especialista universitario en

Tecnologías Java Enterprise para Aplicaciones Web

Acceso a bases de datos con JDBC

Índice

1. INTRODUCCIÓN A JDBC 3

1.1. INSTALACIÓN DE DRIVERS 4

1.2. CONEXIÓN A LA BD 5

2. CONSULTA A UNA BD CON JDBC 7

2.1. CREACIÓN Y EJECUCIÓN DE SENTENCIAS SQL 7

2.2. SENTENCIAS DE CONSULTA 7

2.3. SENTENCIAS DE ACTUALIZACIÓN 9

2.4. OTRAS LLAMADAS A LA BD 10

2.5. RESTRICCIONES, MOVIMIENTOS Y ACTUALIZACIONES EN EL *RESULTSET* 12

3. INFORMACIÓN DE LA BD Y OPTIMIZACIÓN EN LAS CONSULTAS 14

3.1. OBTENCIÓN DE INFORMACIÓN PROPIA DE LA BD 14

3.2. OPTIMIZACIÓN DE SENTENCIAS 16

3.3. LLAMADAS A PROCEDIMIENTOS 17

4. OPCIONES AVANZADAS 20

4.1. FUENTES DE DATOS 20

4.2. RESERVA DE CONEXIONES 21

4.3. TRANSACCIONES 22

4.4. ROWSET 25

EJERCICIOS 29

Tema 1: Introducción a JDBC

JDBC nos permitirá acceder a bases de datos (BD) desde Java. Con JDBC no es necesario escribir distintos programas para distintas BD, sino que un único programa sirve para acceder a BD de distinta naturaleza. Incluso, podemos acceder a más de una BD de distinta fuente (Oracle, Access, MySql, etc.) en la misma aplicación. Podemos pensar en JDBC como el puente entre una base de datos y nuestro programa Java. Un ejemplo sencillo puede ser un applet que muestra dinámicamente información contenida en una base de datos. El applet utilizará JDBC para obtener dichos datos.

El esquema a seguir en un programa que use JDBC es el siguiente:

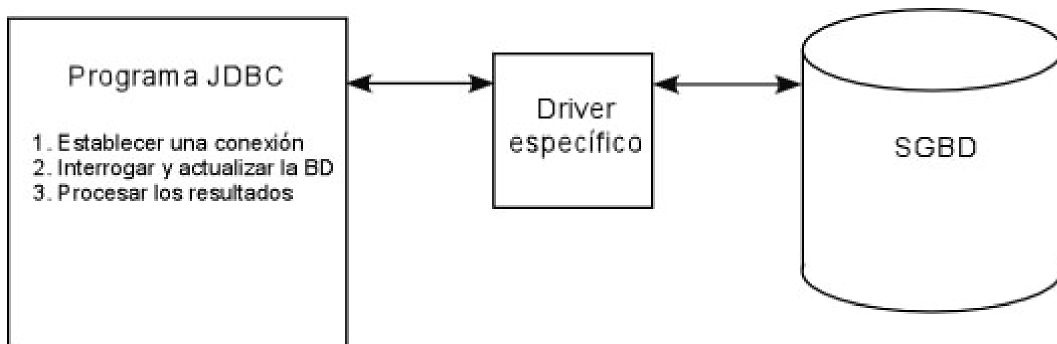

Figura 1. Esquema general de conexión con una base de datos.

Un programa Java que utilice JDBC primero deberá establecer una conexión con el SGBD. Para realizar dicha conexión haremos uso de un driver específico para cada SGBD que estemos utilizando. Una vez establecida la conexión ya podemos interrogar la BD con cualquier comando SQL (select, update, create, etc.). El resultado de un comando *select* es un objeto de la clase ResultSet, que contiene los datos que devuelve la consulta. Disponemos de métodos en *ResultSet* para manejar los datos devueltos. También podemos realizar cualquier operación en SQL (creación de tablas, gestión de usuarios, etc.).

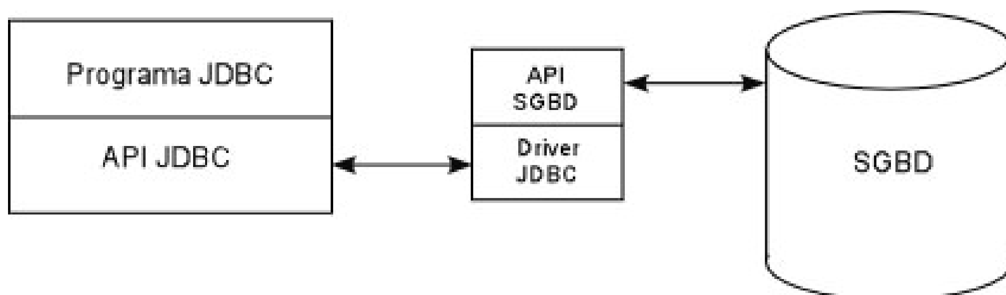

Figura 2. Conexión a través del API y un driver de JDBC.

Para realizar estas operaciones necesitaremos contar con un SGBD (sistema gestor de bases de datos) además de un driver específico para poder acceder a este SGBD. Vamos a utilizar dos SGBD: MySQL (disponible para Windows y Linux, de libre distribución) y PostGres (sólo para Linux, también de libre distribución).

1.1. Instalación de drivers

Los drivers para poder acceder a cada SGBD no forman parte de la distribución de Java por lo que deberemos obtenerlos por separado. Un driver es una capa intermedia que traduce las llamadas de JDBC a los APIs específicos del SGBD. Normalmente un driver es creado por el propio desarrollador del SGBD.

La distribución de JDBC incorpora los drivers para el puente JDBC-ODBC que nos permite acceder a cualquier BD que se gestione con ODBC. Para MySQL, deberemos descargar e instalar el SGBD y el driver, que puede ser obtenido en la dirección http://www.mysql.com/downloads/api-jdbc.html. El driver para PostGres se obtiene en http://jdbc.postgresql.org

Para instalar el driver lo único que deberemos hacer es incluir el fichero JAR que lo contiene en el CLASSPATH. Por ejemplo, para MySQL:

```
export CLASSPATH=$CLASSPATH:/directorio-donde-este/mm.mysql-2.0.4-bin.jar
```

Con el driver instalado, podremos cargarlo desde nuestra aplicación simplemente cargando dinámicamente la clase correspondiente al driver:

```
Class.forName("org.gjt.mm.mysql.Driver");
```

El driver JDBC-ODBC se carga como se muestra a continuación:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Y de forma similar para PostGres:

```
Class.forName("org.postgresql.Driver");
```

La carga del driver se deberá hacer siempre antes de conectar con la BD.

Pueden existir distintos tipos de drivers para la misma base de datos. Por ejemplo, a una BD en MySQL podemos acceder mediante ODBC o mediante su propio driver. Podríamos pensar que la solución más sencilla sería utilizar ODBC para todas las conexiones a SGBD. Sin embargo, dependiendo de la complejidad de la aplicación a desarrollar esto nos podría dar problemas. Determinados SGBD permiten realizar operaciones (transacciones, mejora de rendimiento, escabilidad, etc.) que se ven mermadas al realizar su conexión a través del driver ODBC. Por ello es preferible hacer uso de driver específicos para el SGBD en cuestión.

El ejemplo más claro de problemas en el uso de drivers es con los *Applets*. Cuando utilicemos acceso a bases de datos mediante JDBC desde un *Applet*, deberemos tener en cuenta que el *Applet* se ejecuta en la máquina del cliente, por lo que si la BD está alojada en nuestro servidor tendrá que establecer una conexión remota.

