CHAPTER 2

Database Metadata, Part 1

Example isn't another way to teach, it is the only way to teach.

Albert Einstein

The goal of this chapter (and the next) is to show you how to use JDBC's database metadata API, which you can use to get information about tables, views, column names, column types, stored procedures, result sets, and databases. It will be of most interest to those who need to write applications that adapt themselves to the specific capabilities of several database systems or to the content of any database. If you write programs—such as graphical user interface (GUI) database applications using database adapters—that use advanced database features or programs that discover database stored procedures and tables or views at runtime (i.e., dynamically), you will have to use metadata. You can use database metadata to

- Discover database schema and catalog information.
- Discover database users, tables, views, and stored procedures.
- Understand and analyze the result sets returned by SQL queries.
- Find out the table, view, or column privileges.
- Determine the signature of a specific stored procedure in the database.
- Identify the primary/foreign keys for a given table.

As you will discover, metadata not only helps you to effectively manage resources, it also helps you find the data you need and determine how best to use it. In addition, metadata provides a structured description of database information resources and services. Some JDBC methods, such as `getProcedureColumns()` and `getProcedures()`, return the result as a `ResultSet` object. Unfortunately, this is not very useful to the client; because the `ResultSet` object cannot be passed to some client programs, these programs cannot analyze and understand the content of the `ResultSet` object. For this reason, you need to return the results in XML (and possibly an XML object serialized as a `String` object), which is suitable for all clients. To be efficient, you generate XML expressed as a `String` object, which can be easily converted to an XML document (using the `org.w3c.dom.Document` interface). The Source Code section of the Apress website provides utilities for converting `Strings` to `org.w3.dom.Document` and `org.w3.dom.Document` objects to `Strings`.

When you write JDBC applications, you should strive for good performance. But what is "good" performance? Should it be subjective or objective? This depends on the requirements

of your application. In other words, if you write "slow" code, the JDBC driver does not throw an exception, but you get a performance hit (which might translate to losing clients). "Good" performance means that you are satisfying your project's performance requirements, which should be defined precisely in requirements and design documents. To get acceptable performance from JDBC drivers, avoid passing `null` parameters to most of the methods; for example, passing a `null` value to the `schema` parameter might result in a search of all database schemas—so if you know the name of your desired schema, then pass the actual schema value.

In general, developing performance-oriented JDBC applications is not easy. In every step of the solution, you must make sure that your application will not choke under heavy requirements. For performance reasons, you should avoid excessive metadata calls, because database metadata methods that generate `ResultSet` objects are relatively slow. JDBC applications may cache information returned from result sets that generate database metadata methods so that multiple executions are not needed. For this purpose you may use Java Caching System (JCS) from the Apache Software Foundation. JCS is a distributed caching system written in Java for server-side Java applications; for more information, see `http://jakarta.apache.org/turbine/jcs/`.

A Java class called `DatabaseMetaDataTool` (which is defined in the `jcb.meta` package) will be available for download from the Source Code section of the Apress website. It provides ready-to-use methods for answering the database metadata questions. For questions about database metadata, I list portions of these classes in some sections of this book, but you'll find the complete class definitions (including JavaDoc-style comments) at the Apress website.

All of the methods in this chapter are static, and each method is written to be as independent as possible. This is so you can cut and paste solutions from this book whenever possible. The methods will return the result as XML (serialized as a `String` object, which can be easily converted to an XML document such as `org.w3c.dom.Document` or `org.jdom.Document`). Also, I provide a utility class, `DocumentManager`, which can

- Convert `org.w3c.dom.Document` to XML as a serialized `String` object
- Convert `org.jdom.Document` to XML as a serialized `String` object
- Convert XML as a serialized `String` object into `org.w3c.dom.Document`
- Convert XML as a serialized `String` object into `org.jdom.Document`

In general, it is efficient to create XML as a serialized `String` object. The Java and JDBC solutions are grouped in a Java package called `jcb` (**J**DBC **C**ook**B**ook). Table 2-1 shows the structure of the package. All of the code will be available from the Source Code section of the Apress website.

Table 2-1. *JCB Package Structure*

Component	Description
`jcb`	Package name for **J**DBC **C**ook**B**ook
`jcb.meta`	Database and `ResultSet` metadata-related interfaces and classes
`jcb.db`	Database-related interfaces and classes
`jcb.util`	Utility classes
`servlets`	Java servlets
`client`	All client and test programs using the `jcb` package

When using the `DatabaseMetaData` object, you should observe two key facts:

- The MySQL database does not understand "schema"; you have to use "catalog."
- The Oracle database does not understand "catalog"; you have to use "schema."

2.1. What Is Metadata?

Metadata is data about data (or information about information), which provides structured, descriptive information about other data. According to Wikipedia (`http://en.wikipedia.org/wiki/Metadata`):

> *Metadata (Greek: meta-+ Latin: data "information"), literally "data about data", is information that describes another set of data. A common example is a library catalog card, which contains data about the contents and location of a book: It is data about the data in the book referred to by the card. Other common contents of metadata include the source or author of the described dataset, how it should be accessed, and its limitations.*

The following quote from the NOAA Coastal Services Center, or CSC (`http://www.csc.noaa.gov/metadata/text/whatismet.htm`), illustrates the importance of the concept of metadata:

> *Imagine trying to find a book in a library without the help of a card catalog or computerized search interface. Could you do it? Perhaps, but it would be difficult at best. The information contained in such a system is essentially metadata about the books that are housed at that library or at other libraries. It provides you with vital information to help you find a particular book and aids you in making a decision as to whether that book might fit your needs. Metadata serves a similar purpose for geospatial data.*

The NOAA CSC further adds that "metadata is a component of data which describes the data. It is 'data about data.'" Metadata describes the content, quality, condition, and other characteristics of data. Metadata describes the who, what, when, where, why, and how of a data set. Without proper documentation, a data set is incomplete.

KTWEB (`http://www.ktweb.org/rgloss.cfm`) defines metadata as "data about data, or information about information; in practice, metadata comprises a structured set of descriptive elements to describe an information resource or, more generally, any definable entity."

Relational databases (such as MySQL and Oracle) use tables and other means (such as operating system file systems) to store their own data and metadata. Each relational database has its own proprietary methods for storing metadata. Examples of relational database metadata include

- A list of all the tables in the database, including their names, sizes, and the number of rows
- A list of the columns in each database, and what tables they are used in, as well as the type of data stored in each column

For example, the Oracle database keeps metadata in several tables (I have listed two here):

- `ALL_TABLES`: A list of all tables in the current database
- `ALL_TAB_COLS`: A list of all columns in the database

Imagine, at runtime, trying to execute a SQL query in a relational database without knowing the name of tables, columns, or views. Could you do it? Of course not. Metadata helps you to find out what is available in the database and then, with the help of that information (called metadata), you can build proper SQL queries at runtime. Also, having access to structured database metadata relieves a JDBC programmer of having to know the characteristics of relational databases in advance.

Metadata describes the data but is not the actual data itself. For example, the records in a card catalog in a local library give brief details about the actual book. The card catalog—as metadata—provides enough information to tell you what the book is called, its unique identification number, and how and where you can find it. These details are metadata—in this case, bibliographic elements such as author, title, abstract, publisher, and published date.

In a nutshell, database metadata enables *dynamic database access*. Typically, most JDBC programmers know their target database's schema definitions: the names of tables, views, columns, and their associated types. In this case, the JDBC programmer can use the strongly typed JDBC interfaces. However, there is another important class of database access where an application (or an application builder) dynamically (in other words, at runtime) discovers the database schema information and uses that information to perform appropriate dynamic data access. This chapter describes the JDBC support for dynamic access. A dynamic database access application may include building dynamic queries, dynamic browsers, and GUI database adapters, just to mention a few.

For further research on metadata, refer to the following websites:

- "Metadata: An Overview": `http://www.nla.gov.au/nla/staffpaper/cathro3.html`
- "Introduction to Metadata": `http://www.getty.edu/research/conducting_research/standards/intrometadata/index.html`
- USGS CMG "Formal Metadata" Definition: `http://walrus.wr.usgs.gov/infobank/programs/html/definition/fmeta.html`

2.2. What Is Database Metadata?

The database has emerged as a major business tool across all enterprises, and the concept of *database metadata* has become a crucial topic. Metadata, which can be broadly defined as "data about data," refers to the searchable definitions used to locate information. On the other hand, *database* metadata, which can be broadly defined as "data about database data," refers to the searchable definitions used to locate database metadata (such as a list of all the tables for a specific schema). For example, you may use database metadata to generate web-based applications (see `http://dev2dev.bea.com/pub/a/2004/06/GenApps_hussey.html`). Or, you may use database metadata to reverse-engineer the whole database and dynamically build your desired SQL queries.

JDBC allows clients to discover a large amount of metadata information about a database (including tables, views, columns, stored procedures, and so on) and any given `ResultSet` via metadata classes.

Most of JDBC's metadata consists of information about one of two things:

- `java.sql.DatabaseMetaData` (database metadata information)
- `java.sql.ResultSetMetaData` (metadata information about a `ResultSet` object)

You should use `DatabaseMetaData` to find information about your database, such as its capabilities and structure, and use `ResultSetMetaData` to find information about the results of a SQL query, such as size and types of columns.

JDBC provides the following important interfaces that deal with database and result set metadata:

- `java.sql.DatabaseMetaData`: Provides comprehensive information about the database as a whole: table names, table indexes, database product name and version, and actions the database supports. Most of the solutions in this chapter are extracted from our solution class `DatabaseMetaDataTool` (you can download this class from the Source Code section of the Apress website). The `DatabaseMetaData` interface includes a number of methods that can be used to determine which SQL features are supported by a particular database. According to the JDBC specification, "The `java.sql.DatabaseMetaData` interface provides methods for retrieving various metadata associated with a database. This includes enumerating the stored procedures in the database, the tables in the database, the schemas in the database, the valid table types, the valid catalogs, finding information on the columns in tables, access rights on columns, access rights on tables, minimal row identification, and so on." Therefore, `DatabaseMetaData` methods can be categorized as
  - The schemas, catalogs, tables, views, columns, and column types
  - The database, users, drivers, stored procedures, and functions
  - The database limits (upper and lower bounds, minimums and maximums)
  - The features supported (and those not supported) by the database
- `java.sql.ResultSetMetaData`: Gets information about the types and properties of the columns in a `ResultSet` object. This interface is discussed in Chapter 4.
- `java.sql.ParameterMetaData`: Gets information about the types and properties of the parameters in a `PreparedStatement` object. `ParameterMetaData`, introduced in JDBC 3.0, retrieves information such as the number of parameters in the `PreparedStatement`, the type of data that can be assigned to the parameter, and whether or not the parameter value can be set to `null`. This interface is discussed in Chapter 5.
- `javax.sql.RowSetMetaData`: Extends the `ResultSetMetaData`, an object that contains information about the columns in a `RowSet` object. This interface is an extension of the `ResultSetMetaData` interface and has methods for setting the values in a `RowSetMetaData` object. When a `RowSetReader` object reads data into a `RowSet` object, it creates a `RowSetMetaData` object and initializes it using the methods in the `RowSetMetaData` interface. Then the reader passes the `RowSetMetaData` object to the rowset. The methods in this interface are invoked internally when an application calls the method `RowSet.execute()`; an application programmer would not use them directly.

2.3. How Do You Discover Database Metadata?

To discover metadata information about a database, you must create a `DatabaseMetaData` object. Note that some database vendors might not implement the `DatabaseMetaData` interface. Once a client program has obtained a valid database connection, the following code can get a database metadata object:

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import jcb.util.DatabaseUtil;
…
Connection conn = null;
DatabaseMetaData dbMetaData = null;
try {
   // Get a valid database connection
   conn = getConnection();
   // Create a database metadata object as DatabaseMetaData
   dbMetaData = conn.getMetaData();
   if (dbMetaData == null) {
      // Database metadata is not supported
      // therefore, you cannot get metadata at runtime
   }
   else {
      // Then we are in business and can invoke
      // over 100 methods defined in DatabaseMetaData

      // Check to see if transactions are supported
      if (dbMetaData.supportsTransactions()) {
         // Transactions are supported
      }
      else {
         // Transactions are not supported
      }
   }
}
catch(SQLException e) {
   // deal and handle the exception
   e.printStackTrace();

