CHAPTER 3

Database Metadata, Part 2

Those who are enamored of practice without theory are like a pilot who goes into a ship without rudder or compass and never has any certainty where he is going.

Leonardo da Vinci

This chapter continues where Chapter 2 left off and shows you how to use JDBC's database metadata API. This API lets you obtain information about tables, views, column names, column types, indexes, table and column privileges, stored procedures, result sets, and databases.

3.1. What Are a Table's Indexes?

According to Wikipedia, "an index is a feature in a database that allows quick access to the rows in a table. The index is created using one or more columns of the table." You can use the `DatabaseMetaData` interface's `getIndexInfo()` method to find the indexes for a specified table. This section illustrates how you'd use `getIndexInfo()`; we define a couple of indexes and then run our solution against a sample table called `ACCOUNT`. The signature of `getIndexInfo()` is

```
public ResultSet getIndexInfo(String catalog,
                              String schema,
                              String table,
                              boolean unique,
                              boolean approximate)
    throws SQLException;
```

The method's parameters are

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

- `table`: A table name; must match the table name as it is stored in this database.

- `unique`: When `true`, returns only indexes for unique values; when `false`, returns indexes regardless of whether or not values are unique.

- approximate: When true, the result is allowed to reflect approximate or out-of-date values; when false, the results are requested to be accurate (all table statistics are exact). Some drivers (such as the MiniSoft JDBC Driver) ignore this parameter and ensure that all table statistics are exact.

This method retrieves a ResultSet object containing information about the indexes or keys for the table. The returned ResultSet is ordered by NON_UNIQUE, TYPE, INDEX_NAME, and ORDINAL_POSITION. Each index column description has the columns shown in Table 3-1 (each row has 13 columns).

Table 3-1. *Result Columns for Invoking getIndexInfo()*

Field Name	Type	Description
TABLE_CAT	String	Table catalog (may be null).
TABLE_SCHEM	String	Table schema (may be null).
TABLE_NAME	String	Table name.
NON_UNIQUE	boolean	Indicates whether index values can be non-unique. false when TYPE is tableIndexStatistic.
INDEX_QUALIFIER	String	Index catalog (may be null); null when TYPE is tableIndexStatistic.
INDEX_NAME	String	Index name; null when TYPE is tableIndexStatistic.
TYPE	short	Index type: tableIndexStatistic: Identifies table statistics that are returned in conjunction with a table's index descriptions tableIndexClustered: Is a clustered index tableIndexHashed: Is a hashed index tableIndexOther: Is some other style of index
ORDINAL_POSITION	short	Column sequence number within the index; zero when TYPE is tableIndexStatistic.
COLUMN_NAME	String	Column name; null when TYPE is tableIndexStatistic.
ASC_OR_DESC	String	Column sort sequence. A means ascending; D means descending; may be null if sort sequence is not supported; null when TYPE is tableIndexStatistic.
CARDINALITY	int	When TYPE is tableIndexStatistic, then this is the number of rows in the table; otherwise, it is the number of unique values in the index.
PAGES	int	When TYPE is tableIndexStatistic, then this is the number of pages used for the table, otherwise it is the number of pages used for the current index.
FILTER_CONDITION	String	Filter condition, if any (may be null).

This method returns ResultSet, in which each row is an index column description. If a database access error occurs, it throws SQLException.

As you can see from the returned ResultSet, it contains a lot of information. The best way to represent that information is XML, which may be used by any type of client.

The Solution: getIndexInformation()

The index information can be useful in sending proper SQL queries to the database. During runtime, for better response from the database, in formulating SQL's SELECT statement you can use the index columns in the WHERE clauses (otherwise, the database tables will be scanned sequentially). In passing actual parameters to the DatabaseMetaData.getIndexInfo() method, try to minimize passing null and empty values (passing null values might slow down your metadata retrieval).

```
/**
 * Retrieves a description of the given table's indexes and
 * statistics.  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 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 unique when true, return only indexes for unique values;
 * when false, return indexes regardless of whether unique or not
 * @param approximate when true, result is allowed to reflect
 * approximate or out of data values; when false, results are
 * requested to be accurate
 * @return an XML.
 * @exception Failed to get the Index Information.
 */
 public static String getIndexInformation(java.sql.Connection conn,
                                          String catalog,
                                          String schema,
                                          String tableName,
                                          boolean unique,
                                          boolean approximate)
    throws Exception {
    ResultSet rs = null;
    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.
        rs = meta.getIndexInfo(catalog, schema, tableName,
                               unique, approximate);
        StringBuffer sb = new StringBuffer("<?xml version='1.0'>");
        sb.append("<indexInformation>");
        while (indexInformation.next()) {
           String dbCatalog = rs.getString(COLUMN_NAME_TABLE_CATALOG);
           String dbSchema = rs.getString(COLUMN_NAME_TABLE_SCHEMA);
           String dbTableName = rs.getString(COLUMN_NAME_TABLE_NAME);
           boolean dbNoneUnique =rs.getBoolean(COLUMN_NAME_NON_UNIQUE);
           String dbIndexQualifier = rs.getString(COLUMN_NAME_INDEX_QUALIFIER);
           String dbIndexName = rs.getString(COLUMN_NAME_INDEX_NAME);
           short dbType = rs.getShort(COLUMN_NAME_TYPE);
           short dbOrdinalPosition = rs.getShort(COLUMN_NAME_ORDINAL_POSITION);
           String dbColumnName = rs.getString(COLUMN_NAME_COLUMN_NAME);
           String dbAscOrDesc = rs.getString(COLUMN_NAME_ASC_OR_DESC);
           int dbCardinality = rs.getInt(COLUMN_NAME_CARDINALITY);
           int dbPages = rs.getInt(COLUMN_NAME_PAGES);
           String dbFilterCondition = rs.getString(COLUMN_NAME_FILTER_CONDITION);
           sb.append("<index name=\"");
           sb.append(dbIndexName);
           sb.append("\" table=\"");
           sb.append(dbTableName);
           sb.append("\" column=\"");
           sb.append(dbColumnName);
           sb.append("\">");
           appendXMLTag(sb, "catalog", dbCatalog);
           appendXMLTag(sb, "schema", dbSchema);
           appendXMLTag(sb, "nonUnique", dbNoneUnique);
           appendXMLTag(sb, "indexQualifier", dbIndexQualifier);
           appendXMLTag(sb, "type", dbType);
           appendXMLTag(sb, "ordinalPosition", dbOrdinalPosition);
           appendXMLTag(sb, "ascendingOrDescending", dbAscOrDesc);
           appendXMLTag(sb, "cardinality", dbCardinality);
           appendXMLTag(sb, "pages", dbPages);
           appendXMLTag(sb, "filterCondition", dbFilterCondition);
           sb.append("</index>");
        }
        sb.append("</indexInformation>");
            return sb.toString();
        }
        catch(Exception e) {
            throw new Exception("could not get table's Index Info: "+e.toString());
        }
        finally {
            DatabaseUtil.close(rs);
        }
   }
```

Oracle Database Setup

For testing, let's create an ACCOUNT table and a couple of indexes:

```
$ sqlplus octopus/octopus
SQL*Plus: Release 9.2.0.1.0 - Production on Sat Feb 15 18:07:05 2003

SQL> create table ACCOUNT(
  2     id  varchar(20) not null primary key,
  3     owner varchar(60) not null,
  4     balance number,
  5     status  varchar(10));

Table created.

SQL> describe ACCOUNT;
 Name             Null?    Type
 ---------------- -------- ------------
 ID               NOT NULL VARCHAR2(20)
 OWNER            NOT NULL VARCHAR2(60)
 BALANCE                   NUMBER
 STATUS                    VARCHAR2(10)
```

Next, let's create some indexes on the ACCOUNT table using the Oracle database. Since id is a primary key, Oracle will automatically create a unique index for this column; we define three additional indexes.

```
SQL> create index ID_OWNER_INDEX on ACCOUNT(id, owner);
SQL> create index ID_STATUS_INDEX on ACCOUNT(id, status);
SQL> create unique index OWNER_INDEX on ACCOUNT(owner);
SQL> commit;
Commit complete.
```

Client 1: Oracle

```
System.out.println("-------- getIndexInformation -------------");
String indexInformation = DatabaseMetaDataTool.getIndexInformation
                     (conn,
                     "",
                     "OCTOPUS",      // user
                     "ACCOUNT",      // table name
                     true,           // unique indexes?
                     true);
System.out.println("-------- getIndexInformation -------------");
System.out.println(indexInformation);
System.out.println("------------------------------------");
```

Output 1: Oracle

Note that when the index type is tableIndexStatistic, the index name will be null. When the schema does not assign a proper index name (for example, for primary keys), the database server will assign a generated name.

```
<?xml version='1.0'>
<indexInformation>
    <index name="null" table="ACCOUNT" column="null">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexStatistic</type>
        <ordinalPosition>0</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="OWNER_INDEX" table="ACCOUNT" column="OWNER">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexClustered</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="SYS_C003011" table="ACCOUNT" column="ID">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexClustered</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
</indexInformation>
```

Client 2: Oracle

```
    System.out.println("-------- getIndexInformation -------------");
    String indexInformation = DatabaseMetaDataTool.getIndexInformation
                         (conn,
                         "",
                         "OCTOPUS",      // user
                         "ACCOUNT",      // table name
                         false,          // unique indexes?
                         true);
    System.out.println("-------- getIndexInformation -------------");
    System.out.println(indexInformation);
    System.out.println("------------------------------------");
```

Output 2: Oracle

```
<?xml version='1.0'>
<indexInformation>
    <index name="null" table="ACCOUNT" column="null">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexStatistic</type>
        <ordinalPosition>0</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="OWNER_INDEX" table="ACCOUNT" column="OWNER">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexClustered</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
        </index>
    <index name="SYS_C003011" table="ACCOUNT" column="ID">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexClustered</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_OWNER_INDEX" table="ACCOUNT" column="ID">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexClustered</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_OWNER_INDEX" table="ACCOUNT" column="OWNER">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexClustered</type>
        <ordinalPosition>2</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_STATUS_INDEX" table="ACCOUNT" column="ID">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexClustered</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_STATUS_INDEX" table="ACCOUNT" column="STATUS">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier>null</indexQualifier>
        <type>tableIndexClustered</type>
        <ordinalPosition>2</ordinalPosition>
        <ascendingOrDescending>null</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
</indexInformation>
```

MySQL Database Setup

For testing, let's create an ACCOUNT table and a couple of indexes:

```
mysql> create table ACCOUNT( id  varchar(20) not null primary key,
    -> owner varchar(60) not null,
    -> balance integer, status  varchar(10));
Query OK, 0 rows affected (0.10 sec)
mysql> describe ACCOUNT;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| id      | varchar(20) |      | PRI |         |       |
| owner   | varchar(60) |      |     |         |       |
| balance | int(11)     | YES  |     | NULL    |       |
| status  | varchar(10) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
4 rows in set (0.05 sec)
```

Next, let's create some indexes on the ACCOUNT table using the MySQL database. Because id is a primary key, MySQL will automatically create a unique index for this column; we define three additional indexes.

```
    mysql> create index ID_OWNER_INDEX on ACCOUNT(id, owner);
    Query OK, 0 rows affected (0.29 sec)
    Records: 0  Duplicates: 0  Warnings: 0
    mysql>  create index ID_STATUS_INDEX on ACCOUNT(id, status);
    Query OK, 0 rows affected (0.29 sec)
    Records: 0  Duplicates: 0  Warnings: 0
    mysql> create unique index OWNER_INDEX on ACCOUNT(owner);
    Query OK, 0 rows affected (0.26 sec)
    Records: 0  Duplicates: 0  Warnings: 0
    mysql> commit;
    Query OK, 0 rows affected (0.00 sec)
    mysql>  describe ACCOUNT;
```

```
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| id      | varchar(20) |      | PRI |         |       |
| owner   | varchar(60) |      | UNI |         |       |
| balance | int(11)     | YES  |     | NULL    |       |
| status  | varchar(10) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
4 rows in set (0.02 sec)
```