Aquí encontramos el problema de que si el *Applet* es visible desde Internet, es muy posible que el puerto en el que escucha el servidor de base de datos puede estar cortado por algún *firewall*, por lo que el acceso desde el exterior no sería posible.

El uso del puente JDBC-ODBC tampoco es recomendable en *Applets*, ya que requiere que cada cliente tenga configurada la fuente de datos ODBC adecuada en su máquina. Esto podemos controlarlo en el caso de una intranet, pero en el caso de Internet será mejor utilizar otros métodos para la conexión.

En cuanto a las excepciones, debemos capturar la excepción *SQLException* en casi todas las operaciones en las que se vea involucrado algún objeto JDBC.

1.2. Conexión a la BD

Una vez cargado el driver apropiado para nuestro SGBD deberemos establecer la conexión con la BD. Para ello utilizaremos el siguiente método:

```
Connection con = DriverManager.getConnection(url);
Connection con = DriverManager.getConnection(url, login,
password);
```

La conexión a la BD está encapsulada en un objeto **Connection**. Para su creación debemos proporcionar la *url* de la BD y, si la BD está protegida con contraseña, el *login* y *password* para acceder a ella. El formato de la *url* variará según el driver que utilicemos. Sin embargo, todas las *url* tendrán la siguiente forma general: *jdbc*:<*subprotocolo*>:<*nombre*>, con *subprotocolo* indicando el tipo de SGBD y con *nombre* indicando el nombre de la BD y aportando información adicional para la conexión.

Para conectar a una fuente ODBC de nombre *bd*, por ejemplo, utilizaremos la siguiente URL:

```
Connection con =
           DriverManager.getConnection("jdbc:odbc:bd");
```

En el caso de MySQL, si queremos conectarnos a una BD de nombre *bd* alojada en la máquina local (*localhost*) y con usuario *miguel* y contraseña *m++24*, lo haremos de la siguiente forma:

```
Connection con = DriverManager.getConnection(

"jdbc:mysql://localhost/bd", "miguel", "m++24");
```

En el caso de PostGres (notar que hemos indicado un puerto de conexión, el 5432):

```
Connection con = DriverManager.getConnection(

   "jdbc:postgresql://localhost:5432/bd", "miguel", "m++24");
```

Podemos depurar la conexión y determinar qué llamadas está realizando JDBC. Para ello haremos uso de un par de métodos que incorpora **DriverManager**. En el siguiente ejemplo se indica que las operaciones que realice JDBC se mostrarán por la salida estándar:

```
DriverManager.setLogWriter(new PrintWriter(System.out, true));
```

Una vez realizada esta llamada también podemos mostrar mensajes usando:

```
DriverManager.println("Esto es un mensaje");
```

Tema 2: Consulta a una base de datos con JDBC

2.1. Creación y ejecución de sentencias SQL

Una vez obtenida la conexión a la BD, podemos utilizarla para realizar consultas, inserción y/o borrado de datos de dicha BD. Todas estas operaciones se realizarán mediante lenguaje SQL. La clase **Statement** es la que permite realizar todas estas operaciones. La instanciación de esta clase se realiza haciendo uso del siguiente método que proporciona el objeto **Connection**:

```
Statement stmt = con.createStatement();
```

Podemos dividir las sentencias SQL en dos grupos: las que actualizan la BD y las que únicamente la consultan. En las siguientes secciones veremos cómo podemos realizar estas dos acciones.

2.2. Sentencias de consulta

Para obtener datos almacenados en la BD podemos realizar una consulta SQL (*query*). Podemos ejecutar la consulta utilizando el objeto **Statement**, pero ahora haciendo uso del método **executeQuery** al que le pasaremos una cadena con la consulta SQL. Los datos resultantes nos los devolverá como un objeto **ResultSet**.

```
ResultSet result = stmt.executeQuery(query);
```

La consulta SQL nos devolverá una tabla, que tendrá una serie de campos y un conjunto de registros, cada uno de los cuales consistirá en una tupla de valores correspondientes a los campos de la tabla.

Los campos que tenga la tabla resultante dependerán de la consulta que hagamos, de los datos que solicitemos que nos devuelva. Por ejemplo, podemos solicitar que una consulta nos devuelva los campos *expediente* y *nombre* de los alumnos o bien que nos devuelva todos los campos de la tabla *alumnos*.

Veamos el funcionamiento de las consultas SQL mediante un ejemplo:

```
String query = "SELECT * FROM ALUMNOS WHERE sexo = 'M'";
ResultSet result = stmt.executeQuery(query);
```

En esta consulta estamos solicitando todos los registros de la tabla ALUMNOS en los que el sexo sea *mujer* (M), pidiendo que nos devuelva todos los campos (indicado con *) de dicha tabla. Nos devolverá una tabla como la siguiente:

exp	nombre	sexo
1286	Amparo	M
1287	Manuela	M
1288	Lucrecia	M

Estos datos nos los devolverá como un objeto **ResultSet**. A continuación veremos cómo podemos acceder a los valores de este objeto y cómo podemos movernos por los distintos registros.

El objeto **ResultSet** dispone de un *cursor* que estará situado en el registro que podemos consultar en cada momento. Este *cursor* en un principio estará situado en una posición anterior al primer registro de la tabla. Podemos mover el cursor al siguiente registro con el método **next** del **ResultSet**. La llamada a este método nos devolverá **true** mientras pueda pasar al siguiente registro, y **false** en el caso de que ya estuviésemos en el último registro de la tabla. Para la consulta de todos los registros obtenidos utilizaremos normalmente un bucle como el siguiente:

```
while(result.next()) {
    // Leer registro
}
```

Ahora necesitamos obtener los datos del registro que marca el *cursor*, para lo cual podremos acceder a los campos de dicho registro. Esto lo haremos utilizando los métodos **getXXXX(campo)** donde **XXXX** será el tipo de datos de Java en el que queremos que nos devuelva el valor del campo. Hemos de tener en cuenta que el tipo del campo en la tabla debe ser convertible al tipo de datos Java solicitado. Para especificar el campo que queremos leer podremos utilizar bien su nombre en forma de cadena, o bien su índice que dependerá de la ordenación de los campos que devuelve la consulta. También debemos tener en cuenta que no podemos acceder al mismo campo dos veces seguidas en el mismo registro. Si lo hacemos nos dará una excepción.

Los tipos principales que podemos obtener son los siguientes:

getInt	Datos enteros
getDouble	Datos reales
getBoolean	Campos booleanos (si/no)
getString	Campos de texto
getDate	Tipo fecha

Si queremos imprimir todos los datos obtenidos de nuestra tabla ALUMNOS del ejemplo podremos hacer lo siguiente:

```
int exp;
String nombre;
String sexo;

while(result.next())
{
    exp = result.getInt("exp");
    nombre = result.getString("nombre");
    sexo = result.getString("sexo");
    System.out.println(exp + "\t" + nombre + "\t" + sexo);
}
```

Cuando un campo de un registro de una tabla no tiene asignado ningún valor, la consulta de ese valor devuelve NULL. Esta situación puede dar problemas al intentar manejar ese dato. La clase **ResultSet** dispone de un método **wasNull** que llamado después de acceder a un registro nos dice si el valor devuelto fue NULL. Esto no sucede así para los datos numéricos, ya que devuelve el valor 0. Comprobemos qué sucede en el siguiente código:

```
String sexo;

while(result.next())
{
    exp = result.getInt("exp");
    nombre = result.getString("nombre");
    sexo = result.getString("sexo");
    System.out.println(exp + "\t" + nombre.trim() + "\t" +
sexo);
}
```