   // other things to handle the exception
}
finally {
   // close resources
   DatabaseUtil.close(conn);
}
```

You can use a `dbMetaData` object to invoke over 100 methods that are defined in the `DatabaseMetaData` interface. Therefore, to do something useful with `DatabaseMetaData`, you

must get a valid `Connection` object of type `java.sql.Connection`. The `DatabaseMetaData` object provides information about the entire database, such as the names of database tables or the names of a table's columns. Since various databases support different variants of SQL, there are also a large number of methods querying the database about what SQL methods it supports. Table 2-2 offers a partial listing of these methods.

Table 2-2. *Some DatabaseMetaData Methods*

Method Name	Description
`getCatalogs()`	Returns a list of catalogs of information (as a `ResultSet` object) in that database. With the JDBC-ODBC bridge driver, you get a list of databases registered with ODBC. According to JDBC, a database may have a set of catalogs, and each catalog may have a set of schemas. The terms *catalog* and *schema* can have different meanings depending on the database vendor. In general, the DBMS maintains a set of tables containing information about most of the objects in the database. These tables and views are collectively known as the *catalog*. The catalog tables contain metadata about objects such as tables, views, indexes, stored procedures, triggers, and constraints. To do anything (read, write, update) with these catalog tables and views, you need a special privilege. It is the DBMS's responsibility to ensure that the catalog contains accurate descriptions of the metadata objects in the database at all times. Oracle treats *schema* as a database name, while MySQL treats *catalog* as a database name. So, to get the name of databases from Oracle, you must use `getSchemas()`; to get the name of databases from MySQL, you must use `getCatalogs()`.
`getSchemas()`	Retrieves the schema names (as a `ResultSet` object) available in this database. Typically, a schema is a set of named objects. Schemas provide a logical classification of database objects (tables, views, aliases, stored procedures, user-defined types, and triggers) in an RDBMS.
`getTables(catalog, schema, tableNames, columnNames)`	Returns table names for all tables matching `tableNames` and all columns matching `columnNames`.
`getColumns(catalog, schema, tableNames, columnNames)`	Returns table column names for all tables matching `tableNames` and all columns matching `columnNames`.
`getURL()`	Gets the name of the URL you are connected to.
`getPrimaryKeys(catalog, schema, tableName)`	Retrieves a description of the given table's primary key columns.
`getDriverName()`	Gets the name of the database driver you are connected to.

2.4. What Is JDBC's Answer to Database Metadata?

JDBC provides a low-level interface called `DatabaseMetaData`. This chapter explains how to dissect the `DatabaseMetaData` object in order to find out the table names, column names or types, stored procedures names and signatures, and other useful information. Before delving into the solution, let's take a look at the relationships (see Figure 2-1) between important low-level

interfaces and classes. There are several ways that you can create a `Connection` object. Once you have a valid `Connection` object, then you can create a `DatabaseMetaData` object.

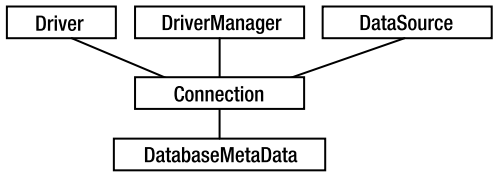

Figure 2-1. *Relationships between important interfaces and classes*

According to JDK 1.5, the `DatabaseMetaData` allows you to obtain information about the database and has over one hundred methods. You can find a description of `DatabaseMetaData` at `http://java.sun.com/j2se/1.5.0/docs/api/java/sql/DatabaseMetaData.html`.

To obtain a `DatabaseMetaData` object, use these general steps:

1. Connect to a database by using an instance of the `Connection` object.
2. To find out the names of the database schema, tables, and columns, get an instance of the `DatabaseMetaData` object from the `Connection`.
3. Perform the actual query by issuing a SQL query string. Then use the `Connection` to create a `Statement` class to represent your query.
4. The query returns a `ResultSet`. To find out the names of the column rows in that `ResultSet`, obtain an instance of the `ResultSetMetaData` class.

To get a `DatabaseMetaData`, use the following snippet:

```
Connection conn = null;
DatabaseMetaData dbMetaData = null;
try {
   // Get a valid database connection
   conn = getConnection();      // Get an instance of a DatabaseMetaData object
   dbMetaData = conn.getMetaData();
   if (dbMetaData == null) {
      // Database metadata is NOT supported
   }
   else {
      // Database metadata is supported and you can invoke
      // over 100 methods defined in DatabaseMetaData

      // Now that we have a valid database metadata (DatabaseMetaData) object
      // it can be used to do something useful:

      // Retrieves whether this database supports using columns not included in
      // the SELECT statement in a GROUP BY clause provided that all of the
      // columns in the SELECT statement are included in the GROUP BY clause.
      System.out.println(dbMetaData.supportsGroupByBeyondSelect());
```

```
            // Retrieves whether this database supports using a column that is not in
            // the SELECT statement in a GROUP BY clause.
            System.out.println(dbMetaData.supportsGroupByUnrelated());
            ...
        }
    }
    catch(SQLException e) {
        // deal and handle the SQLException
        ...
    }
    catch(Exception e2) {
        // deal with other exceptions
        ...
    }
```

2.5. What Is the Vendor Name Factor in Database Metadata?

Sometimes, for a given problem, there are different solutions based on the database vendor. For example, the code that gets the table names for an Oracle database is different from the code that gets the tables names for a MySQL database. When you develop an application or framework for a relational database, be sure that your connection pool manager takes the vendor name as a parameter. Depending on the vendor name, you might be calling different methods, or you might be issuing a different set of SQL queries. For example, when you're using the `BLOB` data type, the vendor name makes a difference in reading or writing `BLOB` data. For instance, Oracle requires an `empty_blob()` function use for setting empty `BLOB`s, but MySQL does not (empty `BLOB`s are denoted by `NULL` in MySQL).

The vendor name also plays an important role in connection pool management and database metadata. Suppose you have a pool of connections that you use in a production environment. If for some reason the database server goes down, then all of the connections in the pool will be obsolete or defunct—that is, they become dead connections. Using a dead connection will throw an exception. One of the important tasks a pool manager must do is that, before handing a connection to the client, it must make sure that the connection is valid. For this reason, a pool manager must issue a *minimal SQL query* against that database to make sure that the connection is valid. If it is valid, then it can be given to a client; otherwise, you need to obtain another available connection or throw an exception. This minimal SQL query is called a validity check statement, which can differ from vendor to vendor. A validity check statement is a SQL statement that will return at least one row. For Oracle, this validity check statement is `"select 1 from dual"` and for MySQL and Sybase Adaptive Server, it is `"select 1"`. Without knowing the `vendor` parameter, it is impossible to check for the validity of database connections. Also, note that without a valid database connection, you cannot get a `DatabaseMetaData` object. Therefore, you have to make sure that you have a valid database connection before attempting to create a `DatabaseMetaData` object.

Some JDBC metadata methods require knowledge of the database vendor. For example, getting the name of database tables is not the same in every case. For an Oracle database, you need to select the names from Oracle's `user_objects` table, while for other databases, the `DatabaseMetaData.getTables()` method will be sufficient.

Therefore, when you write a Java or JDBC application program or framework, you have to keep in mind that the same program will run against many relational databases (MySQL, Oracle, Sybase, and others). For example, if your application or framework runs on Oracle, you should be able to run the same program, with minimal changes to parameters and configurations, using MySQL. This means that you need to create database-dependent parameters (such as `vendor code`—specifying the vendor of the database) for your database URLs, SQL queries, and connection properties; avoid hard-coding any values that depend on a specific database vendor.

Here is an example of a vendor name in a configuration file. Based on the `<db-name>.vendor` key, you will be able to make smart decisions.

```
db.list=db1,db2,db3

db1.vendor=mysql
db1.url=jdbc:mysql://localhost/octopus
db1.driver=org.gjt.mm.mysql.Driver
db1.username=root
db1.password=mysql
db1.<…>=…

db2.vendor=oracle
db2.url=jdbc:oracle:thin:@localhost:1521:kitty
db2.driver=oracle.jdbc.driver.OracleDriver
db2.username=scott
db2.password=tiger
db2.<…>=…

db3.vendor=hsql
db3.url=jdbc:hsqldb:/members/alex/vrc/vrcdb
db3.driver=org.hsqldb.jdbcDriver
db3.username=alexis
db3.password=mypassword
db3.<…>=…
```

Here is another example of a vendor name in an XML document. In this example, the `<db-name>.vendor` key helps you make smart decisions.

```
<?xml version='1.0'>
<databases>
   <database id="db1">
      <vendor>mysql</vendor>
      <url>jdbc:mysql://localhost/octopus</url>
      <driver>org.gjt.mm.mysql.Driver</driver>
      <username>root</username>
      <password>mysql</password>
      ...
   </database>
```

```
   <database id="db2">
      <vendor>oracle</vendor>
      <url>jdbc:oracle:thin:@localhost:1521:kitty</url>
      <driver>oracle.jdbc.driver.OracleDriver</driver>
      <username>scott</username>
      <password>tiger</password>
      ...
   </database>
   <database id="db3">
      <vendor>hsql</vendor>
      <url>jdbc:hsqldb:/members/alex/vrc/vrcdb</url>
      <driver>org.hsqldb.jdbcDriver</driver>
      <username>alexis</username>
      <password>mypassword</password>
      ...
   </database>
</databases>
```

Now, using this configuration file, depending on the name of the vendor, you may select the appropriate database connection's validity check statement. Also, based on the name of the vendor, you might issue different JDBC methods for getting the database's table or view names.

2.6. How Do You Find JDBC's Driver Information?

To find out a database's vendor name and version information, you can invoke the following four methods from the `DatabaseMetaData` interface:

- `int getDatabaseMajorVersion()`: Retrieves the major version number of the underlying database. (Note that in `oracle.jdbc.OracleDatabaseMetaData`, this method is not supported. Therefore, use a `try-catch` block in code. If the method returns a `SQLException`, we return the message "unsupported feature" in the XML result.)
- `int getDatabaseMinorVersion()`: Retrieves the minor version number of the underlying database.
- `String getDatabaseProductName()`: Retrieves the name of this database product.
- `String getDatabaseProductVersion()`: Retrieves the version number of this database product.

Our solution combines these methods into a single method and returns the result as an XML `String` object, which any client can use. The result has the following syntax:

```
<?xml version='1.0'>
<DatabaseInformation>
    <majorVersion>database-major-version</majorVersion>
    <minorVersion>database-minor-version</minorVersion>
    <productName>database-product-name</productName>
    <productVersion>database-product-version</productVersion>
</DatabaseInformation>
```

The Solution

The solution is generic and can support MySQL, Oracle, and other relational databases. Note that the `getDatabaseMajorVersion()` method (implemented by the `oracle.jdbc.Oracle➥DatabaseMetaData` class) is an unsupported feature; therefore, we have to use a `try-catch` block. If the method returns a `SQLException`, we return the message "unsupported feature" in the XML result.

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
…
/**
 * Get database product name and version information.
 * This method calls 4 methods (getDatabaseMajorVersion(),
 * getDatabaseMinorVersion(), getDatabaseProductName(),
 * getDatabaseProductVersion()) to get the required information
 * and it represents the information as XML.
 *
 * @param conn the Connection object
 * @return database product name and version information
 * as an XML document (represented as a String object).
 */
public static String getDatabaseInformation(Connection conn)
    throws Exception {
    try {
        DatabaseMetaData meta = conn.getMetaData();
        if (meta == null) {
           return null;
        }

        StringBuffer sb = new StringBuffer("<?xml version='1.0'>");
        sb.append("<DatabaseInformation>");

        // Oracle (and some other vendors) do not
        // support some of the following methods
        // (such as getDatabaseMajorVersion() and
        // getDatabaseMajorVersion()); therefore,
        // we need to use a try-catch block.
        try {
           int majorVersion = meta.getDatabaseMajorVersion();
           appendXMLTag(sb, "majorVersion", majorVersion);
        }
        catch(Exception e) {
           appendXMLTag(sb, "majorVersion", "unsupported feature");
        }
```

```
        try {
           int minorVersion = meta.getDatabaseMinorVersion();
           appendXMLTag(sb, "minorVersion", minorVersion);
        }
        catch(Exception e) {
           appendXMLTag(sb, "minorVersion", "unsupported feature");
        }

        String productName = meta.getDatabaseProductName();
        String productVersion = meta.getDatabaseProductVersion();
        appendXMLTag(sb, "productName", productName);
        appendXMLTag(sb, "productVersion", productVersion);
        sb.append("</DatabaseInformation>");
        return sb.toString();
    }
    catch(Exception e) {
        e.printStackTrace();
        throw new Exception("could not get the database information:"+
                            e.toString());
    }
}
```

Client: Program for Getting Database Information

```
import java.util.*;
import java.io.*;
import java.sql.*;

import jcb.util.DatabaseUtil;
import jcb.meta.DatabaseMetaDataTool;
import jcb.db.VeryBasicConnectionManager;

public class TestDatabaseMetaDataTool_DatabaseInformation {

    public static void main(String[] args) {
        String dbVendor = args[0];  // { "mysql", "oracle" }
        Connection conn = null;
        try {
            conn = VeryBasicConnectionManager.getConnection(dbVendor);
            System.out.println("--- getDatabaseInformation ---");
            System.out.println("conn="+conn);
            String dbInfo = DatabaseMetaDataTool.getDatabaseInformation(conn);
            System.out.println(dbInfo);
            System.out.println("-----------------------------------");
        }
```

```
        catch(Exception e){
            e.printStackTrace();
            System.exit(1);
        }
        finally {
            DatabaseUtil.close(conn);
        }
    }
}
```

Running the Solution for a MySQL Database

```
$ javac TestDatabaseMetaDataTool_DatabaseInformation.java
$ java TestDatabaseMetaDataTool_DatabaseInformation mysql
--- getDatabaseInformation ---
conn=com.mysql.jdbc.Connection@1837697
<?xml version='1.0'>
<DatabaseInformation>
    <majorVersion>4</majorVersion>
    <minorVersion>0</minorVersion>
    <productName>MySQL</productName>
    <productVersion>4.0.4-beta-max-nt</productVersion>
</DatabaseInformation>
-----------------------------------
```

Running the Solution for an Oracle Database

The following output is formatted to fit the page:

```
$ javac TestDatabaseMetaDataTool_DatabaseInformation.java
$ java TestDatabaseMetaDataTool_DatabaseInformation oracle
--- getDatabaseInformation ---
conn= oracle.jdbc.driver.OracleConnection@169ca65
<?xml version='1.0'>
<DatabaseInformation>
     <majorVersion>unsupported feature</majorVersion>
     <minorVersion>unsupported feature</minorVersion>
     <productName>Oracle</productName>
     <productVersion>
         Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production
         With the Partitioning, OLAP and Oracle Data Mining options
         JServer Release 9.2.0.1.0 - Production
     </productVersion>
</DatabaseInformation>
-----------------------------------
```

2.7. What Are a Database's SQL Keywords?

In GUI database applications, if you are forming a SQL query at runtime, then you might need the database's SQL keywords. `DatabaseMetaData` provides the `getSQLKeywords()` method for getting the database's SQL keywords. This method retrieves a comma-separated list of all of the database's SQL keywords but excludes SQL-92 keywords. The return type of this method is not very useful, because once the client receives the result as a `String`, it has to be tokenized to find the actual keywords. Our solution returns this as a `java.util.List`, where each element will be a keyword. You can try the following two solutions for this problem. The first solution returns the list of SQL keywords as a `java.util.List` object, where each element is a SQL keyword (as a `String` object). The second solution returns the result as XML.