Client 1: MySQL

```
System.out.println("-------- getIndexInformation -------------");
String indexInformation = DatabaseMetaDataTool.getIndexInformation
                     (conn,
                      conn.getCatalog(),
                      null,           // MySQL has no schema
                      "ACCOUNT",      // table name
                      true,           // unique indexes?
                      true);
System.out.println("-------- getIndexInformation -------------");
System.out.println(indexInformation);
System.out.println("------------------------------------------");
```

Output 1: MySQL

```
<?xml version='1.0'>
<indexInformation>
    <index name="PRIMARY" table="ACCOUNT" column="id">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="OWNER_INDEX" table="ACCOUNT" column="owner">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_OWNER_INDEX" table="ACCOUNT" column="id">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_OWNER_INDEX" table="ACCOUNT" column="owner">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>2</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_STATUS_INDEX" table="ACCOUNT" column="id">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_STATUS_INDEX" table="ACCOUNT" column="status">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>2</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
</indexInformation>
```

Client 2: MySQL

```
    System.out.println("-------- getIndexInformation -------------");
    String indexInformation = DatabaseMetaDataTool.getIndexInformation
                         (conn,
                          conn.getCatalog(),
                          null,            // MySQL has no schema
                          "ACCOUNT",       // table name
                          false,           // unique indexes?
                          true);
    System.out.println("-------- getIndexInformation -------------");
    System.out.println(indexInformation);
    System.out.println("------------------------------------");
```

Output 2: MySQL

```
<?xml version='1.0'>
<indexInformation>
    <index name="PRIMARY" table="ACCOUNT" column="id">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="OWNER_INDEX" table="ACCOUNT" column="owner">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>false</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_OWNER_INDEX" table="ACCOUNT" column="id">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_OWNER_INDEX" table="ACCOUNT" column="owner">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>2</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_STATUS_INDEX" table="ACCOUNT" column="id">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>1</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
    <index name="ID_STATUS_INDEX" table="ACCOUNT" column="status">
        <catalog>tiger</catalog>
        <schema>null</schema>
        <nonUnique>true</nonUnique>
        <indexQualifier></indexQualifier>
        <type>tableIndexOther</type>
        <ordinalPosition>2</ordinalPosition>
        <ascendingOrDescending>A</ascendingOrDescending>
        <cardinality>0</cardinality>
        <pages>0</pages>
        <filterCondition>null</filterCondition>
    </index>
</indexInformation>
```

3.2. Does an Index Exist for a Specific Table?

Given a table, such as the ACCOUNT table created in the previous section, you can find out whether a particular index exists. There is no such explicit method in the JDBC API, but you can use the DatabaseMetaData.getIndexInfo() method in the solution to solve the problem.

The Solution: indexExists()

```
public static boolean indexExists(java.sql.Connection conn,
                                  String catalog,
                                  String schema,
                                  String tableName,
                                  String indexName)
     throws Exception {
     if ((tableName == null) || (tableName.length() == 0) ||
         (indexName == null) || (indexName.length() == 0)) {
          return false;
     }

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

     ResultSet rs = dbMetaData.getIndexInfo(catalog,
                    schema, tableName, false, true);
     while (rs.next()) {
         String dbIndexName = rs.getString(COLUMN_NAME_INDEX_NAME);
         if (indexName.equalsIgnoreCase(dbIndexName)) {
           return true;
         }
     }
     return false;
}
```

A Client: MySQL

```
    System.out.println("-------- Does index exist? -------------");
    System.out.println("conn="+conn);
    boolean indexExist = DatabaseMetaDataTool.indexExists
                (conn,
                 conn.getCatalog(),     // catalog
                 null,                  // schema
                 "ACCOUNT",             // table name
                 "ID_STATUS_INDEX");    // index name
    System.out.println("Index name: ID_STATUS_INDEX");
    System.out.println("Table name: ACCOUNT");
    System.out.println("Index Exist?: " + indexExist);

    System.out.println("-------- Does index exist? -------------");
    boolean indexExist22 = DatabaseMetaDataTool.indexExists
                (conn,
                 conn.getCatalog(),     // catalog
                 null,                  // schema
                 "ACCOUNT",         // table name
                 "ID_STATUS_INDEX22");  // index name
    System.out.println("Index name: ID_STATUS_INDEX22");
    System.out.println("Table name: ACCOUNT");
    System.out.println("Index Exist?: " + indexExist22);
```

Output: MySQL

```
    -------- Does index exist? -------------
    conn=com.mysql.jdbc.Connection@337d0f
    Index name: ID_STATUS_INDEX
    Table name: ACCOUNT
    Index Exist?: true
    -------- Does index exist? -------------
    Index name: ID_STATUS_INDEX22
    Table name: ACCOUNT
    Index Exist?: false
```

A Client: Oracle

```
    System.out.println("-------- Does index exist? -------------");
    System.out.println("conn="+conn);
    boolean indexExist = DatabaseMetaDataTool.indexExists
                (conn,
                 conn.getCatalog(),     // catalog
                 null,                  // schema
                 "ACCOUNT",             // table name
                 "ID_STATUS_INDEX");    // index name
    System.out.println("Index name: ID_STATUS_INDEX");
    System.out.println("Table name: ACCOUNT");
    System.out.println("Index Exist?: " + indexExist);

    System.out.println("-------- Does index exist? -------------");
    boolean indexExist22 = DatabaseMetaDataTool.indexExists
                (conn,
                 conn.getCatalog(),     // catalog
                 null,                  // schema
                 "ACCOUNT",         // table name
                 "ID_STATUS_INDEX22");  // index name
    System.out.println("Index name: ID_STATUS_INDEX22");
    System.out.println("Table name: ACCOUNT");
    System.out.println("Index Exist?: " + indexExist22);
```

Output: Oracle

```
    -------- Does index exist? -------------
    conn=oracle.jdbc.driver.OracleConnection@d0a5d9
    Index name: ID_STATUS_INDEX
    Table name: ACCOUNT
    Index Exist?: true
    -------- Does index exist? -------------
    Index name: ID_STATUS_INDEX22
    Table name: ACCOUNT
    Index Exist?: false
```

3.3. What Are the Names of a Database's Stored Procedures?

In a relational database management system such as Oracle, a *stored procedure* is a precompiled set of SQL statements and queries that can be shared by a number of programs. It is stored under a name as an executable unit. A *stored function* is similar to a function (like in Java and C/C++); it accepts zero, one, or more parameters and returns a single result.

Stored procedures and functions are helpful in the following ways:

- **Controlling access to data**: They can restrict client programs to data accessible only through the stored procedure.
- **Preserving data integrity**: They ensure that information is entered in a consistent manner.
- **Improving productivity**: You need to write a stored procedure only once.

Oracle, Microsoft SQL Server 2000, and Sybase Adaptive Server support stored procedures, but MySQL does not (stored procedures and views will be supported in MySQL 5.0.1, however). In general, you can use stored procedures to maximize security and increase data access efficiency. Because stored procedures execute in the database server, they minimize the network traffic between applications and the database, increasing application and system performance. Most of the time, stored procedures run faster than SQL. They also allow you to isolate your SQL code from your application.

Using Oracle9i database, consider the following table:

```
SQL> describe zemps;
```

```
 Name                                  Null?     Type
 ------------------------------------  --------  -------------
 ID                                    NOT NULL  NUMBER(38)
 FIRSTNAME                             NOT NULL  VARCHAR2(32)
 LASTNAME                              NOT NULL  VARCHAR2(32)
 DEPT                                  NOT NULL  VARCHAR2(32)
 TITLE                                           VARCHAR2(32)
 SALARY                                          NUMBER(38)
 EMAIL                                           VARCHAR2(64)
 COUNTRY                                         VARCHAR2(32)
```

Next, try a basic query of the zemps table:

```
SQL> select id, firstName, lastName from zemps;
        ID   FIRSTNAME                LASTNAME
----------   ----------               -----------
      4401   Donald                   Knuth
      4402   Charles                  Barkeley
      4403   Alex                     Badame
      4404   Jeff                     Torrango
      4405   Mary                     Smith
      4406   Alex                     Sitraka
      4408   Jessica                  Clinton
      4409   Betty                    Dillon
      5501   Troy                     Briggs
      5502   Barb                     Tayloy
      6601   Pedro                    Hayward
      6602   Chris                    Appleseed
      6603   Tao                      Yang
      6604   Kelvin                   Liu

14 rows selected.
```

The following stored procedure, getEmpCount, returns the number of records in the zemps table:

```
SQL> CREATE OR REPLACE function getEmpCount return int is
  2     empCount int;
  3  BEGIN
  4     SELECT count(*) INTO empCount FROM zEmps;
  5     RETURN empCount;
  6  END getEmpCount;
  7
  8
  9  /

Function created.
```

In order to make sure that getEmpCount is created correctly, you can execute it as follows, without passing any parameters:

```
SQL> var empCount number;
SQL> exec :empCount := getEmpCount;
PL/SQL procedure successfully completed.
SQL> print empCount;
   EMPCOUNT
----------
        14
```

The output proves that the getEmpCount performed correctly because it returned 14, which is the total number of records in the zemps table.

Overloading Stored Procedures

Oracle's PL/SQL allows two or more packaged subprograms to have the same name. A *package* is a set of logically related functions and procedures, also known as a stored procedure. When stored procedures have the same name but different parameters, this is called *overloading*. This option is useful when you want a subprogram or function to accept parameters that have different data types. Be very cautious when you call overloaded subprogram or functions. You must make sure that you are passing the expected number of arguments and data types. For example, in Oracle 9i, the following package defines two functions named empPackage.

Oracle's package specification is as follows:

```
CREATE or REPLACE PACKAGE empPackage AS
    FUNCTION getEmployeeID(eFirstName VARCHAR2) return INT;
    FUNCTION getEmployeeID(eFirstName VARCHAR2, eLastName VARCHAR2) return INT;
END empPackage;
```

Oracle's package implementation is as follows:

```
CREATE or REPLACE PACKAGE BODY empPackage AS
    FUNCTION getEmployeeID (eFirstName VARCHAR2) return INT is
       empID INT;
    BEGIN
       SELECT id INTO empID FROM zEmps where firstName = eFirstName;
       RETURN empID;
    END getEmployeeID;
    FUNCTION getEmployeeID (eFirstName VARCHAR2, eLastName VARCHAR2) return INT is
       empID INT;
    BEGIN
       SELECT id INTO empID FROM zEmps
              where firstName = eFirstName and lastName = eLastName;
       RETURN empID;
    END getEmployeeID;
END empPackage;
```

Here's the empPackage description from the database:

```
SQL> describe empPackage;
FUNCTION GETEMPLOYEEID RETURNS NUMBER(38)
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 EFIRSTNAME                     VARCHAR2                IN
FUNCTION GETEMPLOYEEID RETURNS NUMBER(38)
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 EFIRSTNAME                     VARCHAR2                IN
 ELASTNAME                      VARCHAR2                IN
```

Now execute these two functions:

```
SQL> var id1 NUMBER;
SQL> exec :id1:= empPackage.getEmployeeID('Donald');
PL/SQL procedure successfully completed.
SQL> print id1;
       ID1
----------
      4401
SQL> var id2 NUMBER;
SQL> exec :id2:= empPackage.getEmployeeID('Betty', 'Dillon');
PL/SQL procedure successfully completed.
SQL> print id2;
       ID2
----------
      4409
```

■ **Note** You may be wondering what this discussion has to do with getting the names of the stored procedures. This is because stored procedure names can be overloaded, and so you must be very careful in selecting the stored procedure names and their associated input parameter types.

How Can You Find the Package Code in Oracle?

The following SQL statement provides a way to see the Oracle package code:

```
select LINE, TEXT
    from USER_SOURCE
        where NAME ='&PKG' and TYPE = '&PACKAGE_TYPE'
```

where:

- `PKG` refers to the package name.
- `PACKAGE_TYPE` is the `PACKAGE` for the package specification.
- `PACKAGE BODY` displays the body.