La llamada al método **trim** devolverá una excepción si el objeto **nombre** es NULL. Por ello podemos realizar la siguiente modificación:

```
String sexo;

while(result.next())
{
    exp = result.getInt("exp");
    System.out.print(exp + "\t");
    nombre = result.getString("nombre");
    if (result.wasNull()) {
            System.out.print("Sin nombre asignado");
    else
            System.out.print(nombre.trim());
    sexo = result.getString("sexo");
    System.out.println("\t" + sexo);
}
```

2.3. Sentencias de actualización

La clase **statement** dispone de un método llamado **executeUpdate** el cual recibe como parámetro la cadena de caracteres que contiene la sentencia SQL a ejecutar. Este método únicamente permite realizar sentencias de

actualización de la BD: creación de tablas (CREATE), inserción (INSERT) y borrado de datos (DELETE). El método a utilizar es el siguiente:

```
stmt.executeUpdate(sentencia);
```

Vamos a ver a continuación un ejemplo de estas operaciones. Crearemos una tabla ALUMNOS en nuestra base de datos y añadiremos datos a la misma. La sentencia para la creación de la tabla será la siguiente:

```
String st_crea = "CREATE TABLE ALUMNOS (
                        exp INTEGER,
                        nombre VARCHAR(32),
                        sexo CHAR(1),
                        PRIMARY KEY (exp)
                  )";

stmt.executeUpdate(st_crea);
```

Una vez creada la tabla podremos insertar datos en ella como se muestra a continuación:

```
String st_inserta = "INSERT INTO ALUMNOS(exp, nombre)
                     VALUES(1285, 'Manu', 'M')";

stmt.executeUpdate(st_inserta);
```

Cuando tengamos datos dentro de la tabla, podremos modificarlos utilizando para ello una sentencia UPDATE:

```
String st_actualiza = "UPDATE FROM ALUMNOS
                       SET sexo = 'H'
                       WHERE exp = 1285";

stmt.executeUpdate(st_actualiza);
```

Si queremos eliminar un registro de la tabla utilizaremos una sentencia DELETE como se muestra a continuación:

```
String st_borra = "DELETE FROM ALUMNOS
                   WHERE exp = 1285";

stmt.executeUpdate(st_borra);
```

El método **executeUpdate** nos devuelve un entero que nos dice el número de registros a los que ha afectado la operación, en caso de sentencias INSERT, UPDATE y DELETE. La creación de tablas nos devuelve siempre 0.

2.4 Otras llamadas a la BD

En la interfaz **Statement** podemos observar un tercer método que podemos utilizar para la ejecución de sentencias SQL. Hasta ahora hemos visto como para la ejecución de sentencias que devuelven datos (consultas) debemos usar **executeQuery**, mientras que para las sentencias INSERT, DELETE, UPDATE e instrucciones DDL utilizamos **executeUpdate**. Sin embargo, puede haber

ocasiones en las que no conozcamos de antemano el tipo de la sentencia que vamos a utilizar (por ejemplo si la sentencia la introduce el usuario). En este caso podemos usar el método **execute**.

```
boolean hay_result = stmt.execute(sentencia);
```

Podemos ver que el método devuelve un valor *booleano*. Este valor será *true* si la sentencia ha devuelto resultados (uno o varios objetos **ResultSet**), y *false* en el caso de que sólo haya devuelto el número de registros afectados. Tras haber ejecutado la sentencia con el método anterior, para obtener estos datos devueltos proporciona una serie de métodos:

```
int n = stmt.getUpdateCount();
```

El método **getUpdateCount** nos devuelve el número de registros a los que afecta la actualización, inserción o borrado, al igual que el resultado que devolvía **executeUpdate**.

```
ResultSet rs = stmt.getResultSet();
```

El método **getResultSet** nos devolverá el objeto **ResultSet** que haya devuelto en el caso de ser una consulta, al igual que hacía **executeQuery**. Sin embargo, de esta forma nos permitirá además tener múltiples objetos **ResultSet** como resultado de una llamada. Eso puede ser necesario por ejemplo en el caso de una llamada a un procedimiento, que nos puede devolver varios resultados como veremos más adelante. Para movernos al siguiente **ResultSet** utilizaremos el siguiente método:

```
boolean hay_mas_results = stmt.getMoreResults();
```

La llamada a este método nos moverá al siguiente **ResultSet** devuelto, devolviéndonos *true* en el caso de que exista, y *false* en el caso de que no haya más resultados. Si existe, una vez nos hayamos movido podremos consultar el nuevo **ResultSet** llamando nuevamente al método **getResultSet**.

Otra llamada disponible es el método **executeBatch**. Este método nos permite enviar varias sentencias SQL a la vez. No puede contener sentencias SELECT. Devuelve un array de enteros que indicará el número de registros afectados por las sentencias SQL. Para añadir sentencias haremos uso del método **addBatch**. Un ejemplo de ejecución es el siguiente:

```
stmt.addBatch("INSERT INTO ALUMNOS(exp, nombre)
                 VALUES(1285, 'Manu', 'M')");
stmt.addBatch("INSERT INTO ALUMNOS(exp, nombre)
                 VALUES(1299, 'Miguel', 'M')");
int[] res = stmt.executeBatch();
```

2.5 Restricciones, movimientos y actualizaciones en el *ResultSet*

Cuando realizamos llamadas a BD de gran tamaño el resultado de la consulta puede ser demasiado grande y no deseable en términos de eficiencia y memoria. JDBC permite restringir el número de filas que se devolverán en el **ResultSet**. La clase **Statement** incorpora dos métodos, **getMaxRows** y **setMaxRows**, que permiten obtener e imponer dicha restricción. Por defecto, el límite es cero, indicando que no se impone la restricción. Si, por ejemplo, antes de ejecutar la consulta imponemos un límite de 30 usando el método **setMaxRows(30)**, el resultado devuelto sólo contendrá las 30 primeras filas que cumplan con los criterios de la consulta.

Hasta ahora, el manejo de los datos devueltos en una consulta se realizaba con el método **next** de **ResultSet**. Podemos manejar otros métodos para realizar un movimiento no lineal por el **ResultSet**. Es lo que se conoce como **ResultSet** arrastable. Para que esto sea posible debemos utilizar el siguiente método en la creación del **Statement**:

```
Statement createStatement (int resultSetType, int
resultSetConcurrency)
```

Los posibles valores que puede tener **resultSetType** son: **ResultSet.TYPE_FORWARD_ONLY**, **ResultSet.TYPE_SCROLL_INSENSITIVE**, **ResultSet.TYPE_SCROLL_SENSITIVE**. El primer valor es el funcionamiento por defecto: el **ResultSet** sólo se mueve hacia adelante. Los dos siguientes permiten que el resultado sea arrastable. Una característica importante en los resultados arrastables es que los cambios que se produzcan en la BD se reflejan en el resultado, aunque dichos cambios se hayan producido después de la consulta. *Esto dependerá de si el driver y/o la BD soporta este tipo de comportamiento*. En el caso de **INSENSITIVE**, el resultado no es sensible a dichos cambios y en el caso de **SENSITIVE**, sí. Los métodos que podemos utilizar para movernos por el **ResultSet** son:

next	Pasa a la siguiente fila
previous	Ídem fila anterior
last	Ídem última fila
first	Ídem primera fila
absolute(int fila)	Pasa a la fila número fila
relative(int fila)	Pasa a la fila número fila desde la actual
getRow	Devuelve la número de fila actual
isLast	Devuelve si la fila actual es la última
isFirst	Ídem la primera

El otro parámetro, **resultSetConcurrency**, puede ser uno de estos dos valores: **ResultSet.CONCUR_READ_ONLY** y **ResultSet.CONCUR_UPDATABLE**. El primero es el utilizado por defecto y no permite actualizar el resultado. El segundo permite que los cambios realizados en el **ResultSet** se actualicen en la base de datos. Si queremos modificar los datos obtenidos en una consulta y queremos reflejar esos cambios en la BD debemos crear una sentencia con, al menos, **TYPE_FORWARD_SENSITIVE** y **CONCUR_UPDATABLE**. MySQL y PostGres soportan este tipo de actualización.