Solution 1: Returns the SQL Keywords as a List

```
import java.util.List;
import java.util.ArrayList;
import java.sql.*;

/**
 * Get the SQL keywords for a database.
 * @param conn the java.sql.Connection object.
 * @return the list of SQL keywords for a database.
 * Each element in the list is a SQL keyword.
 * @exception Failed to get the SQL keywords for a database.
 */
 public static List getSQLKeywords(Connection conn)
     throws Exception {
     DatabaseMetaData meta = conn.getMetaData();
     if (meta == null) {
         return null;
     }

     String sqlKeywords = meta.getSQLKeywords();
     if ((sqlKeywords == null) || (sqlKeywords.length() == 0)) {
         return null;
     }

     List list = new ArrayList();
     // SQL keywords are separated by ","
     StringTokenizer st = new StringTokenizer(sqlKeywords, ",");
     while(st.hasMoreTokens()) {
         list.add(st.nextToken().trim());
     }
     return list;
}
```

MySQL Client Code Using getSQLKeywords()

This code is the same for the Oracle client:

```
//
// Get list of SQL Keywords as a java.util.List
// print the list of SQL Keywords
//
List list = DatabaseMetaDataTool.getSQLKeywords(conn);
System.out.println("--- Got results: list of SQL Keywords ---");
for (int i=0; i < list.size(); i++) {
    String sqlKeyword = (String) list.get(i);
    System.out.println("sqlKeyword= " + sqlKeyword);
}
```

Output: MySQL Database

```
--- Got results: list of SQL Keywords ---
sqlKeyword= AUTO_INCREMENT
sqlKeyword= BINARY
sqlKeyword= BLOB
sqlKeyword= ENUM
sqlKeyword= INFILE
sqlKeyword= LOAD
sqlKeyword= MEDIUMINT
sqlKeyword= OPTION
sqlKeyword= OUTFILE
sqlKeyword= REPLACE
sqlKeyword= SET
sqlKeyword= TEXT
sqlKeyword= UNSIGNED
sqlKeyword= ZEROFILL
```

Output: Oracle Database

```
--- Got results: list of SQL Keywords ---
sqlKeyword= ACCESS
sqlKeyword= ADD
sqlKeyword= ALTER
sqlKeyword= AUDIT
sqlKeyword= CLUSTER
sqlKeyword= COLUMN
sqlKeyword= COMMENT
sqlKeyword= COMPRESS
sqlKeyword= CONNECT
sqlKeyword= DATE
sqlKeyword= DROP
sqlKeyword= EXCLUSIVE
```

```
sqlKeyword= FILE
sqlKeyword= IDENTIFIED
sqlKeyword= IMMEDIATE
sqlKeyword= INCREMENT
sqlKeyword= INDEX
sqlKeyword= INITIAL
sqlKeyword= INTERSECT
sqlKeyword= LEVEL
sqlKeyword= LOCK
sqlKeyword= LONG
sqlKeyword= MAXEXTENTS
sqlKeyword= MINUS
sqlKeyword= MODE
sqlKeyword= NOAUDIT
sqlKeyword= NOCOMPRESS
sqlKeyword= NOWAIT
sqlKeyword= NUMBER
sqlKeyword= OFFLINE
sqlKeyword= ONLINE
sqlKeyword= PCTFREE
sqlKeyword= PRIOR
sqlKeyword= all_PL_SQL_reserved_ words
```

Solution 2: Returns the SQL Keywords as XML

```
import java.sql.*;
import java.util.*;

/**
 * Get the SQL Keywords for a database.
 * @param conn the Connection object.
 * @return the list of SQL keywords for a database as XML.
 * @exception Failed to get the SQL keywords for a database.
 */
 public static String getSQLKeywordsAsXML(Connection conn)
    throws Exception {
     DatabaseMetaData meta = conn.getMetaData();
     if (meta == null) {
         return null;
     }

     String sqlKeywords = meta.getSQLKeywords();
     if ((sqlKeywords == null) || (sqlKeywords.length() == 0)) {
         return null;
     }
```

```
        StringBuffer sb = new StringBuffer("<?xml version='1.0'>");
        sb.append("<sql_keywords>");

        // SQL keywords are separated by ","
        StringTokenizer st = new StringTokenizer(sqlKeywords, ",");
        while(st.hasMoreTokens()) {
           sb.append("<keyword>");
           sb.append(st.nextToken().trim());
           sb.append("</keyword>");
        }
        sb.append("</sql_keywords>");
        return sb.toString();
    }
```

Client Code: MySQL Database

This code is the same for the Oracle client:

```
//
// Get list of SQL keywords as XML
// print the list of SQL keywords
//
String listOfSQLKeywords = DatabaseMetaDataTool.getSQLKeywordsAsXML(conn);
System.out.println("--- Got results: list of SQL Keywords ---");
System.out.println("listOfSQLKeywords= " + listOfSQLKeywords);
```

Output: MySQL Database

```
<?xml version='1.0'>
<sql_keywords>
    <keyword>AUTO_INCREMENT</keyword>
    <keyword>BINARY</keyword>
    <keyword>BLOB</keyword>
    <keyword>ENUM</keyword>
    <keyword>INFILE</keyword>
    <keyword>LOAD</keyword>
    <keyword>MEDIUMINT</keyword>
    <keyword>OPTION</keyword>
    <keyword>OUTFILE</keyword>
    <keyword>REPLACE</keyword>
    <keyword>SET</keyword>
    <keyword>TEXT</keyword>
    <keyword>UNSIGNED</keyword>
    <keyword>ZEROFILL</keyword>
</sql_keywords>
```

Output: Oracle Database

```
<?xml version='1.0'>
<sql_keywords>
    <keyword>ACCESS</keyword>
    <keyword>ADD</keyword>
    <keyword>ALTER</keyword>
    <keyword>AUDIT</keyword>
    <keyword>CLUSTER</keyword>
    <keyword>COLUMN</keyword>
    <keyword>COMMENT</keyword>
    <keyword>COMPRESS</keyword>
    <keyword>CONNECT</keyword>
    <keyword>DATE</keyword>
    <keyword>DROP</keyword>
    <keyword>EXCLUSIVE</keyword>
    <keyword>FILE</keyword>
    <keyword>IDENTIFIED</keyword>
    <keyword>IMMEDIATE</keyword>
    <keyword>INCREMENT</keyword>
    <keyword>INDEX</keyword>
    <keyword>INITIAL</keyword>
    <keyword>INTERSECT</keyword>
    <keyword>LEVEL</keyword>
    <keyword>LOCK</keyword>
    <keyword>LONG</keyword>
    <keyword>MAXEXTENTS</keyword>
    <keyword>MINUS</keyword>
    <keyword>MODE</keyword>
    <keyword>NOAUDIT</keyword>
    <keyword>NOCOMPRESS</keyword>
    <keyword>NOWAIT</keyword>
    <keyword>NUMBER</keyword>
    <keyword>OFFLINE</keyword>
    <keyword>ONLINE</keyword>
    <keyword>PCTFREE</keyword>
    <keyword>PRIOR</keyword>
    <keyword>all_PL_SQL_reserved_ words</keyword>
<sql_keywords>
```

2.8. What Are the Available SQL Data Types?

If you want to provide table-creation services from GUI database applications, then you may need to provide available SQL data types for databases. The `DatabaseMetaData` interface contains methods that list the available SQL types used by a database. The method `getAvailableSqlTypes()` retrieves the SQL data types supported by a database and driver. How do you get the name of a JDBC type? The following method implements a convenient

way to convert a java.sql.Types integer value into a printable name. This method, which is useful for debugging purposes, uses reflection to get all the field names from java.sql.Types. It then retrieves their values and creates a map of values to names.

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
…
/**
 * Listing of available SQL types used by a database. This method
 * retrieves the SQL data types supported by a database and driver.
 *
 *
 * @param conn the Connection object
 * @return an XML (as a String object).
 * @exception Failed to get the available SQL types used by a database.
 */
 public static String getAvailableSqlTypes(Connection conn)
    throws Exception {
    ResultSet rs = null;
    try {
        // Get database metadata
        DatabaseMetaData meta = conn.getMetaData();
        if (meta == null) {
             return null;
        }

        // Get type infornmation
        rs = meta.getTypeInfo();

        // Retrieve type info from the result set
        StringBuffer sb = new StringBuffer();
        sb.append("<sqlTypes>");
        while (rs.next()) {
            // Get the database-specific type name
            String typeName = rs.getString("TYPE_NAME");

            // Get the java.sql.Types type to which this
            // database-specific type is mapped
            short dataType = rs.getShort("DATA_TYPE");

            // Get the name of the java.sql.Types value.
            String jdbcTypeName = getJdbcTypeName(dataType);
```

```
            sb.append("<typeName>");
            sb.append(typeName);
            sb.append("</typeName>");
            sb.append("<dataType>");
            sb.append(dataType);
            sb.append("</dataType>");
            sb.append("<jdbcTypeName>");
            sb.append(jdbcTypeName);
            sb.append("</jdbcTypeName>");
        }
        sb.append("</sqlTypes>");
        return sb.toString();
    }
    finally {
        DatabaseUtil.close(rs);
    }
}

/**
 * Get the name of a JDBC type. This method implements a
 * convenient method for converting a java.sql.Types integer
 * value into a printable name. This method is useful for debugging.
 * The method uses reflection to get all the field names from
 * java.sql.Types. It then retrieves their values and creates a
 * map of values to names.
 * This method returns the name of a JDBC type.
 * Returns null if jdbcType is not recognized.
 *
 * @param jdbcType the JDBC type as an interger
 * @return the equivalent JDBC type name
 */
 public static String getJdbcTypeName(int jdbcType) {
    // Return the JDBC type name
    return (String) JDBC_TYPE_NAME_MAP.get(new Integer(jdbcType));
 }
```

The `JDBC_TYPE_NAME_MAP` table is defined as follows:

```
static final Map JDBC_TYPE_NAME_MAP = new HashMap();
static {
    // Get all fields in java.sql.Types
    Field[] fields = java.sql.Types.class.getFields();
    for (int i=0; i<fields.length; i++) {
        try {
            // Get field name
            String name = fields[i].getName();
```

```
            // Get field value
            Integer value = (Integer)fields[i].get(null);

            // Add to map
            JDBC_TYPE_NAME_MAP.put(value, name);
        }
        catch (IllegalAccessException e) {
            // ignore
        }
    }
}
```

Testing getAvailableSqlTypes(): Using a MySQL Database

```
//
// getAvailableSqlTypes
//
String  availableSqlTypes = DatabaseMetaDataTool.getAvailableSqlTypes(conn);
System.out.println("-------- MySQL availableSqlTypes -------------");
System.out.println(availableSqlTypes);
System.out.println("----------------------------------------");

-------- MySQL availableSqlTypes -------------
<sqlTypes>
    <type name="TINYINT" dataType="-6" jdbcTypeName="TINYINT"/>
    <type name="BIGINT" dataType="-5" jdbcTypeName="BIGINT"/>
    <type name="MEDIUMBLOB" dataType="-4" jdbcTypeName="LONGVARBINARY"/>
    <type name="LONGBLOB" dataType="-4" jdbcTypeName="LONGVARBINARY"/>
    <type name="BLOB" dataType="-4" jdbcTypeName="LONGVARBINARY"/>
    <type name="TINYBLOB" dataType="-3" jdbcTypeName="VARBINARY"/>
    <type name="CHAR" dataType="1" jdbcTypeName="CHAR"/>
    <type name="NUMERIC" dataType="2" jdbcTypeName="NUMERIC"/>
    <type name="DECIMAL" dataType="3" jdbcTypeName="DECIMAL"/>
    <type name="INT" dataType="4" jdbcTypeName="INTEGER"/>
    <type name="MEDIUMINT" dataType="4" jdbcTypeName="INTEGER"/>
    <type name="SMALLINT" dataType="5" jdbcTypeName="SMALLINT"/>
    <type name="FLOAT" dataType="6" jdbcTypeName="FLOAT"/>
    <type name="DOUBLE" dataType="8" jdbcTypeName="DOUBLE"/>
    <type name="DOUBLE PRECISION" dataType="8" jdbcTypeName="DOUBLE"/>
    <type name="REAL" dataType="8" jdbcTypeName="DOUBLE"/>
    <type name="VARCHAR" dataType="12" jdbcTypeName="VARCHAR"/>
    <type name="DATE" dataType="91" jdbcTypeName="DATE"/>
    <type name="TIME" dataType="92" jdbcTypeName="TIME"/>
    <type name="DATETIME" dataType="93" jdbcTypeName="TIMESTAMP"/>
    <type name="TIMESTAMP" dataType="93" jdbcTypeName="TIMESTAMP"/>
</sqlTypes>
----------------------------------------
```

Testing getAvailableSqlTypes(): Using an Oracle Database

```
//
// getAvailableSqlTypes
//
String  availableSqlTypes = DatabaseMetaDataTool.getAvailableSqlTypes(conn);
System.out.println("-------- Oracle availableSqlTypes ------------");
System.out.println(availableSqlTypes);
System.out.println("----------------------------------------");
Output:

-------- Oracle availableSqlTypes -------------
<sqlTypes>
   <type name="INTERVALDS" dataType="-104" jdbcTypeName="null"/>
   <type name="INTERVALYM" dataType="-103" jdbcTypeName="null"/>
   <type name="TIMESTAMP WITH LOCAL TIME ZONE" dataType="-102" jdbcTypeName="null"/>
   <type name="TIMESTAMP WITH TIME ZONE" dataType="-101" jdbcTypeName="null"/>
   <type name="NUMBER" dataType="-7" jdbcTypeName="BIT"/>
   <type name="NUMBER" dataType="-6" jdbcTypeName="TINYINT"/>
   <type name="NUMBER" dataType="-5" jdbcTypeName="BIGINT"/>
   <type name="LONG RAW" dataType="-4" jdbcTypeName="LONGVARBINARY"/>
   <type name="RAW" dataType="-3" jdbcTypeName="VARBINARY"/>
   <type name="LONG" dataType="-1" jdbcTypeName="LONGVARCHAR"/>
   <type name="CHAR" dataType="1" jdbcTypeName="CHAR"/>
   <type name="NUMBER" dataType="2" jdbcTypeName="NUMERIC"/>
   <type name="NUMBER" dataType="4" jdbcTypeName="INTEGER"/>
   <type name="NUMBER" dataType="5" jdbcTypeName="SMALLINT"/>
   <type name="FLOAT" dataType="6" jdbcTypeName="FLOAT"/>
   <type name="REAL" dataType="7" jdbcTypeName="REAL"/>
   <type name="VARCHAR2" dataType="12" jdbcTypeName="VARCHAR"/>
   <type name="DATE" dataType="91" jdbcTypeName="DATE"/>
   <type name="DATE" dataType="92" jdbcTypeName="TIME"/>
   <type name="TIMESTAMP" dataType="93" jdbcTypeName="TIMESTAMP"/>
   <type name="STRUCT" dataType="2002" jdbcTypeName="STRUCT"/>
   <type name="ARRAY" dataType="2003" jdbcTypeName="ARRAY"/>
   <type name="BLOB" dataType="2004" jdbcTypeName="BLOB"/>
   <type name="CLOB" dataType="2005" jdbcTypeName="CLOB"/>
   <type name="REF" dataType="2006" jdbcTypeName="REF"/>
</sqlTypes>
----------------------------------------
```

2.9. What Are Catalogs and Schemas?

If you want to provide catalog and schema services to database applications, then you might need to provide catalog and schema values to client applications. The words *catalog* and *schema* have different meanings, depending on the database vendor. Again, the `vendor` parameter is very important in understanding the semantics of catalogs and schemas. Oracle treats "schema" as a database name, while MySQL treats "catalog" as a database name. So, in order to get the name of

databases from Oracle, you must use `getSchemas()`; to get the name of databases from MySQL, you must use `getCatalogs()`. If you use `getCatalogs()` for an Oracle database, or `getSchemas()` for MySQL, it returns nothing (as `null` objects). In the JDBC API, `getSchemas()` claims that it returns a set of two columns ("table schema" and "table catalog"), but in reality it just returns "table schema" as a first column of the result set. Once again, this proves at least two points:

- You have to test your code against different databases; that is, databases can have different semantics by using the same JDBC API.
- When you define connections, make sure that the `vendor` parameter is defined. When you know the database vendor, you can invoke the correct methods.

getSchemas() Used for an Oracle Database

```
/**
 * Get Schemas(): Retrieves the schema names available
 * in this database. The results are ordered by schema name.
 *
 *
 * @param conn the Connection object.
 * @return an XML.
 * @exception Failed to get the Get Schemas.
 */
public static String getSchemas(java.sql.Connection conn)
    throws Exception {
    ResultSet schemas = null;
    StringBuffer sb = new StringBuffer();
    try {
         DatabaseMetaData meta = conn.getMetaData();
         if (meta == null) {
             return null;
         }

         schemas = meta.getSchemas();
         sb.append("<schemas>");
         while (schemas.next()) {
             String tableSchema = schemas.getString(1);    // "TABLE_SCHEM"
             //String tableCatalog = schemas.getString(2); // "TABLE_CATALOG"
             sb.append("<tableSchema>");
             sb.append(tableSchema);
             sb.append("</tableSchema>");
         }
         sb.append("</schemas>");
         return sb.toString();
     }
     catch(Exception e) {
         throw new Exception("Error: could not get schemas: "+e.toString());
     }
```

```
    finally {
        DatabaseUtil.close(schemas);
    }
}
```

getCatalogs() Used for a MySQL Database

```
/**
 * Get Catalogs: Retrieves the catalog names available in
 * this database.  The results are ordered by catalog name.
 *
 * @param conn the Connection object
 * @return an XML.
 * @exception Failed to get the Get Catalogs.
 */
public static String getCatalogs(java.sql.Connection conn)
    throws Exception {
    ResultSet catalogs = null;
    StringBuffer sb = new StringBuffer();
    try {
        DatabaseMetaData meta = conn.getMetaData();
        if (meta == null) {
            return null;
        }

        catalogs = meta.getCatalogs();
        sb.append("<catalogs>");
        while (catalogs.next()) {
            String catalog = catalogs.getString(1);  // "TABLE_CATALOG"
            sb.append("<catalog>");
            sb.append(catalog);
            sb.append("</catalog>");
        }
        sb.append("</catalogs>");
        return sb.toString();
    }
    catch(Exception e) {
        throw new Exception("Error: could not get catalogs: "+e.toString());
    }
    finally {
        DatabaseUtil.close(catalogs);
    }
}
```

2.10. What Are the Table Names for a Database?

If you are providing dynamic SQL queries for a GUI database application, then you might need the names of the user tables. In building database adapters and GUI database applications, the GUI developers often need the name of the tables. The following program provides such a solution. The solution can vary depending on the database vendor. For a complete solution, refer to the `DatabaseMetaDataTool` class, described under the `jcb.meta` package (you can download the entire package from the Source Code section of the Apress website). Here, I'll just list the portions of the programs that are most relevant to this topic.

The `DatabaseMetaData.getTables()` method returns the table names for a given database connection object. The `getTables()` method works well for MySQL, but it does not work well for Oracle databases (in addition to user's tables, it returns system tables, which are not needed by most of the client programs). To get a list of user-defined tables and views, I use the Oracle's metadata table called `user_objects`, which keeps track of objects (tables, views, ...) owned by the user. To get a list of user's tables for an Oracle database, you may use the following SQL query:

```
select object_name from user_objects
  where object_type = 'TABLE';
```

The `DatabaseMetaData.getTables()` has the following signature:

```
ResultSet getTables(String catalog,
                    String schemaPattern,
                    String tableNamePattern,
                    String[] types)  throws SQLException
```

This method retrieves a description of the tables available in the given catalog. Only table descriptions matching the catalog, schema, table name and type criteria are returned. The returned `ResultSet` object has 10 columns (for details, see JDK 1.5 documentation), which are ordered by `TABLE_TYPE`, `TABLE_SCHEM` and `TABLE_NAME` (column names for the returned `ResultSet` object). Here, for MySQL solution, I use the `getTables()` method. For better performance of this method and other metadata methods, it is highly recommended not to pass null/empty values as an actual parameters to these methods. Try to pass non-null and non-empty values to these metadata methods.

MySQL Solution

```
import java.util.*;
import java.io.*;
import java.sql.*;

/**
 * This class provides class-level methods for getting database
 * metadata.  This class provides wrapper methods for most of
 * the useful methods in DatabaseMetaData and whenever possible
 * it returns the result as an XML (serialized as a String object
 * -- for efficiency purposes) rather than a ResultSet object.
 * The reason for returning the result as XML is so it can be used
 * by all types of clients.
```

```
*
* The wrapper methods in this class are generic and have been
* tested with MySQL 4.0 and Oracle 9.2. These methods should run
* with other relational databases such as DB2, Sybase,
* and MS SQL Server 2000.
*
*/
public class DatabaseMetaDataTool {

   private static final String[] DB_TABLE_TYPES = { "TABLE" };
   private static final String[] DB_VIEW_TYPES = { "VIEW" };
   private static final String[] DB_MIXED_TYPES = { "TABLE", "VIEW" };

   private static final String COLUMN_NAME_TABLE_NAME = "TABLE_NAME";
   private static final String COLUMN_NAME_COLUMN_NAME = "COLUMN_NAME";
   private static final String COLUMN_NAME_DATA_TYPE = "DATA_TYPE";
   private static final String COLUMN_NAME_VIEW_NAME = "VIEW_NAME";
   private static final String COLUMN_NAME_TYPE_NAME = "TYPE_NAME";


   /**
    * Get the table names for a given connection object.
    * @param conn the Connection object
    * @return the list of table names as a List.
    * @exception Failed to get the table names from the database.
    */
    public static List getTableNames(Connection conn)
       throws Exception {
       ResultSet rs = null;
       try {
            DatabaseMetaData meta = conn.getMetaData();
            if (meta == null) {
                return null;
            }

            rs = meta.getTables(null, null, null, DB_TABLE_TYPES);
            if (rs == null) {
                return null;
            }

            List list = new ArrayList();
            while (rs.next()) {
                String tableName =
                   DatabaseUtil.getTrimmedString(rs, COLUMN_NAME_TABLE_NAME);
                System.out.println("getTableNames(): tableName="+tableName);
                if (tableName != null) {
                    list.add(tableName);
                }
            }
```

```
            return list;
        }
        catch(Exception e) {
            e.printStackTrace();
            throw e;
        }
        finally {
            DatabaseUtil.close(rs);
        }
    }

    // other methods ...
}
```

Testing the MySQL Solution: Client Program

```
    //
    // Print the list of tables
    //
    java.util.List tables = DatabaseMetaDataTool.getTableNames(conn);
    System.out.println("--- Got results: list of tables ---");
    for (int i=0; i < tables.size(); i++) {
        // process results one element at a time
        String tableName = (String) tables.get(i);
        System.out.println("table name = " + tableName);
    }
```

Testing the MySQL Solution: Output of the Client Program

```
--- Got results: list of tables ---
table name = artist
table name = artist_exhibit
...
table name = zdepts
table name = zemps
table name = zperson
table name = zz
table name = zzz
```

Oracle Solution

For Oracle databases, `DatabaseMetaData.getTables()` method does not work well (in addition to the user's tables, it returns system-level tables). To get user's tables, I use the following query:

```
    select object_name from user_objects
      where object_type = 'TABLE';
```

Oracle's user_object's table keeps track of user objects (tables, views, and other useful objects). As you can observe, again, the database vendor name plays an important role in fetching metadata (based on vendor name, you may apply different methods for solving a specific metadata problem).

```
import java.util.*;
import java.io.*;
import java.sql.*;

/**
 * This class provides class-level methods
 * for getting database metadata.
 *
 */
 public class DatabaseMetaDataTool {

    private static final String[] DB_TABLE_TYPES = { "TABLE" };
    private static final String[] DB_VIEW_TYPES = { "VIEW" };
    private static final String[] DB_MIXED_TYPES = { "TABLE", "VIEW" };

    private static final String COLUMN_NAME_TABLE_NAME = "TABLE_NAME";
    private static final String COLUMN_NAME_COLUMN_NAME = "COLUMN_NAME";
    private static final String COLUMN_NAME_DATA_TYPE = "DATA_TYPE";
    private static final String COLUMN_NAME_VIEW_NAME = "VIEW_NAME";
    private static final String COLUMN_NAME_TYPE_NAME = "TYPE_NAME";

    private static final String  ORACLE_VIEWS =
        "select object_name from user_objects where object_type = 'VIEW'";
    private static final String  ORACLE_TABLES =
        "select object_name from user_objects where object_type = 'TABLE'";
    private static final String  ORACLE_TABLES_AND_VIEWS =
        "select object_name from user_objects where object_type = 'TABLE' "+
        "or object_type = 'VIEW'";


    /**
     * Get the Oracle table names for a given connection object.
     * If you use getTableNames() for an Oracle database, you
     * will get lots of auxiliary tables, which belong to the user,
     * but the user is not interested in seeing them.
     *
     * @param conn the Connection object
     * @return the list of table names as a List.
     * @exception Failed to get the table names from the database.
     */
```

```
    public static java.util.List getOracleTableNames(java.sql.Connection conn)
        throws Exception {
        Statement stmt = null;
        ResultSet rs = null;
        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery(ORACLE_TABLES);

            if (rs == null) {
                return null;
            }

            java.util.List list = new java.util.ArrayList();
            while (rs.next()) {
                String tableName = DatabaseUtil.getTrimmedString(rs, 1);
                System.out.println("tableName="+tableName);
                if (tableName != null) {
                     list.add(tableName);
                }
            }

            return list;
        }
        catch (Exception e ) {
            e.printStackTrace();
            throw e;
        }
        finally {
            DatabaseUtil.close(rs);
            DatabaseUtil.close(stmt);
        }
    }

    // other methods ...
}
```

Testing the Oracle Solution: Client Program

```
    //
    // Print the list of tables
    //
    java.util.List tables = DatabaseMetaDataTool.getOracleTableNames(conn);
    System.out.println("Got results: list of tables --------------");
    for (int i=0; i < tables.size(); i++) {
        // process results one element at a time
        String tableName = (String) tables.get(i);
        System.out.println("table name = " + tableName);
    }
```

Output of the Client Program

```
Got results: list of tables --------------
table name = ALL_TYPES
table name = AUTHENTICATION_TYPES
table name = COMPANIES
...
table name = LOGS
table name = MYPAYROLLTABLE
table name = VIEW_CATEGORY_MEMBERS
table name = VIEW_CATEGORY_PERMISSIONS
table name = VIEW_DEFINITIONS
table name = ZDEPTS
table name = ZEMPS
```

To simplify this for clients (so that you don't have to call different methods to get the table names), we can introduce a wrapper object, which includes a `Connection` object and a vendor name:

```
public class ConnectionWrapper {

    private Connection conn = null;
    private String vendorName = null;

    public ConnectionWrapper() {
    }

    public ConnectionWrapper(Connection conn, String vendorName) {
        this.conn = conn;
        this.vendorName = vendorName;
    }

    public Connection getConnection() {
        return this.conn;
    }

    public void setConnection(Connection conn) {
        this.conn = conn;
    }

    public String getVendorName() {
        return this.vendorName;
    }

    public void setVendorName(String vendorName) {
        this.vendorName = vendorName;
    }
}
```

Now, we can get the table names from the following method:

```
/**
 * Get the table names for a given connection wrapper object.
 *
 * @param connWrapper the ConnectionWrapper object
 * @return the list of table names as a List.
 * @exception Failed to get the table names from the database.
 */
public static java.util.List getTableNames(ConnectionWrapper connWrapper)
    throws Exception {
    if (connWrapper == null) {
        return null;
    }

    if (connWrapper.getVendorName().equals("oracle")) {
        return getOracleTableNames(connWrapper.getConnection());
    }
    else {
        return getTableNames(connWrapper.getConnection());
    }
}
```

2.11. What Are the View Names for a Database?

A *view* is an alternative representation of data from one or more tables or views. A view can include all or some of the columns contained in one or more tables on which it is defined. A view is effectively a SQL query stored in the database catalog. Some database vendors, including Oracle, support the concept of views, while others, such as MySQL, do not at the present time (however, MySQL will support views starting with version 5.0). Therefore, a view is a virtual table and can be defined by SQL statements, but it may be vendor dependent. In general, views can be used for security purposes, such as hiding a salary field, or for the convenience of programmers or database administrators. Views enable the user to see only the information he or she needs at the moment, and provides security for the database managers.

Using `DatabaseMetaData.getTables(catalog, schemaPattern, tableNamePattern, types)` method, and by passing {"VIEW"} to the `types` parameter, you can get a list of views belonging to a database user. For Oracle databases, this method returns user-defined and system views. To get only user-created views, I use Oracle's metadata table, `user_objects`, which includes user-created tables and views. To get the views, use the following query:

```
select object_name from user_objects
  where object_type = 'VIEW';
```

To get the tables and views together, you can issue:

```
select object_name from user_objects
   where object_type = 'TABLE' or object_type = 'VIEW';
```

Consider the following table, which lists names of employees and their salaries:

```
create table MyPayrollTable(
    id varchar(9) not null primary key,
    name varchar(30) not null,
    salary int not null
);
```

If you want the employee names available but not their salaries, you can define the following view:

```
create view EmployeeNamesView (id, name)
    as select id, name from MyPayrollTable;
```

Here is how the `EmployeeNamesView` is created in the Oracle 9i SQL*Plus program:

```
$ sqlplus octopus/octopus
SQL*Plus: Release 9.2.0.1.0 - Production on Mon Dec 2 14:08:29 2002
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production

SQL> create table MyPayrollTable(
  2     id varchar(9) not null primary key,
  3     name varchar(30) not null,
  4     salary int not null
  5  );
Table created.
SQL> create view MyView2 (id, name)
  2     as select id, name from MyPayrollTable;
View created.
SQL> describe MyView2;
 Name                  Null?     Type
 --------------------  --------  ------------
 ID                    NOT NULL  VARCHAR2(9)
 NAME                  NOT NULL  VARCHAR2(30)
SQL> select object_name from user_objects where object_type='VIEW';
OBJECT_NAME
-----------
MYVIEW2
```

Oracle Solution: getOracleViewNames()

Next, let's look at the JDBC solution for finding the views for a given database. Here is the solution for Oracle:

```
/**
 * Get the Oracle view names for a given connection object.
 * If you use the getViewNames() for an Oracle database, you
 * will get lots of auxiliary views, which belong to the user,
 * but the user is not interested in seeing them.
 *
```

```
 * @param conn the Connection object
 * @return the list of view names as a List.
 * @exception Failed to get the view names from the database.
 */
public static java.util.List getOracleViewNames(java.sql.Connection conn)
    throws Exception {

    Statement stmt = null;
    ResultSet rs = null;
    try {
        stmt = conn.createStatement();
        // private static final String  ORACLE_VIEWS =
        // "select object_name from user_objects where object_type = 'VIEW'";
        rs = stmt.executeQuery(ORACLE_VIEWS);

        if (rs == null) {
            return null;
        }

        java.util.List list = new java.util.ArrayList();
        while (rs.next()) {
            String viewName = DatabaseUtil.getTrimmedString(rs, 1);
            System.out.println("viewName="+viewName);
            if (viewName != null) {
                 list.add(viewName);
            }
        }

        return list;
    }
    finally {
        DatabaseUtil.close(rs);
        DatabaseUtil.close(stmt);
    }
}
```

Testing getOracleViewNames()

```
//
// print the list of views
//
java.util.List views = DatabaseMetaDataTool.getOracleViewNames(conn);
System.out.println("Got results: list of views -----------");
if (views != null) {
   for (int i=0; i < views.size(); i++) {
       // process results one element at a time
       String viewName = (String) views.get(i);
       System.out.println("view name = " + viewName);
   }
}
```

Output

```
view name = MYVIEW2
```

2.12. Does a Table Exist in a Database?

How do you test whether a given table name exists in a database? For the sake of our discussion, let's say that the table name is `TABLE_NAME`. At least four possible solutions exist:

- **Solution 1**: Use the `getTableNames()` method as described earlier and then check to see if the list contains your desired table (e.g., `TABLE_NAME`).
- **Solution 2**: Execute the following SQL statement; if execution is successful (meaning there was no `SQLException`) then the table exists; otherwise the table does not exist:

  ```
  select * from TABLE_NAME where 1=0;
  ```

  This solution is preferable to others. Because the boolean expression `1=0` evaluates to `false`, no rows will be selected. This expression only checks whether or not the table exists (in other words, if the table exists, then it returns no records and no exception is raised; otherwise it will throw a `SQLException`).
- **Solution 3**: Execute the following SQL statement; if execution is successful (meaning there was no `SQLException`) then the table exists; otherwise the table does not exist:

  ```
  select count(*) from TABLE_NAME;
  ```

  This solution might require a full table scan (to obtain the number of rows or records), and therefore using this solution might be more expensive than the others.
- **Solution 4**: You may use a database vendor's catalog to find out if a given table exists (this will be a proprietary solution). Using an Oracle database, for example, you may execute the following SQL query:

  ```
  select object_name from user_objects
      where object_type = 'TABLE' and
            object_name = 'YOUR-TABLE-NAME';
  ```

  If this query returns any rows, then the table does exist; otherwise it does not. This solution is an optimal one for Oracle databases (but it's not portable to other databases).

Solution 1: Use getTableNames()

```
/**
 * Table Exist: Solution 1
 * Check whether a given table (identified by a tableName
 * parameter) exists for a given connection object.
 * @param conn the Connection object
 * @param tableName the table name (to check to see if it exists)
 * @return true if table exists, otherwise return false.
 */
```

```
    public static boolean tableExist1(java.sql.Connection conn,
                                      String tableName) {

        if ((tableName == null) || (tableName.length() == 0)) {
            return false;
        }

        try {
            java.util.List allTables = getTableNames(conn);
            for (int i = 0; i < allTables.size(); i++) {
                String dbTable = (String) allTables.get(i);
                if (dbTable != null) {
                    if (dbTable.equalsIgnoreCase(tableName)) {
                        return true;
                    }
                }
            }

            // table does not exist
            return false;
        }
        catch(Exception e) {
            //e.printStackTrace();
            // table does not exist or some other problem
            return false;
        }
    }
```

Testing Solution 1

```
        //
        // does table TestTable77 exist
        //
        boolean exist77 = DatabaseMetaDataTool.tableExist1(conn, "TestTable77");
        System.out.println("-------- does table TestTable77 exist -------");
        System.out.println(exist77);
        System.out.println("-----------------------------------------------");

        //
        // does table TestTable88 exist
        //
        boolean exist88 = DatabaseMetaDataTool.tableExist1(conn, "TestTable88");
        System.out.println("-------- does table TestTable88 exist -------");
        System.out.println(exist88);
        System.out.println("-----------------------------------------------");
```

Output for Testing Solution 1

```
-------- does table TestTable77 exist -------
true
---------------------------------------------
-------- does table TestTable88 exist -------
false
---------------------------------------------
```

Solution 2: Execute select * from TABLE_NAME where 1=0;

```
/**
 * Table Exist: Solution 2
 * Check whether a given table (identified by a tableName
 * parameter) exists for a given connection object.
 * @param conn the Connection object
 * @param tableName the table name (to check to see if it exists)
 * @return true if table exists, otherwise return false.
 */
 public static boolean tableExist2(java.sql.Connection conn,
                                   String tableName) {

    if ((tableName == null) || (tableName.length() == 0)) {
        return false;
    }

    String query = "select * from " + tableName + " where 1=0";
    Statement stmt = null;
    ResultSet rs = null;
    try {
        stmt = conn.createStatement();
        rs = stmt.executeQuery(query);
        return true;
    }
    catch (Exception e ) {
        // table does not exist or some other problem
        //e.printStackTrace();
        return false;
    }
    finally {
        DatabaseUtil.close(rs);
        DatabaseUtil.close(stmt);
    }
}
```

Testing Solution 2

```
//
// does table TestTable77 exist
//
boolean exist77 = DatabaseMetaDataTool.tableExist2(conn, "TestTable77");
System.out.println("-------- does table TestTable77 exist -------");
System.out.println(exist77);
System.out.println("-------------------------------------------");

//
// does table TestTable88 exist
//
boolean exist88 = DatabaseMetaDataTool.tableExist2(conn, "TestTable88");
System.out.println("-------- does table TestTable88 exist -------");
System.out.println(exist88);
System.out.println("-------------------------------------------");
```

Output for Testing Solution 2

```
-------- does table TestTable77 exist -------
true
-------------------------------------------
-------- does table TestTable88 exist -------
false
-------------------------------------------
```

Solution 3: Execute select count(*) from TABLE_NAME;

This solution will work for both MySQL and Oracle databases:

```
/**
 * Table Exist: Solution 3
 * Check whether a given table (identified by a tableName
 * parameter) exists for a given connection object.
 * @param conn the Connection object
 * @param tableName the table name (to check to see if it exists)
 * @return true if table exists, otherwise return false.
 */
public static boolean tableExist3(java.sql.Connection conn,
                                  String tableName) {

   if ((tableName == null) || (tableName.length() == 0)) {
       return false;
   }
```

```
        String query = "select count(*) from " + tableName;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery(query);
            return true;
        }
        catch (Exception e ) {
            // table does not exist or some other problem
            //e.printStackTrace();
            return false;
        }
        finally {
            DatabaseUtil.close(rs);
            DatabaseUtil.close(stmt);
        }
    }
```

Testing Solution 3

```
        //
        // does table TestTable77 exist
        //
        boolean exist77 = DatabaseMetaDataTool.tableExist3(conn, "TestTable77");
        System.out.println("-------- does table TestTable77 exist -------");
        System.out.println(exist77);
        System.out.println("-------------------------------------------");

        //
        // does table TestTable88 exist
        //
        boolean exist88 = DatabaseMetaDataTool.tableExist3(conn, "TestTable88");
        System.out.println("-------- does table TestTable88 exist -------");
        System.out.println(exist88);
        System.out.println("-------------------------------------------");
```

Output for Testing Solution 3

```
        -------- does table TestTable77 exist -------
        true
        -------------------------------------------
        -------- does table TestTable88 exist -------
        false
        -------------------------------------------
```

2.13. What Are a Table's Column Names?

When you're building SQL adapters and database GUI applications, keep in mind that clients might be interested in viewing and selecting columns (and their associated data types). Before you insert new records into a table, you might want to check the table columns' associated types. Doing so can prevent redundant network traffic.

You can use `DatabaseMetaData.getColumns()` to get list of columns for a table or view. In production environments, try to minimize passing null/empty parameter values to this method. Passing non-null and non-empty parameter values to JDBC metadata methods can improve the overall performance of your applications.

getColumnNames()

```
/**
 * Get column names and their associated types. The result
 * is returned as a Hashtable, where key is "column name"
 * and value is "column type". If table name is null/empty
 * it returns null.
 *
 * @param conn the Connection object
 * @param tableName name of a table in the database.
 * @return an Hashtable, where key is "column name"
 * and value is "column type".
 * @exception Failed to get the column names for a given table.
 */
public static java.util.Hashtable getColumnNames(java.sql.Connection conn,
                                                  String tableName)
    throws Exception {
    ResultSet rsColumns = null;
    try {
         if ((tableName == null) || (tableName.length() == 0)) {
             return null;
         }

         DatabaseMetaData meta = conn.getMetaData();
         if (meta == null) {
             return null;
         }

         // Oracle requires table names to in uppercase characters
         // MySQL is case-insensitive to table names
         rsColumns = meta.getColumns(null, null, tableName.toUpperCase(), null);
         Hashtable columns = new Hashtable();
         while (rsColumns.next()) {
             // private static final String COLUMN_NAME_COLUMN_NAME = "COLUMN_NAME";
             // private static final String COLUMN_NAME_TYPE_NAME = "TYPE_NAME";
             String columnType = rsColumns.getString(COLUMN_NAME_TYPE_NAME);
```

```
            String columnName = rsColumns.getString(COLUMN_NAME_COLUMN_NAME);
            if (columnName != null) {
                columns.put(columnName, columnType);
            }
        }
        return columns;
    }
    catch(Exception e) {
        throw new Exception("Error: could not get column names: "+e.toString());
    }
    finally {
        DatabaseUtil.close(rsColumns);
    }
}
```

Test Program

The following test program prints a listing of the column names for the table `MyPayrollTable`:

```
//
// print the list of column names for table MyPayrollTable
//
java.util.Hashtable result =
   DatabaseMetaDataTool.getColumnNames(conn, "MyPayrollTable");
System.out.println("Got results: list of column names -------------");
java.util.Enumeration columns = result.keys();
while (columns.hasMoreElements()) {
    Object columnKey = columns.nextElement();
    String columnName = (String) columnKey;
    String columnType = (String) result.get(columnKey);
    System.out.println("column name = " + columnName);
    System.out.println("column type = " + columnType);
}
```

Output of the Test Program

```
Got results: list of column names -------------
column name = ID
column type = VARCHAR2
column name = SALARY
column type = NUMBER
column name = NAME
column type = VARCHAR2
```

The name and type combination provides information about the table schema, but it is not enough. You need to get other useful information, such as the size of the column and whether the column is nullable. (*Nullable* means that the column accepts the `NULL` value; note that `NULL` in SQL is not a zero or an empty value but instead indicates that the value is missing.) So, you

can modify the program to provide more detailed information for each column. Since you are returning four distinct pieces of information for each column, you will return the result as an XML `String` object. For each column, the following information will be returned:

```
    <column name="NameOfColumn">
        <type>TypeOfColumn</type>
        <size>SizeOfColumn</size>
        <nullable>true|false</nullable>
    </column>
```

Testing getColumnDetails()

```
//
// print the detail of columns for table TestTable77
//
String columnDetails = DatabaseMetaDataTool.getColumnDetails(conn, "TestTable77");
System.out.println("-------- columnDetails -------------");
System.out.println(columnDetails);
System.out.println("-----------------------------------");
```

Output of Testing getColumnDetails()

```
<columns>
    <column name="id">
        <type>varchar</type>
        <size>10</size>
        <nullable>false</nullable>
        <position>1</position>
    </column>
    <column name="name">
        <type>varchar</type>
        <size>20</size>
        <nullable>false</nullable>
        <position>2</position>
    </column>
    <column name="age">
        <type>int</type>
        <size>11</size>
        <nullable>true</nullable>
        <position>3</position>
    </column>
    <column name="address">
        <type>varchar</type>
        <size>100</size>
        <nullable>true</nullable>
        <position>4</position>
    </column>
</columns>
```

getColumnDetails() Method