What Is the user_source Table?

The `user_source` table, which is a property of Oracle's SYS user, is as follows. The output has been modified to include a description column.

```
SQL> describe user_source;
 Name  Type            Description
 ----  --------------- ----------------------------
 NAME  VARCHAR2(30)    Name of the object
 TYPE  VARCHAR2(12)    Type of the object: "TYPE", "TYPE BODY",
                       "PROCEDURE", "FUNCTION", "PACKAGE",
                       "PACKAGE BODY" or "JAVA SOURCE"'
 LINE  NUMBER          Line number of this line of source
 TEXT  VARCHAR2(4000)  Source text

SQL> select name, type, line from user_source;

NAME                           TYPE               LINE
------------------------------ ------------ ----------
EMPPACKAGE                     PACKAGE               1
EMPPACKAGE                     PACKAGE               2
EMPPACKAGE                     PACKAGE               3
EMPPACKAGE                     PACKAGE               4
EMPPACKAGE                     PACKAGE BODY          1
EMPPACKAGE                     PACKAGE BODY          2
EMPPACKAGE                     PACKAGE BODY          3
EMPPACKAGE                     PACKAGE BODY          4
EMPPACKAGE                     PACKAGE BODY          5
EMPPACKAGE                     PACKAGE BODY          6
EMPPACKAGE                     PACKAGE BODY          7
EMPPACKAGE                     PACKAGE BODY          8
EMPPACKAGE                     PACKAGE BODY          9
EMPPACKAGE                     PACKAGE BODY         10
EMPPACKAGE                     PACKAGE BODY         11
EMPPACKAGE                     PACKAGE BODY        12
EMPPACKAGE                     PACKAGE BODY        13
EMPPACKAGE                     PACKAGE BODY        14
EMPPACKAGE                     PACKAGE BODY        15
GETEMPCOUNT                    FUNCTION             1
GETEMPCOUNT                    FUNCTION             2
GETEMPCOUNT                    FUNCTION             3
GETEMPCOUNT                    FUNCTION             4
GETEMPCOUNT                    FUNCTION             5
GETEMPCOUNT                    FUNCTION             6
GETEMPCOUNT                    FUNCTION             7

26 rows selected.
```

What Are the Names of a Database's Stored Procedures?

In the JDBC API, you can use the DatabaseMetaData.getProcedures() method to get the names of a database's stored procedures and functions. However, this is not sufficient for very large databases. For example, in an Oracle database, DatabaseMetadata.getProcedures() can return hundreds of stored procedures; most are system stored procedures, which most likely you do not need to retrieve. When you call this method, be as specific as possible when you provide names and patterns.

JDBC Solution: getProcedures()

Using JDBC, you can use DatabaseMetaData.getProcedures() to retrieve stored procedure names: To have a better performance, try to pass as much as information you can and avoid passing empty and null values to the DatabaseMetaData.getProcedures() method. Passing empty and null values might have a poor performance, and this is due to the fact that it might search all database catalogs and schemas. Therefore, it is best to pass as much as information (actual parameter values) to the DatabaseMetaData.getProcedures() method.

```
/**
 * Get the stored procedures names.
 * @param conn the Connection object
 * @return a table of stored procedures names
 * as an XML document (represented as a String object).
 * Each element of XML document will have the name and
 * type of a stored procedure.
 *
 */
public static String getStoredProcedureNames
    (java.sql.Connection conn,
     String catalog,
     String schemaPattern,
     String procedureNamePattern) throws Exception {
    ResultSet rs = null;
    try {
        DatabaseMetaData meta = conn.getMetaData();
        if (meta == null) {
             return null;
        }

        rs = meta.getProcedures(catalog, schemaPattern, procedureNamePattern);
        StringBuffer sb = new StringBuffer();
        sb.append("<storedProcedures>");

        while (rs.next()) {
           String spName = rs.getString("PROCEDURE_NAME");
           String spType = getStoredProcedureType(rs.getInt("PROCEDURE_TYPE"));
           sb.append("<storedProcedure name=\"");
           sb.append(spName);
           sb.append("\" type=\"");
           sb.append(spType);
           sb.append("\"/>");
        }
        sb.append("</storedProcedures>");
        return sb.toString();
    }
    finally {
       DatabaseUtil.close(rs);
    }
}

private static String getStoredProcedureType(int spType) {
    if (spType == DatabaseMetaData.procedureReturnsResult) {
        return STORED_PROCEDURE_RETURNS_RESULT;
    }
    else if (spType == DatabaseMetaData.procedureNoResult) {
        return STORED_PROCEDURE_NO_RESULT;
    }
    else {
        return STORED_PROCEDURE_RESULT_UNKNOWN;
    }
}
```

A Client Program

Before invoking a client program, let's add another stored function: the `getEmployeeCount` stored function returns the number of employees for a specific department.

```
SQL> create FUNCTION getEmployeeCount(dept INTEGER) RETURN INTEGER IS
  2     empCount INTEGER;
  3  BEGIN
  4     SELECT count(*) INTO empCount FROM EMPLOYEE
  5             WHERE deptNumber = dept;
  6     RETURN empCount;
  7  END getEmployeeCount;
  8  /

Function created.

SQL> describe getEmployeeCount;
FUNCTION getEmployeeCount RETURNS NUMBER(38)
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 DEPT                           NUMBER(38)              IN

SQL> var empCount number;
SQL> exec :empCount := getEmployeeCount(23)

PL/SQL procedure successfully completed.

SQL> print empCount;

  EMPCOUNT
----------
         3
```

A Client Program

```
    String spNames = DatabaseMetaDataTool.getStoredProcedureNames
                         (conn,
                          "",
                          "OCTOPUS",
                          "%");
    System.out.println("-------- getStoredProcedureNames -------------");
    System.out.println(spNames);
    System.out.println("------------------------------------");
```

Output of the Client Program

```
<storedProcedures>
    <storedProcedure name="GETEMPLOYEECOUNT" type="procedureReturnsResult"/>
    <storedProcedure name="RAISESALARY" type="procedureNoResult"/>
    <storedProcedure name="SHOWUSERS" type="procedureNoResult"/>
</storedProcedures>
```

3.4. What Is the Signature of a Stored Procedure?

How can a client investigate the parameters to send into and receive from a database stored procedure? Understanding the signature of a stored procedure is important for SQL adapter development in order to obtain the signature information at runtime. A signature is the name of the procedure and the name and type of its arguments. The `DatabaseMetaData` interface provides a method, `getProcedureColumns()`, which returns detailed metadata information on arguments (columns) of stored procedures. This section provides a few tables and stored procedures that will help you understand how best to use the `getProcedureColumns()` method.

The MySQL database does not support stored procedures yet, but it will in future releases (starting with MySQL 5.0.1). We'll focus here on the Oracle database. We'll define a table, called `EMPLOYEE`, and a stored procedure, `raiseSalary`, to retrieve the salary of a specific department as a percentage.

Oracle Database Setup

```
SQL> create table EMPLOYEE (
  2     badgeNumber number(4) not null,
  3     empName varchar2(40) not null,
  4     jobTitle varchar2(30),
  5     manager number(4),
  6     hireDate date,
  7     salary number(7,2),
  8     deptNumber number(2)
  9  );

Table created.

SQL> describe employee;
 Name             Null?    Type
 ---------------- -------- -------------
 BADGENUMBER      NOT NULL NUMBER(4)
 EMPNAME          NOT NULL VARCHAR2(40)
 JOBTITLE                  VARCHAR2(30)
 MANAGER                   NUMBER(4)
 HIREDATE                  DATE
 SALARY                    NUMBER(7,2)
 DEPTNUMBER                NUMBER(2)
```

Next, let's insert some records into an `EMPLOYEE` table:

```
SQL> insert into EMPLOYEE(badgeNumber , empName, jobTitle, hireDate,
  2  salary, deptNumber)
  3  values(1111, 'Alex Smith', 'Manager', '12-JAN-1981', 78000.00, 23);

SQL> insert into EMPLOYEE(badgeNumber , empName, jobTitle, manager,
  2  hireDate, salary, deptNumber)
  3  values(2222, 'Jane Taylor', 'Engineer', 1111, '12-DEC-1988', 65000.00, 23);
```

```
SQL> insert into EMPLOYEE(badgeNumber , empName, jobTitle, manager,
  2  hireDate, salary, deptNumber)
  3  values(3333, 'Art Karpov', 'Engineer', 1111, '12-DEC-1978', 80000.00, 23);

SQL> insert into EMPLOYEE(badgeNumber , empName, jobTitle, manager,
  2  hireDate, salary, deptNumber)
  3  values(4444, 'Bob Price', 'Engineer', 1111, '12-DEC-1979', 70000.00, 55);

SQL> commit;
Commit complete.

SQL> select badgeNumber, empName, salary, deptNumber  from employee;

BADGENUMBER  EMPNAME             SALARY  DEPTNUMBER
-----------  --------------- ----------  ----------
       1111  Alex Smith           78000          23
       2222  Jane Taylor          65000          23
       3333  Art Karpov           80000          23
       4444  Bob Price            70000          55
```

Next, let's create a stored procedure called raiseSalary:

```
SQL> create procedure raiseSalary(deptNumber_Param number,
  2                              percentage_Param number DEFAULT 0.20) is
  3     cursor empCursor (dept_number number) is
  4             select salary from EMPLOYEE where deptNumber = dept_number
  5                     for update of salary;
  6
  7     empsal number(8);
  8  begin
  9     open empCursor(deptNumber_Param);
 10     loop
 11             fetch empCursor into empsal;
 12             exit when empCursor%NOTFOUND;
 13             update EMPLOYEE set salary = empsal * ((100 + percentage_Param)/100)
 14                     where current of empCursor;
 15     end loop;
 16     close empCursor;
 17     commit;
 18  end raisesalary;
 19  /

Procedure created.
```

```
SQL> describe raiseSalary;
PROCEDURE raiseSalary
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 DEPTNUMBER_PARAM                NUMBER                  IN
 PERCENTAGE_PARAM                NUMBER                  IN     DEFAULT
```

Invoking/Executing raiseSalary As a Stored Procedure

In order to raise the salary of all employees in department number 23, run `raiseSalary` as follows:

```
SQL> execute raiseSalary(23, 10);
PL/SQL procedure successfully completed.

SQL> select badgeNumber, empName, salary, deptNumber  from employee;

BADGENUMBER EMPNAME             SALARY DEPTNUMBER
----------- --------------- ---------- ----------
       1111 Alex Smith           85800         23
       2222 Jane Taylor          71500         23
       3333 Art Karpov           88000         23
       4444 Bob Price            70000         55
```

The Solution: getStoredProcedureSignature()

The `getStoredProcedureSignature()` method retrieves the signature of a stored procedure and returns the metadata as an XML object, serialized as a `String` object for efficiency purposes. Here is the signature of `getStoredProcedureSignature()`:

```
    /**
     * Retrieves a description of the given catalog's stored
     * procedure parameter and result columns. This method
     * calls getProcedureColumns() to get the signature
     * and then transforms the result set into XML.
     *
     * @param conn the Connection object
     * @param catalog a catalog.
     * @param schemaPattern a schema pattern.
     * @param procedureNamePattern name of a stored procedure
     * @param columnNamePattern a column name pattern.
     * @return an XML.
     * @throws Exception Failed to get the stored procedure's signature.
     */
    public static String getStoredProcedureSignature(
            java.sql.Connection conn,
            String catalog,
            String schemaPattern,
            String procedureNamePattern,
            String columnNamePattern)

        throws Exception {...}
```

Oracle9i Considerations for the getProcedureColumns() Method

Inside our solution, `getStoredProcedureSignature()`, we call `getProcedureColumns()`, to which we have to give special consideration. According to Oracle, the methods `getProcedures()` and `getProcedureColumns()` (defined in the `DatabaseMetaData` interface) treat the `catalog`, `schemaPattern`, `columnNamePattern`, and `procedureNamePattern` parameters in the same way. In the Oracle definition of these methods, the parameters are treated differently. Table 3-2 is taken from the Oracle 9i documentation.