Para actualizar un campo disponemos de métodos **updateXXXX**, de la misma forma que teníamos métodos **getXXXX**. Estos métodos reciben dos parámetros: el primero indica el nombre del campo (o número de orden dentro del **ResultSet**); el segundo indica el nuevo valor que tomará el campo del registro actual. Para que los cambios tengan efecto en la BD debemos llamar al método **updateRow**. El siguiente código es un ejemplo de modificación de datos:

```
rs.updateString("nombre","manolito");
rs.updateRow();
```

Si queremos desechar los cambios producidos en la fila actual (antes de llamar a **updateRow**) podemos llamar a **cancelRowUpdates**. Para borrar la fila actual tenemos el método **deleteRow**. La llamada a este método deja una fila vacía en el **ResultSet**. Si intentamos acceder a los datos de esa fila nos dará una excepción. Podemos llamar al método **rowDeleted** el cual devuelve cierto si la fila actual ha sido eliminada (método no implementado en MySQL).

Debemos tener en cuenta varias restricciones a la hora de actualizar un **ResultSet**: la sentencia SELECT que ha generado el **ResultSet** debe:

- Referenciar sólo una tabla.
- No contener una clasúsula *join* o *group by*.
- Seleccionar la clave primaria de la tabla.

Existe un registro especial al que no se puede acceder como hemos visto anteriormente, que es el registro de inserción. Este registro se utiliza para insertar nuevos registros en la tabla. Para situarnos en él deberemos llamar al método **moveToInsertRow**. Una vez situados en él deberemos asignar los datos con los métodos **updateXXXX** anteriormente descritos y una vez hecho esto llamar a **insertRow** para que el registro se inserte en la BD. Podemos volver al registro donde nos encontrábamos antes de movernos al registro de inserción llamando a **moveToCurrentRow**.

Tema 3: Información de la BD y optimización en consultas

3.1. Obtención de información propia de la base de datos

Hasta ahora asumíamos que el programador conocía la estructura de la BD a la que estaba accediendo (el nombre de las tablas, su tipo de datos, etc.). Sin embargo, en determinadas aplicaciones es posible que necesitemos obtener esa información directamente de la propia BD. Esta información se conoce con el nombre de Metadatos, datos sobre la estructura de la base de datos. Para obtener y manejar esta información disponemos de la clase **DatabaseMetaData**. Su uso es el siguiente:

```
DatabaseMetaData dbmd = con.getMetaData();
```

Ya tenemos toda la información de la base de datos. Ahora debemos manejarla. Para ello disponemos de más de 100 métodos en la clase **DatabaseMetaData**. No vamos a verlos todos, sólo los más interesantes. Los siguientes métodos proporcionan información genérica de la base de datos:

getDatabaseProductName	Devuelve el nombre de la BD
getDatabaseProductVersion	Ídem versión
getDriverName	Ídem nombre del driver
supportsResultSetType(int)	Pasándole por parámetro algunos de los tipos de ResultSet (TYPE_FORWARD_ONLY, TYPE_SCROLL_SENSITIVE, etc.), nos devuelve si la BD los soporta

Podemos obtener toda la información referente a las tablas en la BD. Para ello llamamos al siguiente método:

```
String[] tipos = {"TABLE"};
ResultSet resul = dbmd.getTables(null,null,null,tipos);
```

Este método devuelve un objeto de la clase **ResultSet**. Los parámetros pasados al método sirven para: los dos primeros para especificar el catálogo y el esquema de los que se obtendrá la información; el tercer parámetro es el nombre de la tabla a obtener y el último los tipos (también puede ser "VIEW","SYSTEM TABLE", etc.). Los tres primeros parámetros son de tipo cadena y permiten utilizar comodines. Por ejemplo, si quisiéramos obtener todas las tablas cuyo nombre empiece por Nom, pasaríamos el parámetro "Nom%". Al devolver un objeto **ResultSet** podemos visitarlo tal como lo hacíamos antes. El esquema a seguir es el siguiente:

```
while (resul.next()) {
    String nombreTabla = resul.getString("TABLE_NAME");
    ResultSet columnas = dbmd.getColumns(null,null,
                                          nombreTabla,null);
    while (columnas.next()) {
            String nombreColumna =
                            columnas.getString("COLUMN_NAME");
            int tipoDato = columnas.getInt("DATA_TYPE");
            String nombreTipo =
                            columnas.getString("TYPE_NAME");
            if (tipoDato==java.sql.Types.CHAR ||
                    tipoDato==java.sql.Types.VARCHAR) {
                        int tamanyo =
                            columnas.getString("COLUMN_SIZE");
            }
            String nulo = columnas.getString("IS_NULLABLE");
            if (nulo.equalsIgnoreCase("no")
                    System.out.println("NOT NULL");
    }
    ResultSet clavesPrimarias =
            dbmd.getPrimaryKeys(null,null,nombreTabla);
    while(clavesPrimarias.next()) {
            String nombreClave =
                    clavesPrimarias.getString("COLUMN_NAME");
    }
}
```

Como vemos en este ejemplo hemos accedido a todas las tablas de la BD. Para cada tabla obtenemos la información de sus columnas, que también es devuelta mediante un **ResultSet**. Visitamos todas las columnas y obtenemos la información de cada una de ellas y por último obtenemos las claves primarias de la tabla.

También podemos obtener los metadatos de una consulta, es decir, directamente de un **ResultSet** obtenido al llamar a un método **executeQuery**. Vamos a ver con otro ejemplo una forma distinta de acceder a los metadatos de un **ResultSet**:

```
ResultSet resul = stmt.executeQuery("SELECT * FROM ALUMNOS");

int columnas = rsmd.getColumnCount();
for (int i=1; i<=columnas; i++) {
    System.out.println("Nombre tabla="+rsmd.getTableName(i));
    System.out.println("Nombre columna="+
                            rsmd.getColumnName(i));
    System.out.println("Tipo de dato="+rsmd.getTypeName(i));
    System.out.println("Autoincremento="+
                            rsmd.isAutoIncrement(i));
}
```

Hemos indicado algunos de los métodos más útiles. Consultad el API para conocer en detalle el resto de métodos.

3.2. Optimización de sentencias

Cuando vamos a invocar una determinada sentencia repetidas veces, puede ser conveniente dejar esa sentencia preparada (precompilada) para que pueda ser ejecutada de forma más eficiente. Para hacer esto utilizaremos la interfaz **PreparedStatement**, que podrá obtenerse a partir de la conexión a la BD de la siguiente forma:

```
PreparedStatement ps = con.prepareStatement(
                   "UPDATE FROM alumnus SET sexo = 'H'
                    WHERE exp>1200 AND exp<1300");
```

Vemos que a este objeto, a diferencia del objeto **Statement** visto anteriormente, le proporcionamos la sentencia SQL en el momento de su creación, por lo que estará preparado y optimizado para la ejecución de dicha sentencia posteriormente.

