JDBC Metadata, MySQL, and Oracle Recipes

A Problem-Solution Approach

Mahmoud Parsian

Apress®

JDBC Metadata, MySQL, and Oracle Recipes: A Problem-Solution Approach

ISBN-13: 978-1-59090-637-1

ISBN-10: 1-59059-637-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Sumit Pal
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Managers: Beckie Brand, Elizabeth Seymour
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Lori Bring
Compositor: Linda Weideman, Wolf Creek Press
Proofreader: Dan Shaw
Indexer: Lucie Haskins
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

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.

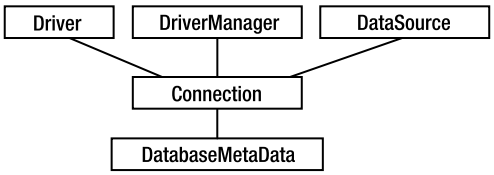

Figure 2-1. *Relationships between important interfaces and classes*

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>
-----------------------------------
```