (or derivative) object in the client application. JDBC invokes the noticeReceived() method (possibly many times) before the call to stmt.execute() completes.

class MyNoticeListener implements NoticeListener
{
    public MyNoticeListener()
    {
    }

    public void noticeReceived(SQLWarning warn)
    {
        System.out.println("NOTICE: "+ warn.getMessage());
    }
}

When JDBC calls the noticeReceived() method, it creates an SQLWarning object that contains the text of the message generated by the RAISE NOTICE statement on the server.

Notice that each Statement object keeps a list of NoticeListeners. When the JDBC driver receives a notification from the server, it consults the list maintained by the Statement object. If the list is empty, the notification is saved in the Statement object (you can retrieve the notifications by calling stmt.getWarnings() once the call to execute() completes). If the list is not empty, the JDBC driver delivers an SQLWarning to each listener, in the order in which the listeners were added to the Statement.
8 Reference - JDBC Data Types

Table 8.1 lists the JDBC data types supported by Advanced Server and the JDBC connector. If you are binding to an Advanced Server type (shown in the middle column) using the `setObject()` method, supply a JDBC value of the type shown in the left column. When you retrieve data, the `getObject()` method will return the object type listed in the right-most column:

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Advanced Server Type</th>
<th><code>getObject()</code> returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>INT4</td>
<td><code>java.lang.Integer</code></td>
</tr>
<tr>
<td>TINYINT, SMALLINT</td>
<td>INT2</td>
<td><code>java.lang.Integer</code></td>
</tr>
<tr>
<td>BIGINT</td>
<td>INT8</td>
<td><code>java.lang.Long</code></td>
</tr>
<tr>
<td>REAL, FLOAT</td>
<td>FLOAT4</td>
<td><code>java.lang.Float</code></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>FLOAT8</td>
<td><code>java.lang.Double</code></td>
</tr>
<tr>
<td>DECIMAL, NUMERIC</td>
<td>NUMERIC</td>
<td><code>java.math.BigDecimal</code></td>
</tr>
<tr>
<td>CHAR</td>
<td>BPCHAR</td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td>VARCHAR, LONGVARCHAR</td>
<td>VARCHAR</td>
<td><code>java.lang.String</code></td>
</tr>
<tr>
<td>DATE</td>
<td>DATE</td>
<td><code>java.sql.Date</code></td>
</tr>
<tr>
<td>TIME</td>
<td>TIME</td>
<td><code>java.sql.Timestamp</code></td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
<td><code>java.sql.Timestamp</code></td>
</tr>
<tr>
<td>BINARY</td>
<td>BYTEA</td>
<td><code>byte[]</code> (primitive)</td>
</tr>
<tr>
<td>BIT</td>
<td>BOOL</td>
<td><code>java.lang.Boolean</code></td>
</tr>
</tbody>
</table>

*Table 8.1 - Supported JDBC Data Types*