Sin embargo, lo más común es que necesitemos hacer variaciones sobre la sentencia, ya que normalmente no será necesario ejecutar repetidas veces la misma sentencia exactamente, sino variaciones de ella. Por ello, este objeto nos permite parametrizar la sentencia. Estableceremos las posiciones de los parámetros con el carácter '?' dentro de la cadena de la sentencia, tal como se muestra a continuación:

```
PreparedStatement ps = con.prepareStatement(
                   "UPDATE FROM alumnus SET sexo = 'H'
                    WHERE exp > ? AND exp < ?");
```

En este caso tenemos dos parámetros, que será el número de expediente mínimo y el máximo del rango que queremos actualizar. Cuando ejecutemos esta sentencia, el sexo de los alumnos desde expediente inferior hasta expediente superior se establecerá a 'H'.

Para dar valor a estos parámetros utilizaremos los métodos **setXXX** donde **XXX** será el tipo de los datos que asignamos al parámetro (recordad los métodos del **ResultSet**), indicando el número del parámetro (que empieza desde 1) y el valor que le queremos dar. Por ejemplo, para asignar valores enteros a los parámetros de nuestro ejemplo haremos:

```
ps.setInt(1,1200);
ps.setInt(2,1300);
```

Una vez asignados los parámetros, podremos ejecutar la sentencia llamando al método **executeUpdate** (ahora sin parámetros) del objeto **PreparedStatement**:

```
int n = ps.executeUpdate();
```

Igual que en el caso de los objetos **Statement**, podremos utilizar cualquier otro de los métodos para la ejecución de sentencias, **executeQuery** o **execute**, según el tipo de sentencia que vayamos a ejecutar.

Una característica importante es que los parámetros sólo sirven para datos, es decir, no podemos sustituir el nombre de la tabla o de una columna por el signo '?'. Otra cosa a tener en cuenta es que una vez asignados los parámetros, estos no desaparecen, sino que se mantienen hasta que se vuelvan a asignar o se ejecute una llamada al método **clearParameters**.

3.3. Llamadas a procedimientos

Los procedimientos (PROCEDURES) son unidades de código que contienen un conjunto de sentencias SQL. Estos pueden servirnos para tener en nuestra BD una serie de acciones comunes predefinidas. Al igual que las sentencias preparadas, los procedimientos aumentan la eficiencia en el acceso a una BD. Los procedimientos están creados en lenguaje SQL y DDL. Otra característica es que pueden tener parámetros de entrada y/o de salida. La principal característica que diferencia a los procedimientos de las sentencias preparadas es que, una vez creado, el procedimiento reside en la BD y puede ser llamado en otras partes del código e incluso por otros programas.

En cuanto a los parámetros, podemos tener de entrada (IN: su valor no puede ser cambiado por el procedimiento), de salida (OUT: dentro del procedimiento se le asigna un valor) y de entrada/salida (INOUT: la combinación de ambas).

Por ejemplo, si cuando realizamos una venta tenemos que añadir información de la venta, y reducir el stock del producto vendido, podemos definir un procedimiento que haga esto de la siguiente forma:

```
String procedure = "CREATE PROCEDURE EFECTUAR_VENTA
    (IN cod_cliente INT, IN cod_producto INT, IN cantidad INT)
    LANGUAGE SQL
    BEGIN
    INSERT
            INTO ventas(cliente, producto, cant)
                    VALUES(cod_cliente, cod_producto,
cantidad);
            UPDATE productos
                    SET stock = stock - cantidad
                    WHERE producto = cod_producto;
    END";

stmt.executeUpdate(procedure);
```

Los procedimientos se crean mediante una sentencia DDL (un tipo de lenguaje procedural) como la del ejemplo, ejecutándola de la misma forma que cualquier sentencia DDL. En este caso tomará tres parámetros de entrada y no devuelve ningún resultado. La llamada a **executeUpdate** crea y almacena el procedimiento en la BD. A partir de ese momento podremos llamar a dicho procedimiento. Si el procedimiento no ha podido ser creado, el sistema lanza una excepción.

En muchos SGBD podemos definir procedimientos también como se muestra a continuación:

```
String procedure = "CREATE PROCEDURE VER_VENTAS_CLIENTE
   AS SELECT cliente, sum(precio) FROM ventas, productos
         WHERE ventas.producto = productos.producto
         GROUP BY cliente";

stmt.executeUpdate(procedure);
```

En este caso este procedimiento no toma parámetros de entrada pero sí que producirá resultados. Podremos tener procedimientos con distinto número de parámetros de entrada, parámetros de salida, y podrán generar incluso varios **ResultSet**.

Para llamar al procedimiento necesitaremos un tipo especial de interfaz llamada **CallableStatement**. Con esta interfaz podremos invocar sentencias para la ejecución de procedimientos como las que tenemos a continuación:

```
CallableStatement cs = con.prepareCall("{call
VER_VENTAS_CLIENTE}");

ResultSet rs = cs.executeQuery();
```

Podremos pasar parámetros de entrada de la misma forma que los pasábamos con la interfaz **PreparedStatement**:

```
CallableStatement cs = con.prepareCall("{call
EFECTUAR_VENTA[?, ?, ?]}");

cs.setInt(1, 112);
cs.setInt(2, 3380);
cs.setInt(3, 1);

cs.executeUpdate();
```

Hay que tener en cuenta que si los parámetros de entrada no son asignados antes de llamar a **executeUpdate** se lanzará una excepción. Disponemos de los mismos métodos **setXXX** que anteriormente, siendo el primer parámetro el orden del parámetro y el segundo el valor asignado.

Para hacer uso de los parámetros de salida (OUT) primero debemos registrar el tipo del parámetro. Si, por ejemplo, el segundo parámetro del procedimiento es de salida y su tipo original es VARCHAR debemos registrarlo de la siguiente forma:

```
cs.registerOutParameter(2, java.sql.Types.STRING);
```

En el caso de parámetros INOUT debemos realizar los dos pasos, registrarlo como de salida (con **registerOutParameter**) y asignarle valor (con **setXXX**). Una vez realizada la consulta a la BD podemos recuperar los valores de los parámetros OUT y INOUT con los métodos **getXXX** disponibles en la clase **CallableStatements**.

Es posible devolver un **ResultSet** como parámetro de salida de un procedimiento. Sin embargo, depende en gran medida del SGBD que estemos utilizando. Incluso algunos no soportan este tipo de parámetros.

Tema 4: Opciones avanzadas en el acceso a una BD

En este tema vamos a detallar las opciones avanzadas que se presentan en el paquete opcional **javax.sql**.

4.1. Fuentes de datos (DataSources)

Hasta el momento, cuando queríamos conectarnos a una base de datos, proporcionábamos su dirección URL. Si cambiábamos la dirección de la BD debíamos recompilar la aplicación. Si tenemos muchas aplicaciones en nuestro sistema la actualización puede ser costosa. Las fuentes de datos en JDBC permiten una utilización más flexible de las conexiones. Un objeto **DataSource** se gestiona de manera independiente de nuestra aplicación. La clase **DataSource** es una interfaz, por lo que no podemos instanciarla directamente. Debemos disponer de una clase que implemente dicha interfaz, usualmente desarrollada por el fabricante de la BD.