```
/**
 * Get column names and their associated attributes (type,
 * size, nullable, ordinal position). The result is returned
 * as XML (as a string object);  if table name is null/empty
 * it returns null.
 *
 * @param conn the Connection object
 * @param tableName name of a table in the database.
 * @return XML (column names and their associated attributes:
 * type, size, nullable, ordinal position).
 * @exception Failed to get the column details for a given table.
 */
public static String getColumnDetails(java.sql.Connection conn,
                                      String tableName)
    throws Exception {

    ResultSet rsColumns = null;
    StringBuilder sb = new StringBuilder();
    try {
         if ((tableName == null) || (tableName.length() == 0)) {
             return null;
         }

         DatabaseMetaData meta = conn.getMetaData();
         if (meta == null) {
             return null;
         }

         rsColumns = meta.getColumns(null, null, tableName.toUpperCase(), null);
         sb.append("<columns>");
         while (rsColumns.next()) {

             String columnType = rsColumns.getString(COLUMN_NAME_TYPE_NAME);
             String columnName = rsColumns.getString(COLUMN_NAME_COLUMN_NAME);
             int size = rsColumns.getInt(COLUMN_NAME_COLUMN_SIZE);
             int nullable = rsColumns.getInt(COLUMN_NAME_NULLABLE);
             int position = rsColumns.getInt(COLUMN_NAME_ORDINAL_POSITION);

             sb.append("<column name=\"");
             sb.append(columnName);
             sb.append("\"><type>");
             sb.append(columnType);
             sb.append("</type><size>");
             sb.append(size);
             sb.append("</size><nullable>");
             if (nullable == DatabaseMetaData.columnNullable) {
                 sb.append("true");
             }
```

```
            else {
                sb.append("false");
            }
            sb.append("</nullable><position>");
            sb.append(position);
            sb.append("</position></column>");
        }
        sb.append("</columns>");
        return sb.toString();
    }
    catch(Exception e) {
        throw new Exception("Error: could not get column names: "+e.toString());
    }
    finally {
        DatabaseUtil.close(rsColumns);
    }
}
```

getColumnDetails() Method: Selecting Type, Size, Nullable, Position

The following method selects the specific metadata that you are interested in by passing a boolean flag for each metadata:

```
/**
 * Get column names and their associated attributes (type,
 * size, nullable). The result is returned as an XML (as
 * a string object);  if table name is null/empty
 * it returns null.
 *
 * @param conn the Connection object
 * @param tableName name of a table in the database.
 * @param includeType if true, then include type information.
 * @param includeSize if true, then include size information.
 * @param includeNullable if true, then include nullable information.
 * @param includePosition if true, then include ordinal position information.
 * @return an XML: column names and their associated attributes: type,
 * size, nullable.
 * @exception Failed to get the column details for a given table.
 */
public static String getColumnDetails(java.sql.Connection conn,
                                      String tableName,
                                      boolean includeType,
                                      boolean includeSize,
                                      boolean includeNullable,
                                      boolean includePosition)
```

```
throws Exception {
ResultSet rsColumns = null;
StringBuffer sb = new StringBuffer();
try {
     if ((tableName == null) || (tableName.length() == 0)) {
         return null;
     }

     DatabaseMetaData meta = conn.getMetaData();
     if (meta == null) {
         return null;
     }

     rsColumns = meta.getColumns(null, null, tableName.toUpperCase(), null);
     sb.append("<columns>");
     while (rsColumns.next()) {
         String columnName = rsColumns.getString(COLUMN_NAME_COLUMN_NAME);
         sb.append("<column name=\"");
         sb.append(columnName);
         sb.append("\">");
         if (includeType) {
             String columnType = rsColumns.getString(COLUMN_NAME_TYPE_NAME);
             sb.append("<type>");
             sb.append(columnType);
             sb.append("</type>");
         }

         if (includeSize) {
             int size = rsColumns.getInt(COLUMN_NAME_COLUMN_SIZE);
             sb.append("<size>");
             sb.append(size);
             sb.append("</size>");
         }

         if (includeNullable) {
            int nullable = rsColumns.getInt(COLUMN_NAME_NULLABLE);
            sb.append("<nullable>");
            if (nullable == DatabaseMetaData.columnNullable) {
                sb.append("true");
            }
            else {
                sb.append("false");
            }
            sb.append("</nullable>");
         }
```

```
            if (includePosition) {
                int position = rsColumns.getInt(COLUMN_NAME_ORDINAL_POSITION);
                sb.append("<position>");
                sb.append(position);
                sb.append("</position>");
            }
            sb.append("</column>");
        }
        sb.append("</columns>");
        return sb.toString();
    }
    catch(Exception e) {
        throw new Exception("Error: could not get column names: "+e.toString());
    }
    finally {
        DatabaseUtil.close(rsColumns);
    }
}
```

2.14. What Are the Table Types Used in a Database?

Using RDBMS, we store data and metadata in logical tables (which are stored in operating system files). Typically, each database vendor has different types of logical storage types (such as `TABLE`, `VIEW`, `SYSTEM TABLE`, etc.) for storing data and metadata. For example, Oracle does support most of the table types, but some databases (such as MySQL) do not support "views" (MySQL 5.0 does support views, however). In GUI database applications, you might want to distinguish different table types. The `getTableTypes()` method, defined in the `DatabaseMetaData` interface, retrieves the table types available in this database. The results are ordered by table type. The result is returned as a `ResultSet` object in which each row has a single `String` column that is a table type. A single column is named as `TABLE_TYPE`.

The table type is

- `"TABLE"`
- `"VIEW"`
- `"SYSTEM TABLE"`
- `"GLOBAL TEMPORARY"`
- `"LOCAL TEMPORARY"`
- `"ALIAS"`
- `"SYNONYM"`

The Solution: getTableTypes()

```
/**
 * Get the table types for a database.
 * @param conn the Connection object.
 * @return the list of table types as a List.
 * @exception Failed to get the table types from the database.
 */
 public static java.util.List getTableTypes(java.sql.Connection conn)
    throws Exception {
    ResultSet rs = null;
    try {
         DatabaseMetaData meta = conn.getMetaData();
         if (meta == null) {
             return null;
         }

         rs = meta.getTableTypes();
         if (rs == null) {
             return null;
         }

         java.util.List list = new java.util.ArrayList();
         //System.out.println("getTableTypes(): --------------");
         while (rs.next()) {
             String tableType = DatabaseUtil.getTrimmedString(rs, 1);
             //System.out.println("tableType="+tableType);
             if (tableType != null) {
                 list.add(tableType);
             }
         }
         //System.out.println("--------------");
         return list;
     }
     catch(Exception e) {
         e.printStackTrace();
         throw e;
     }
     finally {
         DatabaseUtil.close(rs);
     }
}
```

Oracle Client Program

```
//
// print the list of table types
//
java.util.List tableTypes = DatabaseMetaDataTool.getTableTypes(conn);
System.out.println("Got results: list of table types --------------");
for (int i=0; i < tableTypes.size(); i++) {
    // process results one element at a time
    String tableType = (String) tableTypes.get(i);
    System.out.println("table type = " + tableType);
}
```

Oracle Client Program Output

```
Got results: list of table types --------------
table type = SYNONYM
table type = TABLE
table type = VIEW
```

MySQL Client Program

```
//
// print the list of table types
//
java.util.List tableTypes = DatabaseMetaDataTool.getTableTypes(conn);
System.out.println("Got results: list of table types --------------");
for (int i=0; i < tableTypes.size(); i++) {
    // process results one element at a time
    String tableType = (String) tableTypes.get(i);
    System.out.println("table type = " + tableType);
}
```

MySQL Client Program Output

```
Got results: list of table types --------------
table type = TABLE
table type = LOCAL TEMPORARY
```

2.15. What Are the Primary Keys for a Table?

The primary key (PK) of a relational table uniquely identifies each row (or record) in the table. `DatabaseMetaData` provides the `getPrimaryKeys()` method, which retrieves a description of the given table's primary key columns:

```
ResultSet getPrimaryKeys(String catalog, String schema, String table)
```

When invoking the `DatabaseMetaData.getPrimaryKeys()` method, be sure to pass `catalog` and `schema` values (if they are known); if you pass `null` values for the first two parameters, then it might take too long to return the result. Database applications that involve the dynamic insertion of records need to know which columns form the primary key. In this way, the application can control the values sent for the insertion of new records. Again, this can be useful when you build database adapters. The solution is as follows:

```
import java.util.*;
import java.io.*;
import java.sql.*;

/**
 * This class provides class-level methods for getting database metadata.
 *
 */
 public class DatabaseMetaDataTool {

    /**
     * Retrieves a description of the given table's primary key columns.
     * @param conn the Connection object
     * @param tableName name of a table in the database.
     * @return the list of column names (which form the Primary Key) as a List.
     * @exception Failed to get the PrimaryKeys for a given table.
     */
     public static java.util.List getPrimaryKeys(java.sql.Connection conn,
                                                 String tableName)
        throws Exception {
        ResultSet rs = null;
        try {
             if ((tableName == null) || (tableName.length() == 0)) {
                 return null;
             }

             DatabaseMetaData meta = conn.getMetaData();
             if (meta == null) {
                 return null;
             }

             //
             // The Oracle database stores its table names as uppercase,
             // if you pass a table name in lowercase characters, it will not work.
             // MySQL database does not care if table name is uppercase/lowercase.
             // if you know catalog and schema, then pass them explicitly
             // rather than passing null values (you might pay a performance
             // penalty for passing null values)
             rs = meta.getPrimaryKeys(null, null, tableName.toUpperCase());
             if (rs == null) {
                 return null;
             }
```

```
            java.util.List list = new java.util.ArrayList();
            while (rs.next()) {
                String columnName =
                  DatabaseUtil.getTrimmedString(rs, COLUMN_NAME_COLUMN_NAME);
                System.out.println("getPrimaryKeys(): columnName="+columnName);
                if (columnName != null) {
                    list.add(columnName);
                }
            }
            return list;
        }
        catch(Exception e) {
            e.printStackTrace();
            throw e;
        }
        finally {
            DatabaseUtil.close(rs);
        }
    }

    // other methods and constants ...
}
```

Consider the following table in a MySQL database:

```
mysql> describe TestTable77;
```

```
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| id      | varchar(10)  |      | PRI |         |       |
| name    | varchar(20)  |      | PRI |         |       |
| age     | int(11)      | YES  |     | NULL    |       |
| address | varchar(100) | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
4 rows in set (0.02 sec)
```

In order to find the Primary Key column names for the `TestTable77` table, we need to write the following code segment:

```
//
// print the list of PKs
//
java.util.List pks = DatabaseMetaDataTool.getPrimaryKeys(conn, "TestTable77");
System.out.println("Got results: list of PKs -------------");
for (int i=0; i < pks.size(); i++) {
  // process results one element at a time
  String columnName = (String) pks.get(i);
  System.out.println("column name = " + columnName);
}
```

The output will be:

```
getPrimaryKeys(): columnName=id
getPrimaryKeys(): columnName=name
Got results: list of PKs -------------
column name = id
column name = name
```

2.16. What Are a Table's Privileges?

A database table's *privileges* refer to finding a description of the access rights for each table available in a catalog or schema. `DatabaseMetaData` provides a method, `getTablePrivileges()`, to do just that. This method returns the result as a `ResultSet` where each row is a table privilege description. In production applications, returning the result as a `ResultSet` is not quite useful. It is better to return the result as an XML object so that the client can extract the required information and display it in a desired format. It would be wrong to assume that this privilege applies to all columns; while this may be true for some systems, it is not true for all.

`getTablePrivileges()` returns only privileges that match the schema and table name criteria. They are ordered by `TABLE_SCHEM`, `TABLE_NAME`, and `PRIVILEGE`. Each privilege description has the columns shown in Table 2-3.

Table 2-3. *Columns for Result of getTablePrivileges()*

Column's Position	Column's Name	Description
1	`TABLE_CAT`	Table catalog (may be `null`)
2	`TABLE_SCHEM`	Table schema (may be `null`)
3	`TABLE_NAME`	Table name (as a `String`)
4	`GRANTOR`	Grantor of access (may be `null`)
5	`GRANTEE`	Grantee of access
6	`PRIVILEGE`	Name of access (`SELECT`, `INSERT`, `UPDATE`, `REFERENCES`, etc.)
7	`IS_GRANTABLE`	`YES` if grantee is permitted to grant to others; `NO` if not; `null` if unknown

The `getTablePrivileges()` method has the following signature:

```
public java.sql.ResultSet getTablePrivileges(String catalog,
                                             String schemaPattern,
                                             String tableNamePattern)
        throws java.sql.SQLException
```

where

- `catalog`: A catalog name; "" retrieves those without a catalog.
- `schemaPattern`: A schema name pattern; "" retrieves those without a schema.
- `tableNamePattern`: A table name pattern.

The Solution: Get Table Privileges

```
/**
 * Get Table Privileges: retrieves a description of the access
 * rights for each table available in a catalog. Note that a
 * table privilege applies to one or more columns in the table.
 * It would be wrong to assume that this privilege applies to
 * all columns (this may be true for some systems but is not
 * true for all.)  The result is returned as XML (as a string
 * object);  if table name is null/empty it returns null.
 *
 * In JDBC, Each privilege description has the following columns:
 *
 * TABLE_CAT String => table catalog (may be null)
 * TABLE_SCHEM String => table schema (may be null)
 * TABLE_NAME String => table name
 * GRANTOR => grantor of access (may be null)
 * GRANTEE String => grantee of access
 * PRIVILEGE String => name of access (SELECT, INSERT,
 *      UPDATE, REFERENCES, ...)
 * IS_GRANTABLE String => "YES" if grantee is permitted to grant
 *      to others; "NO" if not; null if unknown
 *
 *
 * @param conn the Connection object
 * @param catalogPattern a catalog pattern.
 * @param schemaPattern a schema pattern.
 * @param tableNamePattern a table name pattern; must match
 *  the table name as it is stored in the database .
 * @return an XML.
 * @exception Failed to get the Get Table Privileges.
 */
public static String getTablePrivileges(java.sql.Connection conn,
                                        String catalogPattern,
                                        String schemaPattern,
                                        String tableNamePattern)
    throws Exception {