Table 3-2. *The getProcedureColumns() Method According to Oracle*

Field Name	Description
`catalog`	Oracle does not have multiple catalogs, but it does have packages. Consequently, the `catalog` parameter is treated as the package name. This applies both on input (the catalog parameter) and output (the catalog column in the returned `ResultSet`). On input, the construct "" (the empty string) retrieves procedures and arguments without a package, that is, standalone objects. A `null` value means to drop from the selection criteria, that is, return information about both standalone and packaged objects (same as passing in "%"). Otherwise, the `catalog` parameter should be a package name pattern (with SQL wildcards, if desired).
`schemaPattern`	All objects within Oracle must have a schema, so it does not make sense to return information for those objects without one. Thus, the construct "" (the empty string) is interpreted on input to mean the objects in the current schema (that is, the one to which you are currently connected). To be consistent with the behavior of the `catalog` parameter, `null` is interpreted to drop the schema from the selection criteria (same as passing in "%"). It can also be used as a pattern with SQL wildcards.
`procedureNamePattern`	The empty string ("") does not make sense for either parameter, because all procedures and arguments must have names. Thus, the construct "" will raise an exception. To be consistent with the behavior of other parameters, `null` has the same effect as passing in "%".
`columnNamePattern`	The empty string ("") does not make sense for either parameter, because all procedures and arguments must have names. Thus, the construct "" will raise an exception. To be consistent with the behavior of other parameters, `null` has the same effect as passing in "%".

A Weakness for the JDBC Metadata

Before we delve into the signature of this method, let's look at a weakness of the `getProcedureColumns()` method: inside `getStoredProcedureSignature()`, we use the method `getProcedureColumns()` in the interface `DatabaseMetaData` to obtain a stored procedure's metadata. The exact usage is described in the code that follows. You should note that this method (getProcedureColumns()) can only discover *parameter* values. Some databases (such as Sybase and Microsoft's SQL Server 2000) can return multiple result sets without using any arguments. For databases where a returning ResultSet is created simply by executing a SQL SELECT statement within a stored procedure (thus not sending the return ResultSet to the client application via a declared parameter), the real return value of the stored procedure cannot be detected. This is a weakness for the JDBC metadata.

Signature of getProcedureColumns()

The getProcedureColumns() method's signature is defined in JDK1.4.2 as follows:

```
public ResultSet getProcedureColumns
   (String catalog,
    String schemaPattern,
    String procedureNamePattern, // in Oracle it must be uppercase
    String columnNamePattern)
throws SQLException
```

This method retrieves a description of the given catalog's stored procedure parameter and result columns. Only descriptions matching the schema, procedure, and parameter name criteria are returned. They are ordered by PROCEDURE_SCHEM and PROCEDURE_NAME. Within this, the return value, if any, is first. Next are the parameter descriptions in call order. The column descriptions follow in column number order.

Each row in the ResultSet is a parameter or column description with the fields shown in Table 3-3.

Table 3-3. *Parameter or Column Description Fields*

Field Name	Type	Description
PROCEDURE_CAT	String	The procedure catalog (may be null).
PROCEDURE_SCHEM	String	The procedure schema (may be null).
PROCEDURE_NAME	String	The procedure name.
COLUMN_NAME	String	The column/parameter name.
COLUMN_TYPE	Short	The kind of column or parameter: procedureColumnUnknown: Unknown procedureColumnIn: The IN parameter procedureColumnInOut: The INOUT parameter procedureColumnOut: The OUT parameter procedureColumnReturn: The procedure's return value procedureColumnResult: The result column in ResultSet
DATA_TYPE	int	SQL type from java.sql.Types
TYPE_NAME	String	SQL type name; for a UDT type, the type name is fully qualified.
PRECISION	int	Precision.
LENGTH	int	The length in bytes of data.
SCALE	short	The scale.
RADIX	short	The radix.

Field Name	Type	Description
NULLABLE	short	Specifies whether it can contain NULL: procedureNoNulls: Does not allow NULL values procedureNullable: Allows NULL values procedureNullableUnknown: Nullability unknown
REMARKS	String	A comment describing the parameter or column.

Note Some databases may not return the column descriptions for a procedure. Additional columns beyond REMARKS can be defined by the database.

The parameters for this method are as follows:

- 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.
- schemaPattern: A schema name pattern; 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.
- procedureNamePattern: A procedure name pattern; it must match the procedure name as it is stored in the database.
- columnNamePattern: A column name pattern; it must match the column name as it is stored in the database.

This method returns a ResultSet in which each row describes a stored procedure parameter or column. If a database access error occurs, it throws a SQLException.

The Complete Solution: getStoredProcedureSignature()

You need to be careful in invoking the DatabaseMetaData.getProcedureColumns() method. First, make sure that you pass actual parameter values for catalogs and schemas rather than passing empty and null values (this will speed up your method call). Second, be aware of overloaded stored procedures (each database vendor might handle overloaded stored procedures differently—refer to the vendor's database documentation).

```
/**
    * Retrieves a description of the given catalog's stored
    * procedure parameter and result columns.
    *
    * @param conn the Connection object
    * @param catalog a catalog.
    * @param schemaPattern a schema pattern.
    * @param procedureNamePattern name of a stored procedure
```

```
     * @param columnNamePattern a column name pattern.
     * @return XML.
     * @throws Exception Failed to get the stored procedure's signature.
     */
    public static String getStoredProcedureSignature(
            java.sql.Connection conn,
            String catalog,
            String schemaPattern,
            String procedureNamePattern,
            String columnNamePattern) throws Exception {

          // Get DatabaseMetaData
          DatabaseMetaData dbMetaData = conn.getMetaData();
          if (dbMetaData == null) {
              return null;
          }
          ResultSet rs = dbMetaData.getProcedureColumns(catalog,
                                                  schemaPattern,
                                                  procedureNamePattern,
                                                  columnNamePattern);

          StringBuffer sb = new StringBuffer("<?xml version='1.0'>");
          sb.append("<stored_procedures_signature>");
          while(rs.next()) {
              // get stored procedure metadata
              String procedureCatalog     = rs.getString(1);
              String procedureSchema      = rs.getString(2);
              String procedureName        = rs.getString(3);
              String columnName           = rs.getString(4);
              short  columnReturn         = rs.getShort(5);
              int    columnDataType       = rs.getInt(6);
              String columnReturnTypeName = rs.getString(7);
              int    columnPrecision      = rs.getInt(8);
              int    columnByteLength     = rs.getInt(9);
              short  columnScale          = rs.getShort(10);
              short  columnRadix          = rs.getShort(11);
              short  columnNullable       = rs.getShort(12);
              String columnRemarks        = rs.getString(13);

              sb.append("<storedProcedure name=\"");
              sb.append(procedureName);
              sb.append("\">");
              appendXMLTag(sb, "catalog", procedureCatalog);
              appendXMLTag(sb, "schema", procedureSchema);
              appendXMLTag(sb, "columnName", columnName);
              appendXMLTag(sb, "columnReturn", getColumnReturn(columnReturn));
              appendXMLTag(sb, "columnDataType", columnDataType);
              appendXMLTag(sb, "columnReturnTypeName", columnReturnTypeName);
              appendXMLTag(sb, "columnPrecision", columnPrecision);
              appendXMLTag(sb, "columnByteLength", columnByteLength);
              appendXMLTag(sb, "columnScale", columnScale);
              appendXMLTag(sb, "columnRadix", columnRadix);
              appendXMLTag(sb, "columnNullable", columnNullable);
              appendXMLTag(sb, "columnRemarks", columnRemarks);
              sb.append("</storedProcedure>");
          }
          sb.append("</stored_procedures_signature>");

          // Close database resources
          rs.close();
          //conn.close();
          return sb.toString();
    }
```

getColumnReturn():

```
    private static String getColumnReturn(short columnReturn) {
          switch(columnReturn) {
             case DatabaseMetaData.procedureColumnIn:
                  return "In";
             case DatabaseMetaData.procedureColumnOut:
                  return "Out";
             case DatabaseMetaData.procedureColumnInOut:
                  return "In/Out";
             case DatabaseMetaData.procedureColumnReturn:
                  return "return value";
             case DatabaseMetaData.procedureColumnResult:
                  return "return ResultSet";
             default:
               return "unknown";
          }
    }
```

appendXMLTag():

```
    private static void appendXMLTag(StringBuffer buffer,
                                     String tagName,
                                     int value) {
            buffer.append("<");
            buffer.append(tagName);
            buffer.append(">");
            buffer.append(value);
            buffer.append("</");
            buffer.append(tagName);
            buffer.append(">");
    }
    private static void appendXMLTag(StringBuffer buffer,
                                     String tagName,
                                     String value) {
            buffer.append("<");
            buffer.append(tagName);
            buffer.append(">");
            buffer.append(value);
            buffer.append("</");
            buffer.append(tagName);
            buffer.append(">");
    }
}
```

Client Program 1

```
    String signature = DatabaseMetaDataTool.getStoredProcedureSignature
                         (conn,
                         "",
                         "OCTOPUS",      // user
                         "RAISESALARY",  // stored procedure name
                         "%");           // all columns
    System.out.println(signature);
```

Output of Client Program 1

```
<?xml version='1.0'>
<stored_procedures_signature>

    <storedProcedure name="RAISESALARY">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <columnName>DEPTNUMBERPARAM</columnName>
        <columnReturn>In</columnReturn>
        <columnDataType>3</columnDataType>
        <columnReturnTypeName>NUMBER</columnReturnTypeName>
        <columnPrecision>22</columnPrecision>
        <columnByteLength>22</columnByteLength>
        <columnScale>0</columnScale>
        <columnRadix>10</columnRadix>
        <columnNullable>1</columnNullable>
        <columnRemarks>null</columnRemarks>
    </storedProcedure>

    <storedProcedure name="RAISESALARY">
        <catalog>null</catalog>
        <schema>OCTOPUS</schema>
        <columnName>PERCENTAGE</columnName>
        <columnReturn>In</columnReturn>
        <columnDataType>3</columnDataType>
        <columnReturnTypeName>NUMBER</columnReturnTypeName>
        <columnPrecision>22</columnPrecision>
        <columnByteLength>22</columnByteLength>
        <columnScale>0</columnScale>
        <columnRadix>10</columnRadix>
        <columnNullable>1</columnNullable>
        <columnRemarks>null</columnRemarks>
    </storedProcedure>

</stored_procedures_signature>
```

Client Program 2

For this client program, let's define another stored procedure (call it `showUsers`, which lists all of the users) that does not have any arguments. Note that the `all_users` table holds all of the users in the Oracle database.

```
SQL> describe all_users;
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------
 USERNAME                                  NOT NULL VARCHAR2(30)
 USER_ID                                   NOT NULL NUMBER
 CREATED                                   NOT NULL DATE
SQL>
SQL> CREATE OR REPLACE PROCEDURE showUsers AS
  2  BEGIN
  3    for A_USER  in ( SELECT *  from all_users ) LOOP
  4  --        do something
  5            DBMS_OUTPUT.PUT_LINE('UserName: '|| A_USER.UserName);
  6    end loop;
  7  END showUsers;
  8  /
Procedure created.
SQL> describe showusers;
PROCEDURE showusers

SQL> set serveroutput on
SQL> exec showUsers;
UserName: SYS
UserName: SYSTEM
UserName: OUTLN
UserName: DBSNMP
...
```

```
UserName: QS_CBADM
UserName: QS_CB
UserName: QS_CS
UserName: SCOTT
UserName: OCTOPUS

PL/SQL procedure successfully completed.
    String signature = DatabaseMetaDataTool.getStoredProcedureSignature
                         (conn,
                         "",
                         "OCTOPUS",      // user
                         "SHOWUSERS",    // stored procedure name
                         "%");           // all columns
    System.out.println(signature);
```