4.1.1. Uso de las fuentes de datos

Para hacer uso de las fuentes de datos debemos seguir dos fases. En la primera, haciendo uso de JDNI debemos asociar un objeto **DataSource** con una base de datos. El objeto **DataSource** contendrá los datos necesarios para la conexión a la BD. Este objeto lo almacenaremos en un servicio de directorios con JDNI. En principio podemos tener tantos objetos **DataSource** como queramos en nuestro sistema. Esta primera fase la suele llevar a cabo el servidor de aplicaciones. La segunda fase consiste en la conexión de un cliente a la base de datos. Cuando un cliente necesite conectarse a la base de datos, utilizará JDNI para buscar el objeto **DataSource** asociado y éste a su vez devolverá una conexión, tal como hacía la clase **DriverManager** anteriormente. En el código que se muestra a continuación se puede observar el código que implementará el cliente cuando quiera obtener una conexión.

```
// Se crea un contexto inicial (JDNI)
Context contexto = new InitialContex();
// JDNI nos proporciona el objeto DataSource
DataSource ds = (DataSource) context.lookup("jdbc/MySQL");
// Obtenemos la conexión del objeto DataSource
conexion = ds.getConnection();
```

Con esta abstracción en la conexión hemos conseguido que el cliente sólo conozca el nombre "lógico" de la BD, no su dirección URL. El cliente tampoco conoce los detalles del driver utilizado. Todo esto lo encapsula el objeto **DataSource**. Si queremos realizar algún cambio en nuestra BD (cambio de dominio, actualización, etc.) sólo tendremos que cambiar el objeto **DataSource**, no la aplicación cliente. A partir de este punto ya podemos hacer uso del API de JDBC estándar para realizar consultas a la BD.

4.2. Reserva de conexiones (ConnectionPool)

Cuando accedemos a una base de datos, el proceso de conexión es el más costoso temporalmente. En una aplicación de servidor, el número de conexiones puede llegar a ser enorme. Es por ello que se deba buscar un método para optimizar el coste temporal. Una técnica muy utilizada es la reutilización de los recursos costosos. En este caso el recurso costoso es la conexión, por lo que resulta lógico que se intente reducir el coste de conexión manteniendo un número predeterminado de conexiones creadas e ir sirviendo a las clientes que las soliciten. Cuando una aplicación termina con una conexión la libera y queda disponible para otra aplicación. También en este caso, al igual que en el caso de **DataSource**, es el servidor de aplicaciones el que se encarga de administrar y definir cuántas reservas tendremos en funcionamiento.

A pesar de ser una característica muy útil, la reserva de conexiones está aconsejada en el caso de que se cumpla lo siguiente:

- Las aplicaciones acceden a la base de datos mediante un conjunto muy reducido de cuentas de usuario. Normalmente en la conexión a una base de datos especificamos el usuario con el que accedemos. La reserva de conexiones necesita que todas las conexiones sean para el mismo usuario, por lo que para aplicaciones donde se utilicen diversas cuentas de usuario para acceder y el número de accesos por cada cuenta es reducido es desaconsejable el uso de la reserva de conexiones.
- El acceso a la base de datos se realiza durante una única sesión. Veamos el siguiente ejemplo: en una transacción estamos constantemente accediendo a la base de datos durante mucho tiempo, por ejemplo, un carrito de la compra donde por cada elemento hacemos una actualización de una tabla de la BD. En este caso está desaconsejado el uso de la reserva de conexiones, es preferible el uso de una conexión dedicada.

Una aplicación que utilice el enfoque de reserva de conexiones debe seguir el siguiente orden:

- Obtener una referencia a la reserva o a un objeto que gestione la reserva.
- Consigue una conexión de una reserva (**getConnection**)
- Utiliza la conexión. Aquí se realizarán todas las consultas y actualizaciones en la base de datos. Un aspecto muy importante aquí son las transacciones. Si una aplicación empieza una transacción y no la termina o la deshace, puede que se produzca una pérdida de consistencia de datos.
- Devuelve la conexión a la reserva. Es muy importante que la aplicación que solicita la reserva no cierre la conexión.

4.3. Transacciones

Muchas veces, cuando tengamos que realizar una serie de acciones, queremos que todas se hayan realizado correctamente, o bien que no se realice ninguna de ellas, pero no que se realicen algunas y otras no.

Podemos ver esto mediante un ejemplo, en el que se va a hacer una reserva de vuelos para ir desde Alicante a Osaka. Para hacer esto tendremos que hacer trasbordo en dos aeropuertos, por lo que tenemos que reservar un vuelo Alicante-Madrid, un vuelo Madrid-Amsterdam y un vuelo Amsterdam-Osaka. Si cualquiera de estos tres vuelos estuviese lleno y no pudiésemos reservar, no queremos reservar ninguno de los otros dos porque no nos serviría de nada. Por lo tanto, sólo nos interesa que la reserva se lleve a cabo si podemos reservar los tres vuelos.

Una transacción es un conjunto de sentencias que deben ser ejecutadas como una unidad, de forma que si una de ellas no puede realizarse, no se llevará a cabo ninguna. Dicho de otra manera, las transacciones hacen que la BD pase de un estado consistente al siguiente.

Pero para hacer esto encontramos un problema. Pensemos en nuestro ejemplo de la reserva de vuelos, en la que necesitaremos realizar las siguientes inserciones (reservas):

```
try {
    stmt.executeUpdate("INSERT
            INTO RESERVAS(pasajero, origen, destino)
            VALUES('Paquito', 'Alicante', 'Madrid')");
    stmt.executeUpdate("INSERT
            INTO RESERVAS(pasajero, origen, destino)
            VALUES('Paquito', 'Madrid', 'Amsterdam')");
    stmt.executeUpdate("INSERT
            INTO RESERVAS(pasajero, origen, destino)
            VALUES('Paquito', 'Amsterdam', 'Osaka')");
} catch(SQLException e) {
    // ¿Dónde ha fallado? ¿Qué hacemos ahora?
}
```

En este caso, vemos que si falla la reserva de uno de los tres vuelos obtendremos una excepción, pero en ese caso, ¿cómo podremos saber dónde se ha producido el fallo y hasta qué acción debemos deshacer? Con la excepción lo único que sabemos es que algo ha fallado, pero no sabremos dónde ha sido, por lo que de esta forma no podremos saber hasta qué acción debemos deshacer.

Para hacer esto de una forma limpia asegurando la consistencia de los datos, utilizaremos las operaciones de *commit* y *rollback*.

Cuando realicemos cambios en la base de datos, estos cambios se harán efectivos en ella de forma persistente cuando realicemos la operación *commit*. En el modo de operación que hemos visto hasta ahora, por defecto tenemos activado el modo *auto-commit*, de forma que siempre que ejecutamos alguna

sentencia se realiza *commit* automáticamente. Sin embargo, en el caso de las transacciones con múltiples sentencias, no nos interesará hacer estos cambios persistentes hasta haber comprobado que todos los cambios se pueden hacer de forma correcta. Para ello desactivaremos este modo con:

```
con.setAutoCommit(false);
```

Al desactivar este modo, una vez hayamos hecho las modificaciones de forma correcta, deberemos hacerlas persistentes mediante la operación *commit* llamando de forma explícita a:

```
con.commit();
```

Si por el contrario hemos obtenido algún error, no queremos que esas modificaciones se lleven a cabo finalmente en la BD, por lo que podremos deshacerlas llamando a:

```
con.rollback();
```

Por lo tanto, la operación *rollback* deshará todos los cambios que hayamos realizado para los que todavía no hubiésemos hecho *commit* para hacerlos persistentes, permitiéndonos de esta forma implementar estas transacciones de forma atómica.