    ResultSet privileges = null;
    StringBuffer sb = new StringBuffer();
    try {
         if ((tableNamePattern == null) ||
             (tableNamePattern.length() == 0)) {
             return null;
         }

         DatabaseMetaData meta = conn.getMetaData();
         if (meta == null) {
             return null;
         }
```

```
            // The '_' character represents any single character.
            // The '%' character represents any sequence of zero
            // or more characters.
            privileges = meta.getTablePrivileges(catalogPattern,
                                                 schemaPattern,
                                                 tableNamePattern);
            sb.append("<privileges>");
            while (privileges.next()) {

                String catalog = privileges.getString(COLUMN_NAME_TABLE_CATALOG);
                String schema = privileges.getString(COLUMN_NAME_TABLE_SCHEMA);
                String tableName = privileges.getString(COLUMN_NAME_TABLE_NAME);
                String privilege = privileges.getString(COLUMN_NAME_PRIVILEGE);
                String grantor = privileges.getString(COLUMN_NAME_GRANTOR);
                String grantee = privileges.getString(COLUMN_NAME_GRANTEE);
                String isGrantable = privileges.getString(COLUMN_NAME_IS_GRANTABLE);

                sb.append("<table name=\"");
                sb.append(tableName);
                sb.append("\"><catalog>");
                sb.append(catalog);
                sb.append("</catalog><schema>");
                sb.append(schema);
                sb.append("</schema><privilege>");
                sb.append(privilege);
                sb.append("</privilege><grantor>");
                sb.append(grantor);
                sb.append("</grantor><isGrantable>");
                sb.append(isGrantable);
                sb.append("</isGrantable><grantee>");
                sb.append(grantee);
                sb.append("</grantee></table>");
            }
            sb.append("</privileges>");
            return sb.toString();
        }
        catch(Exception e) {
            throw new Exception("Error: could not get table privileges:"+
               e.toString());
        }
        finally {
            DatabaseUtil.close(privileges);
        }
    }
```

Oracle: Client Call: Get Table Privileges

```
String tablePrivileges = DatabaseMetaDataTool.getTablePrivileges
    (conn,                // connection
     conn.getCatalog(),   // catalog
     "%",                 // schema
     "EMP%");             // table name pattern
System.out.println("-------- TablePrivileges -------------");
System.out.println(tablePrivileges);
System.out.println("-----------------------------------");
```

And here's the output:

```
-------- TablePrivileges -------------
<privileges>
    <table name="EMPLOYEE_PHOTOS">
        <catalog>null</catalog>
        <schema>SYS</schema>
        <privilege>READ</privilege>
        <grantor>SYS</grantor>
        <isGrantable>YES</isGrantable>
        <grantee>OCTOPUS</grantee>
    </table>
    <table name="EMPLOYEE_PHOTOS">
        <catalog>null</catalog>
        <schema>SYS</schema>
        <privilege>WRITE</privilege>
        <grantor>SYS</grantor>
        <isGrantable>YES</isGrantable>
        <grantee>OCTOPUS</grantee>
    </table>
</privileges>
-----------------------------------
```

MySQL: Client Call: Get Table Privileges

The MySQL database stores table privileges in the `tables_priv` table. Here is the description of that table (note that the MySQL database uses the `mysql` database to manage users and privileges):

```
mysql> use mysql;
mysql> desc tables_priv;
+-------------+-----------------+------+-----+-------------------+-------+
| Field       | Type            | Null | Key | Default           | Extra |
+-------------+-----------------+------+-----+-------------------+-------+
| Host        | char(60)        |      | PRI |                   |       |
| Db          | char(64)        |      | PRI |                   |       |
| User        | char(16)        |      | PRI |                   |       |
| Table_name  | char(64)        |      | PRI |                   |       |
| Grantor     | char(77)        |      | MUL |                   |       |
| Timestamp   | timestamp       | YES  |     | CURRENT_TIMESTAMP |       |
| Table_priv  | set('Select',   |      |     |                   |       |
|             |     'Insert',   |      |     |                   |       |
|             |     'Update',   |      |     |                   |       |
|             |     'Delete',   |      |     |                   |       |
|             |     'Create',   |      |     |                   |       |
|             |     'Drop',     |      |     |                   |       |
|             |     'Grant',    |      |     |                   |       |
|             |     'References'|      |     |                   |       |
|             |     ,'Index',   |      |     |                   |       |
|             |      'Alter')   |      |     |                   |       |
| Column_priv | set('Select',   |      |     |                   |       |
|             |     'Insert',   |      |     |                   |       |
|             |     'Update',   |      |     |                   |       |
|             |    'References' |      |     |                   |       |
+-------------+-----------------+------+-----+-------------------+-------+
8 rows in set (0.00 sec)
```

Client Call

```
String tablePrivileges = DatabaseMetaDataTool.getTablePrivileges
    (conn, // connection
     conn.getCatalog(), // catalog
     null, //"%", // schema
     "%");
System.out.println("-------- TablePrivileges -------------");
System.out.println(tablePrivileges);
System.out.println("-----------------------------------");
```

Output

```
-------- TablePrivileges -------------
<privileges>

<table name="artist">
<catalog>octopus</catalog>
<schema>null</schema>
<privilege>SELECT</privilege>
```

```
<grantor>root@127.0.0.1</grantor>
<isGrantable>null</isGrantable>
<grantee>%@%</grantee>
</table>

<table name="artist_exhibit">
<catalog>octopus</catalog>
<schema>null</schema>
<privilege>SELECT</privilege>
<grantor>root@127.0.0.1</grantor>
<isGrantable>null</isGrantable>
<grantee>%@%</grantee>
</table>

<table name="artist_exhibit">
<catalog>octopus</catalog>
<schema>null</schema>
<privilege>INSERT</privilege>
<grantor>root@127.0.0.1</grantor>
<isGrantable>null</isGrantable>
<grantee>%@%</grantee>
</table>

<table name="artist_exhibit">
<catalog>octopus</catalog>
<schema>null</schema>
<privilege>DROP</privilege>
<grantor>root@127.0.0.1</grantor>
<isGrantable>null</isGrantable>
<grantee>%@%</grantee>
</table>

</privileges>
-----------------------------------
```

2.17. What Are a Table Column's Privileges?

To retrieve a description of a given table column's privileges—that is, its access rights—you can use the `DatabaseMetaData` interface's `getColumnPrivileges()` method. This method returns a list of columns and associated privileges for the specified table. The `getColumnPrivileges()` method returns the result as a `ResultSet`, where each row is a column privilege description. In production applications, returning the result as a `ResultSet` is not that useful. It is better to return the result as XML so that the client application can extract the required information and display it in a desired fashion. Note that this privilege does not apply to all columns—this may be true for some systems, but it is not true for all.

The getColumnPrivileges() method retrieves a description of the access rights for a table's columns. Only privileges matching the column name criteria are returned. They are ordered by COLUMN_NAME and PRIVILEGE.

Each privilege description has the columns shown in Table 2-4.

Table 2-4. *Columns for Result of getColumnPrivileges()*

Column's Position	Column's Name	Description
1	TABLE_CAT	Table catalog (may be null)
2	TABLE_SCHEM	Table schema (may be null)
3	TABLE_NAME	Table name (as a String)
4	COLUMN_NAME	Column name (as a String)
5	GRANTOR	Grantor of access (may be null)
6	GRANTEE	Grantee of access
7	PRIVILEGE	Name of access (SELECT, INSERT, UPDATE, REFERENCES, etc.)
8	IS_GRANTABLE	YES if grantee is permitted to grant to others; NO if not; null if unknown

getColumnPrivileges

```
public ResultSet getColumnPrivileges(String catalog,
                                     String schema,
                                     String table,
                                     String pattern)
                              throws SQLException
```

Here are the parameters for the getColumnPrivileges() method:

- catalog: A catalog name; it must match the catalog name as it is stored in the database. "" retrieves those without a catalog; null means that the catalog name should not be used to narrow the search.
- schema: A schema name; it must match the schema name as it is stored in the database. "" retrieves those without a schema; null means that the schema name should not be used to narrow the search.
- table: A table name; it must match the table name as it is stored in the database.
- pattern: A column name pattern; it must match the column name as it is stored in the database.

getColumnPrivileges returns a ResultSet, where each row is a column privilege description. It throws a SQLException if a database access error occurs.

The Solution: Get Table Column Privileges

```
/**
 * Get Table Column Privileges: retrieves a description
 * of the access rights for a table's columns available in
 * a catalog.  The result is returned as XML (as a string
 * object);  if table name is null/empty it returns null.
 *
 * In JDBC, each privilege description has the following columns:
 *
 * TABLE_CAT String => table catalog (may be null)
 * TABLE_SCHEM String => table schema (may be null)
 * TABLE_NAME String => table name
 * COLUMN_NAME String => column name
 * GRANTOR => grantor of access (may be null)
 * GRANTEE String => grantee of access
 * PRIVILEGE String => name of access (SELECT, INSERT, UPDATE, REFERENCES, ...)
 * IS_GRANTABLE String => "YES" if grantee is permitted to grant
 *      to others; "NO" if not; null if unknown
 *
 * @param conn the Connection object
 * @param catalog a catalog.
 * @param schema a schema.
 * @param tableName a table name; must match
 *  the table name as it is stored in the database .
 * @param columnNamePattern a column name pattern.
 * @return an XML.
 * @exception Failed to get the Get Table Column Privileges.
 */
public static String getColumnPrivileges(java.sql.Connection conn,
                                         String catalog,
                                         String schema,
                                         String tableName,
                                         String columnNamePattern)
    throws Exception {
    ResultSet privileges = null;
    StringBuffer sb = new StringBuffer();
    try {
         if ((tableName == null) ||
             (tableName.length() == 0)) {
             return null;
         }

         DatabaseMetaData meta = conn.getMetaData();
         if (meta == null) {
             return null;
         }
```

```
            // The '_' character represents any single character.
            // The '%' character represents any sequence of zero
            // or more characters.
            // NOTE: if you pass a null to schema/tableName, then you might get
            // an exception or you might get an empty ResultSet object
            privileges = meta.getColumnPrivileges(catalog,
                                                  schema,
                                                  tableName,
                                                  columnNamePattern);
            sb.append("<privileges>");
            while (privileges.next()) {
                String dbCatalog = privileges.getString(COLUMN_NAME_TABLE_CATALOG);
                String dbSchema = privileges.getString(COLUMN_NAME_TABLE_SCHEMA);
                String dbTable = privileges.getString(COLUMN_NAME_TABLE_NAME);
                String dbColumn = privileges.getString(COLUMN_NAME_COLUMN_NAME);
                String dbPrivilege = privileges.getString(COLUMN_NAME_PRIVILEGE);
                String dbGrantor = privileges.getString(COLUMN_NAME_GRANTOR);
                String dbGrantee = privileges.getString(COLUMN_NAME_GRANTEE);
                String dbIsGrantable = privileges.getString(COLUMN_NAME_IS_GRANTABLE);

                sb.append("<column name=\"");
                sb.append(dbColumn);
                sb.append("\" table=\"");
                sb.append(tableName);
                sb.append("\"><catalog>");
                sb.append(dbCatalog);
                sb.append("</catalog><schema>");
                sb.append(dbSchema);
                sb.append("</schema><privilege>");
                sb.append(dbPrivilege);
                sb.append("</privilege><grantor>");
                sb.append(dbGrantor);
                sb.append("</grantor><isGrantable>");
                sb.append(dbIsGrantable);
                sb.append("</isGrantable><grantee>");
                sb.append(dbGrantee);
                sb.append("</grantee></column>");
            }
            sb.append("</privileges>");
            return sb.toString();
        }
        catch(Exception e) {
            throw new Exception("Error: could not get table column privileges: "+
                                e.toString());
        }
        finally {
            DatabaseUtil.close(privileges);
        }
    }
```

Oracle: Test the Solution: Get Table Column Privileges

I tested `getColumnPrivileges()` for several Oracle tables, but I could not get any results (it seems that the Oracle driver does not support this feature at all).

The client call looks like this:

```
String columnPrivileges = DatabaseMetaDataTool.getColumnPrivileges
    (conn,               // connection
     conn.getCatalog(),  // catalog
     "SYSTEM",           // schema
     "HELP",             // the help table
     "%");
System.out.println("---- Table's Columns Privileges ----");
System.out.println(columnPrivileges);
System.out.println("-----------------------------------");
```

The output is

```
---- Table Column Privileges ----
<privileges>
</privileges>
-----------------------------------
```

MySQL: Test the Solution: Get Table Column Privileges

The client call looks like this:

```
String columnPrivileges = DatabaseMetaDataTool.getColumnPrivileges
    (conn,               // connection
     conn.getCatalog(),  // catalog
     null,               // schema
     "artist",
     "%");
System.out.println("---- Table Column Privileges ----");
System.out.println(columnPrivileges);
System.out.println("-----------------------------------");
```

The output is

```
---- Table Column Privileges ----
<privileges>
    <column name="ARTIST_ID" table="artist">
        <catalog>octopus</catalog>
        <schema>null</schema>
        <privilege>SELECT</privilege>
        <grantor>root@127.0.0.1</grantor>
        <isGrantable>null</isGrantable>
        <grantee>%@%</grantee>
    </column>
```

```
    <column name="ARTIST_NAME" table="artist">
        <catalog>octopus</catalog>
        <schema>null</schema>
        <privilege>SELECT</privilege>
        <grantor>root@127.0.0.1</grantor>
        <isGrantable>null</isGrantable>
        <grantee>%@%</grantee>
    </column>
</privileges>
-----------------------------------
```

2.18. How Do You Find the Number of Rows Affected by a SQL Query?

Suppose you execute a query to delete some rows and you want to know how many rows were deleted. In general, the JDBC API provides two methods (available in the `java.sql.Statement` interface) to find the number of rows affected by a SQL query: `execute()` and `executeUpdate()`.

Get the Number of Rows Affected Using the executeUpdate() Method

Using the `executeUpdate()` method, we can get the number of rows affected:

```
Connection conn = ... get a java.sql.Connection object ...
String sqlQuery = "delete from employees where employee_status = 'inactive'";
Statement stmt = conn.createStatement();
int rowsAffected = stmt.executeUpdate(sqlQuery);
System.out.println("number of rows affected = "+ rowsAffected);
```

Also, we may write this as a method:

```
/**
 * Get the number of rows affected for a given SQL query.
 * @param conn the connection object.
 * @param sqlQuery the SQL query to be executed.
 * @return the number of rows affected by the execution of the SQL query.
 * @exception Failed to execute the SQL query.
 *
 */
public static int getNumberOfRowsAffected(Connection conn,
                                          String sqlQuery)
    throws Exception {
    Statement stmt = null
    try {
        stmt = conn.createStatement();
        int rowsAffected = stmt.executeUpdate(sqlQuery);
        System.out.println("number of rows affected = "+ rowsAffected);
        return rowsAffected;
    }
```

```
        catch(Exception e) {
            throw new Exception(e.toString+
                        "could not get the number of rows affected");
        }
        finally {
            DatabaseUtil.close(stmt);
        }
    }
```

Get the Number of Rows Affected Using the execute() Method

The `execute()` method executes the given SQL query, which may return multiple results. This method returns a boolean (`true`/`false`). The `execute()` method returns `true` if the first result is a `ResultSet` object; it returns `false` if it is an update count or there are no results.