Output of Client Program 2

As you can observe, there are no signature definitions for the showUsers stored procedure because showUsers has no arguments whatsoever.

```
<?xml version='1.0'>
<stored_procedures_signature>
</stored_procedures_signature>
```

3.5. What Is the Username of the Database Connection?

You can use DatabaseMetaData to get the name of the database user used in creating a connection object. The following snippet shows how:

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
...
Connection conn = null;
try {
    conn = getConnection(); // returns a Connection
    DatabaseMetaData dbMetaData = conn.getMetaData();
    if (dbMetaData == null) {
        System.out.prinln("database does not support metadata.");
        System.exit(0);
    }

    // retrieve the user name as known to this database.
    String user = dbMetaData.getUserName();
    System.out.prinln("database user="+user);
}
catch(Exception e) {
    // handle the exception
    e.printStackTrace();
}
```

3.6. Is the Database Connection Read-Only?

In GUI database applications, before letting the user insert or update records, you need to make sure that the given `Connection` object is `updatable` (which means that records can be inserted or updated). To check for this, you can use the `DatabaseMetaData.isReadOnly()` method. This method returns `true` if the associated database is in read-only mode (which means that inserts or updates are not allowed). The following snippet shows how to use this method:

```
import java.sql.Connection;
import java.sql.DatabaseMetaData;
...
Connection conn = null;
DatabaseMetaData dbMetaData = null;
try {
  conn = getConnection();   // get a valid database connection
  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

     // check to see if the database is read-only
     boolean readOnly = dbMetaData.isReadOnly();
     if (readOnly) {
        // insert/updates are not allowed
     }
     else {
        // insert/updates are allowed
     }
     ...
  }
}
catch(SQLException e) {
  // deal and handle the exception
  ...
}
finally {
  // close resources
}
```

3.7. What Is the JDBC's Driver Information?

`DatabaseMetaData` has four driver-related methods, which are discussed in this section. We will combine all of them into a single method called `getDriverInformation()` and return the result as XML (serialized as a `String` object).

DatabaseMetaData Methods Supporting Driver Information

```
   int getJDBCMajorVersion()
      // Retrieves the major JDBC version number for this driver.
   int getJDBCMinorVersion()
      // Retrieves the minor JDBC version number for this driver.
   String getDriverName()
      // Retrieves the name of this JDBC driver.
   String getDriverVersion()
      // Retrieves the version number of this JDBC driver as a String.
```

XML Syntax for Output (Driver Information)

```
<?xml version='1.0'>
<DriverInformation>
   <driverName>driver name</driverName>
   <driverVersion>driver version</driverVersion>
   <jdbcMajorVersion>JDBC major version</jdbcMajorVersion>
   <jdbcMinorVersion>JDBC minor version</jdbcMinorVersion>
</DriverInformation>
```

The Solution

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

```
/**
 * Get driver name and version information.
 * This method calls 4 methods (getDriverName(),
 * getDriverVersion(), getJDBCMajorVersion(),
 * getJDBCMinorVersion()) to get the required information
 * and it returns the information as XML.
 *
 * @param conn the Connection object
 * @return driver name and version information
 * as an XML document (represented as a String object).
 *
 */
public static String getDriverInformation(java.sql.Connection conn)
    throws Exception {
    try {
        DatabaseMetaData meta = conn.getMetaData();
        if (meta == null) {
           return null;
        }

        StringBuffer sb = new StringBuffer("<?xml version='1.0'>");
        sb.append("<DriverInformation>");
        // Oracle (and some other vendors) do not support
        // some the following methods; therefore, we need
        // to use a try-catch block.
        try {
            int jdbcMajorVersion = meta.getJDBCMajorVersion();
            appendXMLTag(sb, "jdbcMajorVersion", jdbcMajorVersion);
        }
        catch(Exception e) {
            appendXMLTag(sb, "jdbcMajorVersion", "unsupported feature");
        }

        try {
            int jdbcMinorVersion = meta.getJDBCMinorVersion();
            appendXMLTag(sb, "jdbcMinorVersion", jdbcMinorVersion);
        }
        catch(Exception e) {
            appendXMLTag(sb, "jdbcMinorVersion", "unsupported feature");
        }

        String driverName = meta.getDriverName();
        String driverVersion = meta.getDriverVersion();
        appendXMLTag(sb, "driverName", driverName);
        appendXMLTag(sb, "driverVersion", driverVersion);
        sb.append("</DriverInformation>");

        return sb.toString();
    }
    catch(Exception e) {
        // handle exception
        e.printStackTrace();
        throw new Exception("could not get the database information:"+
           e.toString());
    }
}
```

Discussion

To get the driver information (such as the name and version), we call the methods (listed earlier) and the result is returned as XML. The advantage of our solution is that you get the required information with a single call and the result (as XML) can be used by any kind of client. Note that `oracle.jdbc.OracleDatabaseMetaData.getJDBCMajorVersion()` and `oracle.jdbc.OracleDatabaseMetaData.getJDBCMinorVersion()` are unsupported features; 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. The driver information does not change frequently and therefore it can be cached in the server-side.

Client Using MySQL

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

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

public class TestMySqlDatabaseMetaDataTool_DriverInformation {

   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("-------- getDriverInformation -------------");
         System.out.println("conn="+conn);
         String driverInfo = DatabaseMetaDataTool.getDriverInformation(conn);
         System.out.println(driverInfo);
         System.out.println("------------------------------------");
      }
      catch(Exception e){
         e.printStackTrace();
         System.exit(1);
      }
      finally {
         DatabaseUtil.close(conn);
      }
   }
}
```

Output Using MySQL

```
-------- getDriverInformation ------
conn=com.mysql.jdbc.Connection@1837697
<?xml version='1.0'>
<DriverInformation>
   <jdbcMajorVersion>3</jdbcMajorVersion>
   <jdbcMinorVersion>0</jdbcMinorVersion>
   <driverName>MySQL-AB JDBC Driver</driverName>
   <driverVersion>3.0.5-gamma</driverVersion>
</DriverInformation>
-----------------------------------
```

Client Using Oracle

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

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

public class TestOracleDatabaseMetaDataTool_DriverInformation {
   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("-------- getDriverInformation -------------");
         System.out.println("conn="+conn);
         String driverInfo = DatabaseMetaDataTool.getDriverInformation(conn);
         System.out.println(driverInfo);
         System.out.println("-----------------------------------");
      }
      catch (Exception e){
         e.printStackTrace();
         System.exit(1);
      }
      finally {
         DatabaseUtil.close(conn);
      }
   }
}
```

Output Using Oracle

The following output is formatted to fit the page:

```
-------- getDriverInformation ------
conn=oracle.jdbc.driver.OracleConnection@169ca65
<?xml version='1.0'>
<DriverInformation>
   <jdbcMajorVersion>unsupported feature</jdbcMajorVersion>
   <jdbcMinorVersion>unsupported feature</jdbcMinorVersion>
   <driverName>Oracle JDBC driver</driverName>
   <driverVersion>9.2.0.1.0</driverVersion>
   </DriverInformation>
------------------------------------
```

3.8. How Can You Determine Where a Given Table Is Referenced via Foreign Keys?

DatabaseMetaData.getExportedKeys() returns a ResultSet object, which relates to other tables that reference the given table as a foreign key container. In other words, it tells us which tables have foreign keys that reference this table. A *primary key (PK)* is a column or set of columns that uniquely identifies a row or record in a table. A *foreign key (FK)* is one or more columns in one table that are used as a primary key in another table. First, we'll look at these concepts in a simple example, and then we'll develop a JDBC solution and a test client program to show these relationships using DatabaseMetaData.getExportedKeys().

Oracle Database Setup

First, let's create two tables (dept_table and emp_table) and define the PK and FK for these tables. Figure 3-1 illustrates the relationship of these tables.

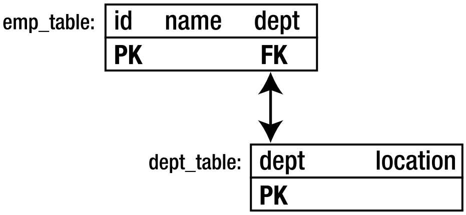

Figure 3-1. *Relationship of tables*

Keep in mind that if you violate the PK and FK rules, the SQL INSERT operation will fail:

```
$ sqlplus scott/tiger
SQL*Plus: Release 10.1.0.2.0 - Production on Tue Aug 24 14:17:06 2004
Copyright (c) 1982, 2004, Oracle.  All rights reserved.
```

```
SQL> create table dept_table (
  2     dept varchar2(2) not null primary key,
  3     location varchar2(8)
  4  );
Table created.
SQL> desc dept_table;
 Name                                      Null?    Type
 ----------------------------------------- -------- -----------
 DEPT                                      NOT NULL VARCHAR2(2)
 LOCATION                                           VARCHAR2(8)
SQL> create table emp_table (
  2     id varchar2(5) not null primary key,
  3     name varchar2(10),
  4     dept varchar2(2) not null references dept_table(dept)
  5  );
Table created.
SQL> desc emp_table;
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------
 ID                                        NOT NULL VARCHAR2(5)
 NAME                                               VARCHAR2(10)
 DEPT                                      NOT NULL VARCHAR2(2)
SQL> insert into dept_table(dept, location) values('11', 'Boston');
SQL> insert into dept_table(dept, location) values('22', 'Detroit');
SQL> insert into emp_table(id, name, dept) values('55555', 'Alex', '11');
SQL> insert into emp_table(id, name, dept) values('66666', 'Mary', '22');
SQL> select * from dept_table;
DEPT LOCATION
---- --------
11   Boston
22   Detroit
SQL> select * from emp_table;
ID     NAME       DEPT
-----  ---------- ----
55555  Alex         11
66666  Mary         22
SQL> insert into emp_table(id, name, dept) values('77777', 'Bob', '33');
insert into emp_table(id, name, dept) values('77777', 'Bob', '33')
*
ERROR at line 1:
ORA-02291: integrity constraint (SCOTT.SYS_C005465) violated - parent key not
Found
```

■ **Note** Since dept 33 is not defined in `dept_table`, Oracle issues an error.

```
SQL> select * from emp_table;
ID     NAME       DEPT
-----  ---------- ----
55555  Alex         11
66666  Mary         22
SQL> commit;
```

DatabaseMetaData.getExportedKeys() According to J2SE

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

This method retrieves a description of the foreign key columns that reference the given table's primary key columns (the foreign keys exported by a table). They are ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and KEY_SEQ. Each foreign key column description has columns shown in Table 3-4.

Table 3-4. *ResultSet Object's Columns for Invoking getExportedKeys()*

Field Name	Type	Description
PKTABLE_CAT	String	The primary key table catalog (may be null)
PKTABLE_SCHEM	String	The primary key table schema (may be null)
PKTABLE_NAME	String	The primary key table name
PKCOLUMN_NAME	String	The primary key column name
FKTABLE_CAT	String	The foreign key table catalog (may be null) that is being exported (may be null)
FKTABLE_SCHEM	String	The foreign key table schema (may be null) that being exported (may be null)
FKTABLE_NAME	String	The foreign key table name that is being exported
FKCOLUMN_NAME	String	The foreign key column name that is being exported
KEY_SEQ	short	The sequence number within the foreign key
UPDATE_RULE	short	Indicates what happens to the foreign key when the primary key is updated: importedNoAction: Do not allow the update of the primary key if it has been imported importedKeyCascade: Change the imported key to agree with the primary key update importedKeySetNull: Change the imported key to NULL if its primary key has been updated importedKeySetDefault: Change the imported key to the default values if its primary key has been updated importedKeyRestrict: The same as importedKeyNoAction (for ODBC 2.*x* compatibility)
DELETE_RULE	short	Indicates what happens to the foreign key when the primary key is deleted: importedKeyNoAction: Do not allow the delete of the primary key if it has been imported importedKeyCascade: Delete rows that import a deleted key importedKeySetNull: Change the imported key to NULL if its primary key has been deleted importedKeyRestrict: The same as importedKeyNoAction (for ODBC 2.*x* compatibility) importedKeySetDefault: Change the imported key to the default if its primary key has been deleted

Field Name	Type	Description
FK_NAME	String	The foreign key name (may be null)
PK_NAME	String	The primary key name (may be null)
DEFERRABILITY	short	Indicates whether the evaluation of foreign key constraints can be deferred until commit: importedKeyInitiallyDeferred: See SQL-92 for definition importedKeyInitiallyImmediate: See SQL-92 for definition importedKeyNotDeferrable: See SQL-92 for definition

The method's parameters are as follows:

- catalog: A catalog name; it must match the catalog name as it is stored in this 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 this database.

This method returns a ResultSet object in which each row is a foreign key column description. If a database access error occurs, it throws a SQLException.

The Solution: Using DatabaseMetaData.getExportedKeys()

In using the DatabaseMetaData.getExportedKeys() method, try to pass all required parameters with non-null and non-empty values. Passing null/empty values might slow down getting the results from this method. If your database is not changing often, you may cache the returned values on the server side.