Nuestro ejemplo de la reserva de vuelos debería hacerse de la siguiente forma:

```
try {

    con.setAutoCommit(false);

    stmt.executeUpdate("INSERT
            INTO RESERVAS(pasajero, origen, destino)
            VALUES('Paquito', 'Alicante', 'Madrid')");
    stmt.executeUpdate("INSERT
            INTO RESERVAS(pasajero, origen, destino)
            VALUES('Paquito', 'Madrid', 'Amsterdam')");
    stmt.executeUpdate("INSERT
            INTO RESERVAS(pasajero, origen, destino)
            VALUES('Paquito', 'Amsterdam', 'Osaka')");
    // Hemos podido reservar los tres vuelos correctamente
    con.commit();
} catch(SQLException e) {
    // Alguno de los tres ha fallado, deshacemos los cambios
    try {
            con.rollback();
    } catch(SQLException e) {};
}
```

Una característica relacionada con las transacciones es la concurrencia en el acceso a la BD. Dicho de otra forma, qué sucede cuando varios usuarios se encuentran accediendo a la vez a los mismos datos y pretenden modificarlos. Un ejemplo sencillo: tenemos una tienda y dos usuarios están accediendo al mismo disco, del cual sólo queda una unidad. El primero de los usuarios consulta el disponible, comprueba que existe una unidad y lo introduce en su

cesta de la compra. El otro usuario en ese preciso momento también está consultando el disponible, también le aparece una unidad y también intenta introducirlo en su cesta de la compra. Al segundo usuario el sistema no debería dejarle actualizar los datos que está manejando el primero.

La concurrencia es manejada por los distintos SGBD de manera distinta. PostGres permite nivel de lectura confirmada. En este nivel, si una transacción lee una fila de la BD y otra transacción cambia la misma fila, en el momento que la primera transacción intente leer de nuevo se actualiza el valor. Si las dos intentan modificar la misma fila la segunda se bloqueará hasta que la primera finalice la transacción. Esto último hay que tenerlo en cuenta para evitar bloqueos. Para saber el nivel concurrencia permitido por el SGBD podemos utilizar el siguiente método de la clase **Connection:**

```
int con.getTransactionIsolation();
```

Los valores más comunes que puede devolver este método son: Connection.**TRANSACTION_NONE** (no soporta transacciones), Connection.**TRANSACTION_READ_UNCOMMITTED** (soporta transacciones en su nivel más bajo), Connection.**TRANSACTION_READ_COMMITTED** (el nivel de PostGres antes comentado). MySQL incorpora transacciones en su última versión (4.0). Sin embargo, su funcionamiento es diferente a PostGres. Si una aplicación actualiza una fila y dentro de su transacción otra aplicación intenta modificarla, MySQL permite la actualización y no se produce ningún problema. Sin embargo, si la primera transacción no realiza commit (debido a algún error) al intentar deshacer los cambios se produce una excepción, pues no puede recuperar el estado anterior a la transacción ya que la segunda aplicación cambio la fila.

4.3.1. Puntos de almacenamiento (*savepoint*)

Los puntos de almacenamiento permiten definir puntos de control en el flujo de una transacción. Comentaremos aquí la definición y cómo podemos utilizarlos. Sin embargo, ni MySQL ni PostGres los tienen implementados. Los puntos de almacenamiento los genera un objeto **Connection**. Permiten marcar uno o más lugares de una transacción para, si algo falla, poder deshacer las acciones realizadas a partir de uno de esos puntos marcados. Los métodos a utilizar son los siguientes:

```
SavePoint con.setSavePoint("Punto de control");
SavePoint con.setSavePoint();
```

Este método permite marcar un punto de almacenamiento. También disponemos de una forma de deshacer los cambios producidos a partir de un punto de almacenamiento:

```
con.rollback (SavePoint sp);
```

En el caso que de una excepción se produzca, podemos consultar el punto de almacenamiento para determinar si las acciones anteriores a la definición del

punto de control se han realizado con éxito. El siguiente código muestra un ejemplo de utilización.

```
SavePoint sp = null;
boolean realizar_commit = false;
try {
    insertarTabla1 (datos);
    realizar_commit = true;
    sp = con.setSavePoint();
    insertarTabla2 (datos2);
} catch {
    // Si sp es null es que falló la primera inserción
    if (sp==null) {
            con.rollback();
    }
    else {
            con.rollback(sp);
    }
}
finally {
    if (realizar_commit) {
            con.commit();
    }
}
```

Vamos a analizar con detalle el código. Se declaran dos variables. La primera es el punto de almacenamiento. La segunda es una variable de control que nos va a permitir saber si se realizó la primera acción en la base de datos. El código dentro de la sentencia **try** llama a dos métodos, **insertarTabla1** e **insertarTabla2**. Estos métodos se encargan de actualizar la base de datos, con la particularidad de que la primera inserción es más importante que la segunda. Además, si la primera no ha tenido éxito se deshace toda la transacción, pero si ha tenido éxito y la segunda no, podemos deshacer sólo la segunda. En este caso es cuando tiene utilidad los puntos de almacenamiento. Hemos colocado un punto de almacenamiento después de la llamada a la primera inserción. Si se ha producido una excepción, sabemos que si el punto de almacenamiento creado es nulo entonces la excepción se produjo en la primera inserción. Si no es nulo, la primera inserción finalizó con éxito y la segunda provocó la excepción. Por ello llamamos al método **roolback** con el punto de almacenamiento como parámetro. Esta acción provoca que se deshagan las acciones realizadas en la segunda inserción. Por último, en el bloque **finally**, realizamos **commit** si se ha realizado la primera inserción con éxito.

4.4. RowSet

Un **RowSet** es un componente que se ajusta a JavaBean y que encapsula el acceso a bases de datos incluido el resultado. Permite olvidarnos de los objetos *Connection*, *ResultSet*, *Statement*, etc. Todas las funcionalidades de estas clases las implementa RowSet. Rowset es un interfaz, por lo que debe ser implementada antes de instanciarse.

La clase mantiene los métodos utilizados por el *ResultSet* para acceder a los datos y a los registros. Disponemos de *next, previous, getXXX, updateXXX* tal como los usábamos en *ResultSet*. Los métodos descritos a continuación sirven para configurar el acceso a una base de datos:

```
//Utilizamos una de las implementaciones como ejemplo
CachedRowSet crs = new CachedRowSet ();
// Asignamos la URL de nuestra base de datos
crs.setUrl ("jdbc:mysql://localhost/bd");
// Asignamos el usuario y la contraseña
crs.setUsername ("miguel");
crs.setPassword ("miguel");
```

A partir de este momento ya podemos utilizar el RowSet y realizar consultas a la BD. Debemos tener en cuenta que debemos cargar el driver de la base de datos tal como lo hacíamos anteriormente:

```
Class.forName ("org.gjt.mm.mysql.Driver");
```

Para ejecutar un comando, primero asignamos el comando y después lo ejecutamos.

```
crs.setCommand ("Select * from vuelo");
crs.execute ();
```

Ya podemos movernos tal como lo hacíamos en el *ResultSet*. También podemos utilizar los métodos *updateXXX* para actualizar los datos del *RowSet*. Una vez realizada la llamada a **updateRow** debemos realizar una llamada al método **acceptChanges**. Este método se encargará de actualizar los datos en la fuente de datos. Esto último dependerá si lo soporta el driver. También permite la preparación de sentencias, tal como se hacía con las sentencias preparadas. Sin embargo, aquí no es necesario el uso de objetos adicionales.

```
crs.setCommand ("select numero from vuelo where aero_inic=?");
crs.setString (1, "ALC");
```

Sun ha desarrollado tres ejemplos de implementaciones que serán las que veamos en esta sección: CachedRowSet, JdbcRowSet y WebRowSet.