```
Connection conn = ... get a java.sql.Connection object ...
String sqlQuery = "delete from employees where employeeStatus = 'inactive'";
Statement stmt = conn.createStatement();
if (!stmt.execute(sqlQuery)) {
    //
    // then there is no result set
    // get the number of rows affected
    //
    int rowsAffected = stmt.getUpdateCount();
    System.out.println("number of rows affected = "+ rowsAffected);
}
```

Also, we may write this as a method:

```
/**
 * Get the number of rows affected for a given SQL query.
 * @param conn the connection object.
 * @param sqlQuery the SQL query to be executed.
 * @return the number of rows affected by the execution of the SQL query.
 * @exception Failed to execute the SQL query.
 *
 */
public static int getNumberOfRowsAffected(Connection conn,
                                          String sqlQuery)
    throws Exception {
    Statement stmt = null
    try {
        stmt = conn.createStatement();
        if (!stmt.execute(sqlQuery)) {
            //
            // then there is no result set
            // get the number of rows affected
            //
            int rowsAffected = stmt.getUpdateCount();
```

```
            System.out.println("number of rows affected = "+ rowsAffected);
            return rowsAffected;
        }
        else {
            return 0;
        }
    }
    catch(Exception e) {
        throw new Exception(e.toString+
                    "could not get the number of rows affected");
    }
    finally {
        DatabaseUtil.close(stmt);
    }
}
```

2.19. What Is a Table's Optimal Set of Columns That Uniquely Identify a Row?

To return a table's optimal set of columns that uniquely identify a row, you can use the `DatabaseMetaData` interface's `getBestRowIdentifier()` method. This method returns the result as a `ResultSet` object, which is not very useful to clients. You can provide a wrapper method, `getBestRowIdentifier()`, which is defined in the `DatabaseMetaDataTool` class. It returns the result in XML, and hence can be used by any type of client.

DatabaseMetaData.getBestRowIdentifier() Declaration

```
public ResultSet getBestRowIdentifier(String catalog,
                                      String schema,
                                      String table,
                                      int scope,
                                      boolean nullable)      throws SQLException
```

This method retrieves a description of a table's optimal set of columns that uniquely identifies a row. They are ordered by `SCOPE`. Table 2-5 describes the columns of a `ResultSet` object returned by `getBestRowIdentifier()`.

Table 2-5. *Columns of ResultSet Returned by getBestRowIdentifier()*

Column Name	Column Type	Description
SCOPE	short	Actual scope of result: DatabaseMetaData.bestRowTemporary: Very temporary; only valid while using row DatabaseMetaData.bestRowTransaction: Valid for remainder of current transaction DatabaseMetaData.bestRowSession: Valid for remainder of current session
COLUMN_NAME	String	Column name.

Continued

Table 2-5. *Continued*

Column Name	Column Type	Description
DATA_TYPE	int	SQL data type from java.sql.Types.
TYPE_NAME	String	Data source–dependent type name; for a UDT the type name is fully qualified.
COLUMN_SIZE	int	Precision.
BUFFER_LENGTH	int	Not used.
DECIMAL_DIGITS	short	Scale.
PSEUDO_COLUMN	short	A pseudocolumn like an Oracle ROWID DatabaseMetaData.bestRowUnknown: May or may not be pseudocolumn DatabaseMetaData.bestRowNotPseudo: Is *not* a pseudocolumn DatabaseMetaData.bestRowPseudo: Is a pseudocolumn

Here are the getBestRowIdentifier() parameters:

- catalog: A catalog name; it must match the catalog name as it is stored in the database. "" retrieves those without a catalog; null means that the catalog name should not be used to narrow the search.
- schema: A schema name; it must match the schema name as it is stored in the database. "" retrieves those without a schema; null means that the schema name should not be used to narrow the search.
- table: A table name; it must match the table name as it is stored in the database.
- scope: The scope of interest; it uses the same values as SCOPE (defined earlier).
- nullable: Include columns that are nullable.

XML Syntax for Output

To be as complete as possible, I've expressed the output in XML syntax. This can help clients find the best row identifiers as easily as possible.

```
<?xml version='1.0'>
<BestRowIdentifier>
    <RowIdentifier tableName="database-table-name">
        <scope>actual-scope-of-result</scope>
        <columnName>column-name</columnName>
        <dataType>data-type</dataType>
        <typeName>type-name</typeName>
        <columnSize>size-of-column</columnSize>
        <decimalDigits>scale-for-numeric-columns</decimalDigits>
        <pseudoColumn>pseudo-column</pseudoColumn>
    </RowIdentifier>
    <RowIdentifier tableName="...">
    </RowIdentifier>
```

```
    ...
    <RowIdentifier tableName="...">
    </RowIdentifier>
</BestRowIdentifier>
```

The Solution: getBestRowIdentifier()

The solution is generic enough and can support MySQL, Oracle, and other relational databases.

```
/**
 * Retrieves a description of a table's optimal set of columns that
 * uniquely identifies a row. They are ordered by SCOPE  The result
 * is returned as an XML (as a serialized string object);  if table
 * name is null/empty it returns null.
 *
 * @param conn the Connection object
 * @param catalog  a catalog name; must match the catalog name
 *      as it is stored in the database; "" retrieves those without
 *      a catalog; null means that the catalog name should not be
 *      used to narrow the search
 * @param schema a schema name; must match the schema name as it
 *      is stored in the database; "" retrieves those without a
 *      schema; null means that the schema name should not be
 *      used to narrow the search
 * @param table  a table name; must match the table name as it
 *      is stored in the database
 * @param scope  the scope of interest; possible values are:
 *
 *      bestRowTemporary - very temporary, while using row
 *      bestRowTransaction - valid for remainder of current transaction
 *      bestRowSession - valid for remainder of current session
 *
 * @param nullable  include columns that are nullable.
 * @return the result is returned as an XML (serialized as a String object)
 * @exception Failed to get the Index Information.
 */
public static String getBestRowIdentifier(java.sql.Connection conn,
                                          String catalog,
                                          String schema,
                                          String table,
                                          int scope,
                                          boolean nullable)
   throws Exception {
   ResultSet rs = null;
   try {
        if ((table == null) || (table.length() == 0)) {
            return null;
        }
```

```
            DatabaseMetaData meta = conn.getMetaData();
            if (meta == null) {
                return null;
            }

            // The '_' character represents any single character.
            // The '%' character represents any sequence of zero
            // or more characters.
            rs = meta.getBestRowIdentifier(catalog,
                                           schema,
                                           table,
                                           scope,
                                           nullable);
            StringBuilder sb = new StringBuilder("<?xml version='1.0'>");
            sb.append("<BestRowIdentifier>");
            while (rs.next()) {

                short actualScope = rs.getShort(COLUMN_NAME_SCOPE);
                String columnName = rs.getString(COLUMN_NAME_COLUMN_NAME);
                int dataType = rs.getInt(COLUMN_NAME_DATA_TYPE);
                String typeName = rs.getString(COLUMN_NAME_TYPE_NAME);
                int columnSize = rs.getInt(COLUMN_NAME_COLUMN_SIZE);
                short decimalDigits  = rs.getShort(COLUMN_NAME_DECIMAL_DIGITS);
                short pseudoColumn  = rs.getShort(COLUMN_NAME_PSEUDO_COLUMN);

                sb.append("<RowIdentifier tableName=\"");
                sb.append(table);
                sb.append("\">");
                appendXMLTag(sb, "scope", actualScope);
                appendXMLTag(sb, "columnName", columnName);
                appendXMLTag(sb, "dataType", dataType);
                appendXMLTag(sb, "typeName", typeName);
                appendXMLTag(sb, "columnSize", columnSize);
                appendXMLTag(sb, "decimalDigits", decimalDigits);
                appendXMLTag(sb, "pseudoColumn", pseudoColumn);
                sb.append("</RowIdentifier>");
            }
            sb.append("</BestRowIdentifier>");
            return sb.toString();
        }
        catch(Exception e) {
            throw new Exception("Error: could not get table's "+
                 "Best Row Identifier: "+e.toString());
        }
        finally {
            DatabaseUtil.close(rs);
        }
    }
```

Discussion of Schema and Catalog

- The MySQL database does not understand "schema"; you have to use "catalog."
- The Oracle database does not understand "catalog"; you have to use "schema."
- For databases, check their JDBC documentation.

Client Using MySQL

```
import java.util.*;
import java.io.*;
import java.sql.*;

import jcb.db.*;
import jcb.meta.*;

public class TestMySqlDatabaseMetaDataTool_BestRowIdentifier {

    public static Connection getConnection() throws Exception {
        String driver = "org.gjt.mm.mysql.Driver";
        String url = "jdbc:mysql://localhost/octopus";
        String username = "root";
        String password = "root";
        Class.forName(driver);  // load MySQL driver
        return DriverManager.getConnection(url, username, password);
    }

    public static void main(String[] args) {
        Connection conn = null;
        try {
            conn = getConnection();
            System.out.println("-------- getBestRowIdentifier ------");
            System.out.println("conn="+conn);
            String bestRowIdentifier =
                DatabaseMetaDataTool.getBestRowIdentifier
                    (conn,
                     "",                // schema
                     conn.getCatalog(), // catalog
                     "MYPICTURES",  // table name
                     DatabaseMetaData.bestRowTemporary, // scope
                     false);    // nullable
            System.out.println(bestRowIdentifier);
            System.out.println("-----------------------------------");
        }
        catch(Exception e){
            e.printStackTrace();
            System.exit(1);
        }
```

```
        finally {
            DatabaseUtil.close(conn);
        }
    }
}
```

Output Using MySQL

```
    -------- getBestRowIdentifier ------
    conn=com.mysql.jdbc.Connection@1837697
    <?xml version='1.0'>
    <BestRowIdentifier>
        <RowIdentifier tableName="MYPICTURES">
            <scope>2</scope>
            <columnName>id</columnName>
            <dataType>0</dataType>
            <typeName>(11)</typeName>
            <columnSize>11</columnSize>
            <decimalDigits>0</decimalDigits>
            <pseudoColumn>1</pseudoColumn>
        </RowIdentifier>
    </BestRowIdentifier>
    ------------------------------------
```

Oracle Database Setup

```
$ sqlplus octopus/octopus
SQL*Plus: Release 9.2.0.1.0 - Production on Thu Feb 20 17:02:51 2003
SQL> describe employees;
 Name              Null?    Type
 ----------------- -------- ----------------------------
 BADGENUMBER       NOT NULL NUMBER(38)
 NAME                       VARCHAR2(60)
 EMPLOYEETYPE               VARCHAR2(30)
 PHOTO                      BINARY FILE LOB
```

Client Using an Oracle Database

```
import java.util.*;
import java.io.*;
import java.sql.*;

import jcb.db.*;
import jcb.meta.*;

public class TestOracleDatabaseMetaDataTool_BestRowIdentifier {
```

```
    public static Connection getConnection() throws Exception {
        String driver = "oracle.jdbc.driver.OracleDriver";
        String url = "jdbc:oracle:thin:@localhost:1521:maui";
        String username = "octopus";
        String password = "octopus";
        Class.forName(driver);  // load Oracle driver
        return DriverManager.getConnection(url, username, password);
    }

    public static void main(String[] args) {
        Connection conn = null;
        try {
            conn = getConnection();
            System.out.println("-------- getBestRowIdentifier -------------");
            System.out.println("conn="+conn);
            String bestRowIdentifier =
                DatabaseMetaDataTool.getBestRowIdentifier
                    (conn,
                     "",            // schema
                     "OCTOPUS",     // user
                     "EMPLOYEES",   // table name
                     DatabaseMetaData.bestRowTransaction,   // scope
                     false);    // nullable
            System.out.println(bestRowIdentifier);
            System.out.println("-----------------------------------");
        }
        catch(Exception e){
            e.printStackTrace();
            System.exit(1);
        }
        finally {
            DatabaseUtil.close(conn);
        }
    }

}
```

Running the Solution for an Oracle Database

The following output is formatted to fit the page:

```
    -------- getBestRowIdentifier -------------
    conn=oracle.jdbc.driver.OracleConnection@169ca65
    <?xml version='1.0'>
    <BestRowIdentifier>
        <RowIdentifier tableName="EMPLOYEES">
            <scope>1</scope>
```

```
            <columnName>ROWID</columnName>
            <dataType>-8</dataType>
            <typeName>ROWID</typeName>
            <columnSize>0</columnSize>
            <decimalDigits>0</decimalDigits>
            <pseudoColumn>2</pseudoColumn>
        </RowIdentifier>
        <RowIdentifier tableName="EMPLOYEES">
            <scope>2</scope>
            <columnName>BADGENUMBER</columnName>
            <dataType>3</dataType>
            <typeName>NUMBER</typeName>
            <columnSize>22</columnSize>
            <decimalDigits>0</decimalDigits>
            <pseudoColumn>1</pseudoColumn>
        </RowIdentifier>
    </BestRowIdentifier>
    -----------------------------------
```