```
/**
 * class name: jcb.meta.DatabaseMetaDataTool
 *
 * Retrieves a description of the foreign key columns that
 * reference the given table's primary key columns (the foreign
 * keys exported by a table). They are ordered by FKTABLE_CAT,
 * FKTABLE_SCHEM, FKTABLE_NAME, and KEY_SEQ.
 *
 * @param conn the Connection object
 * @param catalog database catalog.
 * @param schema database schema.
 * @param tableName name of a table in the database.
 * @return the list (as an XML string) of the foreign key columns
 * that reference the given table's primary key columns
 *
 * @exception Failed to get the ExportedKeys for a given table.
 */
```

```
public static String getExportedKeys(java.sql.Connection conn,
                                      String catalog,
                                      String schema,
                                      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.
        rs = meta.getExportedKeys(catalog, schema, tableName.toUpperCase());
        if (rs == null) {
            return null;
        }

        StringBuffer buffer = new StringBuffer();
        buffer.append("<exportedKeys>");
        while (rs.next()) {
            String fkTableName =
                DatabaseUtil.getTrimmedString(rs, "FKTABLE_NAME");
            String fkColumnName =
                DatabaseUtil.getTrimmedString(rs, "FKCOLUMN_NAME");
            int fkSequence = rs.getInt("KEY_SEQ");
            buffer.append("<exportedKey>");
            buffer.append("<catalog>");
            buffer.append(catalog);
            buffer.append("</catalog>");
            buffer.append("<schema>");
            buffer.append(schema);
            buffer.append("</schema>");
            buffer.append("<tableName>");
            buffer.append(tableName);
            buffer.append("</tableName>");
            buffer.append("<fkTableName>");
            buffer.append(fkTableName);
            buffer.append("</fkTableName>");
            buffer.append("<fkColumnName>");
            buffer.append(fkColumnName);
```

```
                 buffer.append("</fkColumnName>");
                 buffer.append("<fkSequence>");
                 buffer.append(fkSequence);
                 buffer.append("</fkSequence>");
                 buffer.append("</exportedKey>");
             }
             buffer.append("</exportedKeys>");
             return buffer.toString();
        }
        finally {
            DatabaseUtil.close(rs);
        }
     }
```

The Oracle Client Test Program

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

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

public class DemoGetExportedKeys_Oracle {

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

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            System.out.println("------DemoGetExportedKeys_Oracle begin---------");
            conn = getConnection();
            System.out.println("DemoGetExportedKeys_Oracle: conn="+conn);
            String exportedKeysAsXML = DatabaseMetaDataTool.getExportedKeys(
                                        conn, null, "SCOTT", "DEPT_TABLE");
            System.out.println("exportedKeysAsXML=" + exportedKeysAsXML);
            System.out.println("------DemoGetExportedKeys_Oracle end---------");
        }
        catch(Exception e){
            e.printStackTrace();
            System.exit(1);
        }
        finally {
            // release database resources
            DatabaseUtil.close(conn);
        }
    }
}
```

Running the Client Test Program

```
$ javac DemoGetExportedKeys_Oracle.java
$ java DemoGetExportedKeys_Oracle
```

```
------DemoGetExportedKeys_Oracle begin---------
DemoGetExportedKeys_Oracle: conn=oracle.jdbc.driver.OracleConnection@1c6f579
exportedKeysAsXML=
<exportedKeys>
   <exportedKey>
       <catalog>null</catalog>
       <schema>SCOTT</schema>
       <tableName>DEPT_TABLE</tableName>
       <fkTableName>EMP_TABLE</fkTableName>
       <fkColumnName>DEPT</fkColumnName>
       <fkSequence>1</fkSequence>
   </exportedKey>
</exportedKeys>
------DemoGetExportedKeys_Oracle end---------
```

The MySQL Database Setup

In the current version of MySQL (version 4.0.8), only InnoDB table types support the foreign key concept. According to MySQL, starting with MySQL 5.1, foreign keys will be supported for all table types, not just InnoDB. Let's create two tables (dept_table and emp_table) and define the PK and FK. Keep in mind that if you violate the PK and FK rules, the SQL INSERT operation will fail.

```
$ mysql --user=root --password=root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 130 to server version: 4.0.18-nt
mysql> use octopus;
Database changed
```

```
mysql> create table dept_table (
    -> dept char(2) not null,
    -> location varchar(8),
    -> PRIMARY KEY(dept)
    -> ) TYPE=InnoDB;
Query OK, 0 rows affected (0.15 sec)
mysql> create table emp_table (
    -> dept char(2) not null,
    -> id varchar(5) not null,
    -> name varchar(10),
    -> PRIMARY KEY(id),
    -> INDEX dept_index (dept),
    -> CONSTRAINT fk_dept FOREIGN KEY(dept) REFERENCES dept_table(dept)
    -> ) TYPE=InnoDB;
Query OK, 0 rows affected (0.11 sec)
mysql> insert into dept_table(dept, location) values('11', 'Boston');
mysql> insert into dept_table(dept, location) values('22', 'Detroit');
mysql> insert into emp_table(id, name, dept) values('55555', 'Alex', '11');
mysql> insert into emp_table(id, name, dept) values('66666', 'Mary', '22');
mysql> insert into emp_table(id, name, dept) values('77777', 'Bob', '33');
ERROR 1216: Cannot add or update a child row: a foreign key constraint fails
mysql> select * from emp_table;
+------+-------+------+
| dept | id    | name |
+------+-------+------+
| 11   | 55555 | Alex |
| 22   | 66666 | Mary |
+------+-------+------+
2 rows in set (0.00 sec)
mysql> select * from dept_table;
+------+----------+
| dept | location |
+------+----------+
| 11   | Boston   |
| 22   | Detroit  |
+------+----------+
2 rows in set (0.00 sec)
```

The MySQL Client Test Program

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

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

public class DemoGetExportedKeys_MySQL {
    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;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            System.out.println("------DemoGetExportedKeys_MySQL begin---------");
            conn = getConnection();
            System.out.println("DemoGetExportedKeys_MySQL: conn="+conn);

            String exportedKeysAsXML = DatabaseMetaDataTool.getExportedKeys(
                                      conn, "octopus", null, "DEPT_TABLE");
            System.out.println("exportedKeysAsXML=" + exportedKeysAsXML);
            System.out.println("------DemoGetExportedKeys_MySQL end---------");
        }
        catch(Exception e){
            e.printStackTrace();
            System.exit(1);
        }
        finally {
            // release database resources
            DatabaseUtil.close(conn);
        }
    }
}
```

Running the Client Test Program

```
$ javac  DemoGetExportedKeys_MySQL.java
$ java DemoGetExportedKeys_MySQL
```

```
------DemoGetExportedKeys_MySQL begin---------
DemoGetExportedKeys_MySQL: conn=com.mysql.jdbc.Connection@a1807c
exportedKeysAsXML=
<exportedKeys>
    <exportedKey>
        <catalog>octopus</catalog>
        <schema>null</schema>
        <tableName>DEPT_TABLE</tableName>
        <fkTableName>emp_table</fkTableName>
        <fkColumnName>dept</fkColumnName>
        <fkSequence>1</fkSequence>
    </exportedKey>
</exportedKeys>
------DemoGetExportedKeys_MySQL end---------
```

3.9. What Foreign Keys Are Used in a Table?

`DatabaseMetaData.getImportedKeys()` returns a `ResultSet` object with data about foreign key columns, tables, sequence, and update and delete rules. `DatabaseMetaData`'s `getImportedKeys()` returns a `ResultSet` that retrieves a description of the primary key columns referenced by a table's foreign key columns (the primary keys imported by a table). The `ResultSet` object's records are ordered by the column names `PKTABLE_CAT`, `PKTABLE_SCHEM`, `PKTABLE_NAME`, and `KEY_SEQ`.

A *primary key (PK)* is a column or set of columns that uniquely identifies a row or record in a table. A *foreign key (FK)* is one or more columns in one table that are used as a primary key in another table. First, we'll look at these concepts in a simple example, and then we'll develop a JDBC solution and a test client program to show these relationships using `DatabaseMetaData.getImportedKeys()`.

Oracle Database Setup

Let's create three tables (`roles_table`, `emps_table`, and `emps_roles`) and define the PK and FK. Figure 3-2 illustrates the relationships of these tables.

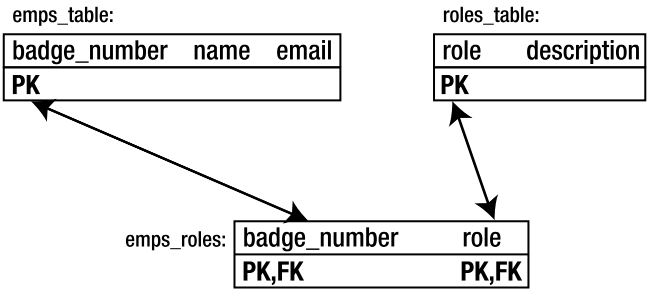

Figure 3-2. *Relationships of three database tables*

Keep in mind that if you violate the PK and FK rules, the SQL `INSERT` operation will fail.

```
create table emps_table (
    badge_number varchar(5) not null,
    name varchar(20) not null,
    email varchar(20) not null,
    primary key (badge_number)
);
```

```
create table roles_table (
    role varchar(5) not null,
    description varchar(25) not null,
    primary key (role)
);


create table emps_roles (
    badge_number varchar(5) not null,
    role varchar(5) not null,

    primary key (badge_number, role),
    foreign key (badge_number) references emps_table(badge_number),
    foreign key (role) references roles_table(role)
);

insert into roles_table(role, description) values('dba', 'database administrator');
insert into roles_table(role, description) values('mgr', 'database manager');
insert into roles_table(role, description) values('dev', 'database developer');

insert into emps_table(badge_number, name, email)
    values('11111', 'Alex', 'alex@yahoo.com');

insert into emps_table(badge_number, name, email)
    values('22222', 'Mary', 'mary@yahoo.com');

insert into emps_roles(badge_number, role)
    values('11111', 'mgr');
insert into emps_roles(badge_number, role)
    values('11111', 'dev');
insert into emps_roles(badge_number, role)
    values('22222', 'dba');


SQL> select * from roles_table;

ROLE  DESCRIPTION
----- ----------------------
dba   database administrator
mgr   database manager
dev   database developer
SQL> select * from emps_table;
```

```
BADGE  NAME  EMAIL
-----  ----  --------------
11111  Alex  alex@yahoo.com
22222  Mary  mary@yahoo.com
SQL> select * from emps_roles;
BADGE  ROLE
-----  -----
11111  dev
11111  mgr
22222  dba
```

DatabaseMetaData.getImportedKeys() Signature

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

This method retrieves a description of the primary key columns that are referenced by a table's foreign key columns (the primary keys imported by a table). They are ordered by `PKTABLE_CAT`, `PKTABLE_SCHEM`, `PKTABLE_NAME`, and `KEY_SEQ`.

Each primary key column description has the columns shown in Table 3-5.

Table 3-5. *ResultSet Object's Columns for Invoking getImportedKeys()*

Field Name	Type	Description
`PKTABLE_CAT`	`String`	The primary key table catalog being imported (may be `null`)
`PKTABLE_SCHEM`	`String`	The primary key table schema being imported (may be `null`)
`PKTABLE_NAME`	`String`	The primary key table name being imported
`PKCOLUMN_NAME`	`String`	The primary key column name being imported
`FKTABLE_CAT`	`String`	The foreign key table catalog (may be `null`)
`FKTABLE_SCHEM`	`String`	The foreign key table schema (may be `null`)
`FKTABLE_NAME`	`String`	The foreign key table name
`FKCOLUMN_NAME`	`String`	The foreign key column name
`KEY_SEQ`	`short`	The sequence number within a foreign key
`UPDATE_RULE`	`short`	Indicates what happens to a foreign key when the primary key is updated: `importedNoAction`: Do not allow the update of the primary key if it has been imported `importedKeyCascade`: Change the imported key to agree with the primary key update `importedKeySetNull`: Change the imported key to `NULL` if its primary key has been updated `importedKeySetDefault`: Change the imported key to the default values if its primary key has been updated `importedKeyRestrict`: The same as `importedKeyNoAction` (for ODBC 2.x compatibility)

Continued

Table 3-5. *Continued*

Field Name	Type	Description
DELETE_RULE	short	Indicates what happens to the foreign key when the primary key is deleted: importedKeyNoAction: Do not allow the delete of the primary key if it has been imported importedKeyCascade: Delete rows that import a deleted key importedKeySetNull: Change the imported key to NULL if its primary key has been deleted importedKeyRestrict: The same as importedKeyNoAction (for ODBC 2.*x* compatibility) importedKeySetDefault: Change the imported key to the default if its primary key has been deleted
FK_NAME	String	The foreign key name (may be null)
PK_NAME	String	The primary key name (may be null)
DEFERRABILITY	short	Specifies whether the evaluation of foreign key constraints can be deferred until commit: importedKeyInitiallyDeferred: See SQL-92 for definition importedKeyInitiallyImmediate: See SQL-92 for definition importedKeyNotDeferrable: See SQL-92 for definition

This method's parameters are as follows:

- 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.

This method returns a ResultSet in which each row is a primary key column description. If a database access error occurs, it throws a SQLException.

The Solution: Using DatabaseMetaData.getImportedKeys()

When using the DatabaseMetaData.getImportedKeys() method, try to pass all required parameters with non-null and non-empty values. Passing null/empty values might slow down getting the results from this method. If your database is not changing often, you may cache the returned values on the server side. This method will give you a good idea about the dependency of your database tables.

```
/**
 * class name: jcb.meta.DatabaseMetaDataTool
 *
 * Retrieves a description of the primary key columns that are
 * referenced by a table's foreign key columns (the primary keys
 * imported by a table). They are ordered by PKTABLE_CAT,
 * PKTABLE_SCHEM, PKTABLE_NAME, and KEY_SEQ.
 *
```

```
* @param conn the Connection object
* @param catalog database catalog.
* @param schema database schema.
* @param tableName name of a table in the database.
* @return the list (as an XML string) of the primary key columns
* that are referenced by a table's foreign key columns
*
* @exception Failed to get the ExportedKeys for a given table.
*/
public static String getImportedKeys(java.sql.Connection conn,
                                             String catalog,
                                             String schema,
                                             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.
        //
        rs = meta.getImportedKeys(catalog, schema, tableName.toUpperCase());
        if (rs == null) {
            return null;
        }

        StringBuffer buffer = new StringBuffer();
        buffer.append("<importedKeys>");
        while (rs.next()) {

            String pkTableName =
               DatabaseUtil.getTrimmedString(rs, "PKTABLE_NAME");
            String pkColumnName =
               DatabaseUtil.getTrimmedString(rs, "PKCOLUMN_NAME");
            String fkTableName =
               DatabaseUtil.getTrimmedString(rs, "FKTABLE_NAME");
            String fkColumnName =
               DatabaseUtil.getTrimmedString(rs, "FKCOLUMN_NAME");
            int fkSequence = rs.getInt("KEY_SEQ");
```

```
                buffer.append("<importedKey>");
                buffer.append("<catalog>");
                buffer.append(catalog);
                buffer.append("</catalog>");
                buffer.append("<schema>");
                buffer.append(schema);
                buffer.append("</schema>");
                buffer.append("<tableName>");
                buffer.append(tableName);
                buffer.append("</tableName>");
                buffer.append("<pkTableName>");
                buffer.append(pkTableName);
                buffer.append("</pkTableName>");
                buffer.append("<pkColumnName>");
                buffer.append(pkColumnName);
                buffer.append("</pkColumnName>");
                buffer.append("<fkTableName>");
                buffer.append(fkTableName);
                buffer.append("</fkTableName>");
                buffer.append("<fkColumnName>");
                buffer.append(fkColumnName);
                buffer.append("</fkColumnName>");
                buffer.append("<fkSequence>");
                buffer.append(fkSequence);
                buffer.append("</fkSequence>");
                buffer.append("</importedKey>");
            }
            buffer.append("</importedKeys>");
            return buffer.toString();
        }
        finally {
            DatabaseUtil.close(rs);
        }
    }
```

Oracle Client Test Program

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

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

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

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            System.out.println("------DemoGetImportedKeys_Oracle begin---------");
            conn = getConnection();
            System.out.println("DemoGetImportedKeys_Oracle: conn="+conn);
            String tableName = args[0];
            System.out.println("tableName=" + tableName);
            String importedKeysAsXML =
               DatabaseMetaDataTool.getImportedKeys(conn, null, "SCOTT", tableName);
            System.out.println("importedKeysAsXML=" + importedKeysAsXML);
            System.out.println("------DemoGetImportedKeys_Oracle end---------");
        }
        catch(Exception e){
            e.printStackTrace();
            System.exit(1);
        }
        finally {
            // release database resources
            DatabaseUtil.close(conn);
        }
    }
}
```

Running the Client Test Program

```
$ javac DemoGetImportedKeys_Oracle.java
$ java DemoGetImportedKeys_Oracle roles_table
------DemoGetImportedKeys_Oracle begin---------
DemoGetImportedKeys_Oracle: conn=oracle.jdbc.driver.OracleConnection@1c6f579
tableName=roles_table
importedKeysAsXML=
<importedKeys>
</importedKeys>
```

```
------DemoGetImportedKeys_Oracle end---------
$ java DemoGetImportedKeys_Oracle emps_table
------DemoGetImportedKeys_Oracle begin---------
DemoGetImportedKeys_Oracle: conn=oracle.jdbc.driver.OracleConnection@1c6f579
tableName=emps_table
importedKeysAsXML=
<importedKeys>
</importedKeys>

------DemoGetImportedKeys_Oracle end---------
$ java DemoGetImportedKeys_Oracle emps_roles
------DemoGetImportedKeys_Oracle begin---------
DemoGetImportedKeys_Oracle: conn=oracle.jdbc.driver.OracleConnection@1c6f579
tableName=emps_roles
importedKeysAsXML=
<importedKeys>
    <importedKey>
        <catalog>null</catalog>
        <schema>SCOTT</schema>
        <tableName>emps_roles</tableName>
        <pkTableName>EMPS_TABLE</pkTableName>
        <pkColumnName>BADGE_NUMBER</pkColumnName>
        <fkTableName>EMPS_ROLES</fkTableName>
        <fkColumnName>BADGE_NUMBER</fkColumnName>
        <fkSequence>1</fkSequence>
    </importedKey>
    <importedKey>
        <catalog>null</catalog>
        <schema>SCOTT</schema>
        <tableName>emps_roles</tableName>
        <pkTableName>ROLES_TABLE</pkTableName>
        <pkColumnName>ROLE</pkColumnName>
        <fkTableName>EMPS_ROLES</fkTableName>
        <fkColumnName>ROLE</fkColumnName>
        <fkSequence>1</fkSequence>
    </importedKey>
</importedKeys>
------DemoGetImportedKeys_Oracle end---------
```

MySQL Database Setup

In the current version of MySQL (version 4.0.8), only InnoDB table types support the foreign key concept. According to MySQL, starting with MySQL 5.1, foreign keys will be supported for all table types, not just InnoDB. Let’s create two tables (dept_table and emp_table) and define the PK and FK. Keep in mind that if you violate the PK and FK rules, the SQL INSERT operation will fail.

```
$ mysql --user=root --password=root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 4.0.18-nt
mysql> use octopus;
Database changed
mysql> create table emps_table (
    ->     badge_number varchar(5) not null,
    ->     name varchar(20) not null,
    ->     email varchar(20) not null,
    ->
    ->     primary key (badge_number)
    -> ) TYPE=InnoDB;
Query OK, 0 rows affected (0.24 sec)
mysql> create table roles_table (
    ->     role varchar(5) not null,
    ->     description varchar(25) not null,
    ->
    ->     primary key (role)
    -> ) TYPE=InnoDB;
Query OK, 0 rows affected (0.13 sec)
mysql> create table emps_roles (
    ->     badge_number varchar(5) not null,
    ->     role varchar(5) not null,
    ->
    ->     primary key (badge_number, role),
    ->     INDEX badge_number_index (badge_number),
    ->     foreign key (badge_number) references emps_table(badge_number),
    ->     INDEX role_index (role),
    ->     foreign key (role) references roles_table(role)
    -> ) TYPE=InnoDB;
Query OK, 0 rows affected (0.24 sec)
mysql> insert into roles_table(role, description)
       values('dba', 'database administrator');

mysql> insert into roles_table(role, description)
       values('mgr', 'database manager');

mysql> insert into roles_table(role, description)
       values('dev', 'database developer');

mysql> insert into emps_table(badge_number, name, email)
       values('11111', 'Alex', 'alex@yahoo.com');

mysql> insert into emps_table(badge_number, name, email)
       values('22222', 'Mary', 'mary@yahoo.com');
```

```
mysql> insert into emps_roles(badge_number, role)
       values('11111', 'mgr');

mysql> insert into emps_roles(badge_number, role)
       values('11111', 'dev');

mysql> insert into emps_roles(badge_number, role)
       values('22222', 'dba');

mysql> insert into emps_roles(badge_number, role) values('22222', 'a');
ERROR 1216: Cannot add or update a child row: a foreign key constraint fails
mysql> insert into emps_roles(badge_number, role) values('2222', 'a');
ERROR 1216: Cannot add or update a child row: a foreign key constraint fails
mysql> select * from emps_table;
+--------------+------+----------------+
| badge_number | name | email          |
+--------------+------+----------------+
| 11111        | Alex | alex@yahoo.com |
| 22222        | Mary | mary@yahoo.com |
+--------------+------+----------------+
2 rows in set (0.02 sec)
mysql> select * from roles_table;
+------+------------------------+
| role | description            |
+------+------------------------+
| dba  | database administrator |
| dev  | database developer     |
| mgr  | database manager       |
+------+------------------------+
3 rows in set (0.00 sec)
mysql> select * from emps_roles;
+--------------+------+
| badge_number | role |
+--------------+------+
| 11111        | dev  |
| 11111        | mgr  |
| 22222        | dba  |
+--------------+------+
3 rows in set (0.00 sec)
```

The MySQL Client Test Program

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

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

```
public class DemoGetImportedKeys_MySQL {

    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;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            System.out.println("------DemoGetImportedKeys_MySQL begin---------");
            conn = getConnection();
            System.out.println("DemoGetImportedKeys_MySQL: conn="+conn);
            String tableName = args[0];
            System.out.println("tableName=" + tableName);
            String importedKeysAsXML = DatabaseMetaDataTool.getImportedKeys(
                                         conn, "octopus", null, tableName);
            System.out.println("importedKeysAsXML=" + importedKeysAsXML);
            System.out.println("------DemoGetImportedKeys_MySQL end---------");
        }
        catch(Exception e){
            e.printStackTrace();
            System.exit(1);
        }
        finally {
            // release database resources
            DatabaseUtil.close(conn);
        }
    }
}
```

Running the Client Test Program

```
$ javac DemoGetImportedKeys_MySQL.java
$ java DemoGetImportedKeys_MySQL emps_table
```

```
------DemoGetImportedKeys_MySQL begin---------
DemoGetImportedKeys_MySQL: conn=com.mysql.jdbc.Connection@a1807c
tableName=emps_table
importedKeysAsXML= <importedKeys></importedKeys>
------DemoGetImportedKeys_MySQL end---------
```

```
$ java DemoGetImportedKeys_MySQL roles_table
```

```
------DemoGetImportedKeys_MySQL begin---------
DemoGetImportedKeys_MySQL: conn=com.mysql.jdbc.Connection@a1807c
tableName=roles_table
importedKeysAsXML= <importedKeys></importedKeys>
------DemoGetImportedKeys_MySQL end---------
```

```
$ java DemoGetImportedKeys_MySQL emps_roles
```

```
------DemoGetImportedKeys_MySQL begin---------
DemoGetImportedKeys_MySQL: conn=com.mysql.jdbc.Connection@a1807c
tableName=emps_roles
importedKeysAsXML=
<importedKeys>
    <importedKey>
        <catalog>octopus</catalog>
        <schema>null</schema>
        <tableName>emps_roles</tableName>
        <pkTableName>emps_table</pkTableName>
        <pkColumnName>badge_number</pkColumnName>
        <fkTableName>EMPS_ROLES</fkTableName>
        <fkColumnName>badge_number</fkColumnName>
        <fkSequence>1</fkSequence>
    </importedKey>
    <importedKey>
        <catalog>octopus</catalog>
        <schema>null</schema>
        <tableName>emps_roles</tableName>
        <pkTableName>roles_table</pkTableName>
        <pkColumnName>role</pkColumnName>
        <fkTableName>EMPS_ROLES</fkTableName>
        <fkColumnName>role</fkColumnName>
        <fkSequence>1</fkSequence>
    </importedKey>
</importedKeys>
------DemoGetImportedKeys_MySQL end---------
```

3.10. What Is the JDBC View of a Database's Internal Structure?

The JDBC views a database in terms of catalog, schema, table, view, column, triggers, indexes, and stored procedures. The JDBC view of a database's internal structure appears in Figure 3-3.

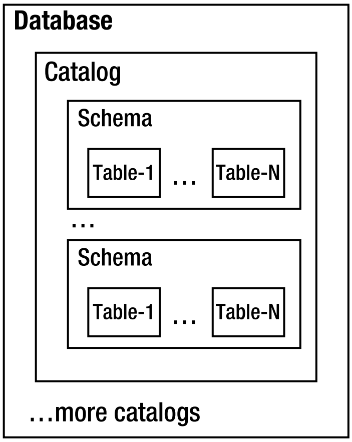

Figure 3-3. *Internal structure of a database*

From the JDBC view of a database:

- A database server has several catalogs (such as database partitions and databases).
- A catalog has several schemas (these are user-specific namespaces).
- A schema has several database objects (tables, views, triggers, indexes, stored procedures, etc.).

The `java.sql.DatabaseMetaData` interface has methods for discovering all the catalogs, schemas, tables, views, indexes, and stored procedures in the database server. These methods return a `ResultSet`, which can be traversed for getting the desired information.

```
public static void main(String[] args) throws Exception {
      // Load the database driver - in this case, we
      // use the Jdbc/Odbc bridge driver.
      Connection conn = null;
      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

         // Open a connection to the database
         conn = DriverManager.getConnection(
              "[jdbcURL]", "[login]", "[passwd]");

         // Get DatabaseMetaData
         DatabaseMetaData dbmd = conn.getMetaData();

         // Get all Catalogs
         System.out.println("\nCatalogs are called '" + dbmd.getCatalogTerm()
            + "' in this RDBMS.");
         processResultSet(dbmd.getCatalogTerm(), dbmd.getCatalogs());
```

```
            // Get all Schemas
            System.out.println("\nSchemas are called '" + dbmd.getSchemaTerm()
               + "' in this RDBMS.");
            processResultSet(dbmd.getSchemaTerm(), dbmd.getSchemas());

            // Get all Table-like types
            System.out.println("\nAll table types supported in this RDBMS:");
            processResultSet("Table type", dbmd.getTableTypes());
        }
        finally {
          // Close the Connection object
        }
    }

    public static void processResultSet(String preamble, ResultSet rs)
            throws SQLException {
        // Printout table data
        while(rs.next()) {
            // Printout
            System.out.println(preamble + ": " + rs.getString(1));
        }

        // Close database resources
        rs.close();
    }
```

3.11. Does a Database Support Batching?

With batch updating, a set of SQL statements is assembled and then sent to the database for execution. Batch updating can improve performance if you send lots of update statements to the database. According to Sun's JDBC Tutorial (http://java.sun.com/docs/books/tutorial/jdbc/jdbc2dot0/batchupdates.html), "A batch update is a set of multiple update statements that is submitted to the database for processing as a batch. Sending multiple update statements to the database together as a unit can, in some situations, be much more efficient than sending each update statement separately. This ability to send updates as a unit, referred to as the batch update facility, is one of the features provided with the JDBC 2.0 API."

Determine Whether a Database Supports Batching

```
    /**
     * Check to see if database supports batching.
     * @param conn connection object to the desired database
     * @return true if database supports batching.
     */
    public static boolean supportsBatching(java.sql.Connection conn) {
        if (conn == null) {
            return false;
        }
        try {
            DatabaseMetaData dbmd = conn.getMetaData();
            if (dbmd == null) {
                // database metadata not supported
                return false;
            }

            if (dbmd.supportsBatchUpdates()) {
                // batching is supported
                return true;
            }
            else {
                // batching is not supported
                return false;
            }
        }
        catch (Exception e) {
            // handle the exception
            return false;
        }
    }
```

Making Batch Updates

Next I'll provide an example that will perform batch updates. This example will be accomplished in several steps:

Step 1: Setting up the database

Step 2: Developing a sample program for batch updating

Step 3: Running the sample program

Step 4: Verifying the database results

Step 5: Discussing the solution

Step 1: Setting up the Database

Let's create a simple table, which will perform batch updates.

```
$ mysql --user=root --password=root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4240 to server version: 4.0.18-nt
mysql> use octopus;
Database changed
mysql> create table batch_table(
    -> id varchar(5) not null,
    -> name varchar(10) not null,
    -> primary key(id)
    -> );
```

```
Query OK, 0 rows affected (0.05 sec)
mysql> describe batch_table;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| id    | varchar(5)  |      | PRI |         |       |
| name  | varchar(10) |      |     |         |       |
+-------+-------------+------+-----+---------+-------+
2 rows in set (0.01 sec)
```

Step 2: Developing a Sample Program for Batch Updating

Here is the solution for batch updates. For discussion purposes, I have added line numbers.

```
 1 import java.sql.Connection;
 2 import java.sql.Statement;
 3 import java.sql.ResultSet;
 4 import java.sql.SQLException;
 5 import java.sql.BatchUpdateException;
 6 import jcb.util.DatabaseUtil;
 7
 8 public class TestBatchUpdate {
 9
10     public static Connection getConnection() throws Exception {
11         String driver = "org.gjt.mm.mysql.Driver";
12         String url = "jdbc:mysql://localhost/octopus";
13         String username = "root";
14         String password = "root";
15         Class.forName(driver);  // load MySQL driver
16         return DriverManager.getConnection(url, username, password);
17     }
18
19     public static void main(String args[]) {
20         ResultSet rs = null;
21         Statement stmt = null;
22         Connection conn = null;
23         try {
24             conn = getConnection();
25             stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
26                                         ResultSet.CONCUR_UPDATABLE);
27             conn.setAutoCommit(false);
28             stmt.addBatch("INSERT INTO batch_table(id, name) "+
29                           "VALUES('11', 'Alex')");
30             stmt.addBatch("INSERT INTO batch_table(id, name) "+
31                           "VALUES('22', 'Mary')");
32             stmt.addBatch("INSERT INTO batch_table(id, name) "+
33                           "VALUES('33', 'Bob')");
```

```
            int[] updateCounts = stmt.executeBatch();
            conn.commit();
            rs = stmt.executeQuery("SELECT * FROM batch_table");
            System.out.println("-- Table batch_table after insertion --");

            while (rs.next()) {
                String id = rs.getString("id");
                String name = rs.getString("name");
                System.out.println("id="+id +"  name="+name);
            }
        }
        catch(BatchUpdateException b) {
            System.err.println("SQLException: " + b.getMessage());
            System.err.println("SQLState: " + b.getSQLState());
            System.err.println("Message: " + b.getMessage());
            System.err.println("Vendor error code: " + b.getErrorCode());
            System.err.print("Update counts: ");
            int [] updateCounts = b.getUpdateCounts();
            for (int i = 0; i < updateCounts.length; i++) {
                System.err.print(updateCounts[i] + " ");
            }
        }
        catch(SQLException ex) {
            System.err.println("SQLException: " + ex.getMessage());
            System.err.println("SQLState: " + ex.getSQLState());
            System.err.println("Message: " + ex.getMessage());
            System.err.println("Vendor error code: " + ex.getErrorCode());
        }
        catch(Exception e) {
            e.printStackTrace();
            System.err.println("Exception: " + e.getMessage());
        }
        finally {
            DatabaseUtil.close(rs);
            DatabaseUtil.close(stmt);
            DatabaseUtil.close(conn);
        }
    }
}
```

Step 3: Running the Sample Program

```
$ javac TestBatchUpdate.java
$ java TestBatchUpdate
-- Table batch_table after insertion --
id=11  name=Alex
id=22  name=Mary
id=33  name=Bob
```

Step 4: Verifying the Database Results

```
$ mysql --user=root --password=root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 4.0.18-nt
mysql> use octopus;
Database changed
mysql> select * from batch_table;
+----+------+
| id | name |
+----+------+
| 11 | Alex |
| 22 | Mary |
| 33 | Bob  |
+----+------+
3 rows in set (0.00 sec)
```

Step 5: Discussing the Solution

Let's look at this solution in detail:

Lines 1–6: Import required classes and interfaces from the `java.sql` package.

Lines 10–17: The `getConnection()` method loads the JDBC driver, and then creates and returns a new database `Connection` object.

Lines 24–35: With the JDBC 2.0 API, `Statement`, `PreparedStatement`, and `CallableStatement` objects have the ability to maintain a list of SQL commands that can be submitted together as a batch. They are created with an associated list, which is initially empty. You can add SQL commands to this list with the method `addBatch()`, and you can empty the list with the method `clearBatch()`. You send all of the commands in the list to the database with the method `executeBatch()`. In lines 32–33, the `stmt` object sends the three SQL commands that were added to its list of commands off to the database to be executed as a batch. Note that `stmt` uses the method `executeBatch()` to send the batch of insertions, not the method `executeUpdate()`, which sends only one command and returns a single update count. The database server will execute the SQL commands in the order in which they were added to the list of commands.

Lines 36–43: The `ResultSet` object is used to retrieve all records from the `batch_table`. The `ResultSet` object is iterated to get information from all of the rows.

Lines 45–61: There are two exceptions that can be thrown during a batch update operation: `SQLException` and `BatchUpdateException`. If a batch update fails, then `BatchUpdateException` will be thrown by the JDBC driver. If there are other database problems, then `SQLException` will be thrown.

Lines 62–65: Finally, if there is any other exception, `java.lang.Exception` will be thrown.

Lines 66–70: This code closes all database resources. It releases the database and JDBC resources immediately instead of waiting for them to be automatically released.