4.4.1. CachedRowSet

Esta implementación permite mantener en memoria los resultados obtenidos de una consulta. Por ello es desaconsejable su uso cuando la consulta genera demasiados datos. Permite el scrolling y la actualización, a pesar de que el driver no los soporte. Esta es una de las ventajas fundamentales de esta implementación. Otra de sus funcionalidades es la posibilidad de recibir datos de fuentes distintas a las bases de datos (hojas de cálculo, ficheros de texto tabulados, etc.). Cuando invocamos al método **execute** se realiza una conexión a la base de datos, se recibe la información resultado de realizar la consulta y se cierra la conexión. Si realizamos una actualización de los datos el objeto realiza una nueva conexión y actualiza los datos.

4.4.2. JdbcRowSet

Esta implementación es la más sencilla de las tres. Simplemente es una capa por encima de *ResultSet* para que parezca y pueda ser utilizado como un JavaBean.

4.4.3. WebRowSet

El objetivo principal de esta implementación es la lectura y escritura de datos en formato XML. En una aplicación cliente-servidor, la parte del servidor es la que se encarga de consultar la base de datos y crear un objeto **WebRowSet** para a continuación crear un fichero XML que es el que se le envía al cliente. El cliente recibe los datos en formato XML, pero los puede manejar como si fuera un resultado de una consulta. Cuando actualiza algún dato y llama al método *acceptChanges* la información se envía de vuelta al servidor en formato XML y el servidor es el que se encarga de escribir en la base de datos. Esta implementación tiene sentido cuando trabajamos detrás de cortafuegos y/o proxys, debido a que las conexiones directas a base de datos no son siempre posibles. Esta clase dispone de dos métodos para leer y escribir datos en XML, ambos asociados a objetos que nos permiten leer y escribir.

```
java.io.FileWriter FW = new java.io.FileWriter("ejemplo.xml");
crs.writeXML (FW);
```

Cuando llamamos al método **writeXML** el contenido del RowSet se escribe en el fichero *ejemplo.xml* en formato XML. De la misma forma disponemos del método **readXML** asociado a un objeto de la clase *java.io.FileReader*. El método de escritura no funciona en la actual implementación.

4.4.4. Manejo de eventos

Otro característica importante de los *RowSet* son los eventos. Podemos definir una clase que actúe de escuchante, para que cuando se produzca un evento se realicen una serie de acciones definidas en el escuchante. El API JDBC especifica la interfaz *javax.sql.RowSetListener*. La interfaz permite especificar tres métodos, asociados a tres eventos distintos:

```
public void cursorMoved (RowSetEvent evento)
public void rowChanged (RowSetEvent evento)
public void rowSetChanged (RowSetEvent evento)
```

El primer método se invocará cuando el cursor se mueva. El cursor es un elemento que indica al objeto *RowSet* en qué fila se encuentra dentro del conjunto de resultado. Dentro de este método podemos realizar las acciones que queramos: notificar el evento a otro objeto, realizar alguna comprobación, cambiar la base de datos, etc. El segundo método se invocará cuando se cambie una fila y el tercero cuando cambie el *RowSet*. A continuación se muestra un ejemplo de definición de uno de los métodos:

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


public class Escuchante implements RowSetListener {
    public void cursorMoved (RowSetEvent evento) {
            System.out.println("Se ha movido el cursor");
    }
.
.
.
```

Para asociar el escuchante a una clase RowSet debemos hacer uso de la anterior clase:

```
Escuchante escucha = new Escuchante();
//crs es un objeto de tipo RowSet
crs.addRowSetListener(escucha);
```

Podemos añadir tantos escuchantes como queramos.

Introducción a JDBC

1. Vamos a realizar una clase que contendrá los datos de la conexión con la base de datos. Al tener esta clase por separado nos permitirá tener centralizado los datos de la conexión en un único sitio y realizar el resto de la aplicación independiente de dichos datos. Vamos a utilizar el SGBD MySQL. La base de datos a utilizar ya está creada. Esta base de datos trata, de forma simplificada, de gestionar los vuelos de una agencia de viajes. La estructura de la BD es la siguiente:

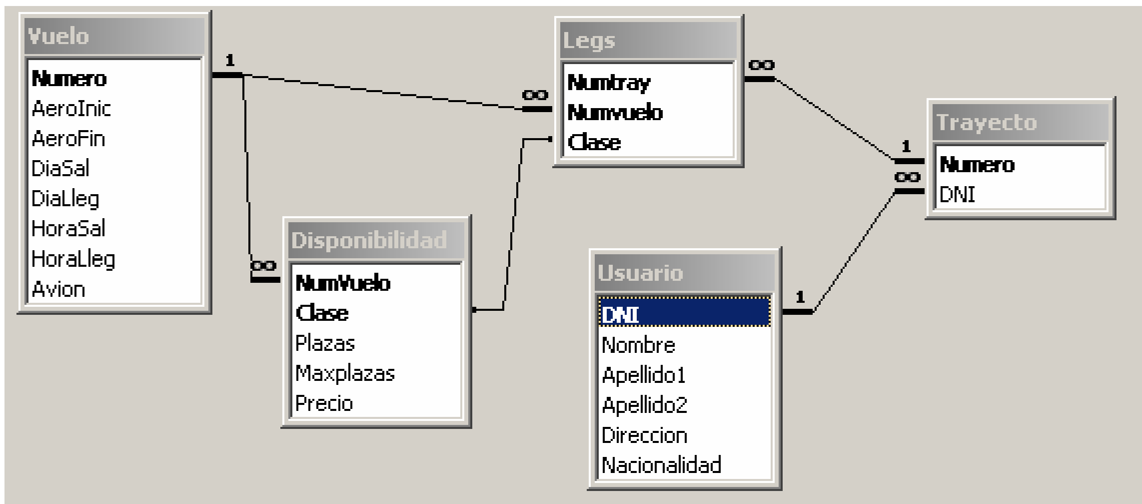

Figura 1. Tablas y relaciones en la base de datos a utilizar.

Si queréis utilizarla en vuestra casa debéis seguir los siguientes pasos:

- Descargad el fichero *Viajes.tgz*
- Lo descomprimís en el directorio donde se encuentren las bases de datos de MySQL. Por defecto está en */var/lib/mysql*. Para descomprimir cambiad al siguiente directorio *cd /var/lib/mysql* y descomprimís con *tar -xvzf Viajes.tgz*
- La base de datos ya está activa y se puede utilizar normalmente.

Tenéis disponible la misma base de datos en Access. Aquí os dejo instrucciones de cómo podéis configurar el gestor de ODBC para acceder a estos datos.

- Copiad el fichero *viajes.mdb* a cualquier directorio de vuestro ordenador.
- Configurad ODBC para que sepa dónde está este fichero:
    - Id a *Panel de Control* y abrid *Fuentes de datos ODBC*.
    - Agregad una nueva DSN (nombre de fuentes de datos) de usuario, que nos permitirá acceder a nuestra BD.
    - Seleccionad el driver para acceder a Access. Pulsad en *Finalizar.*
    - Dad el nombre *Viajes* y si lo deseáis una descripción. Pinchad en *Seleccionar...* y seleccionad el fichero *viajes.mdb* de vuestro ordenador.

- Pinchad en *Aceptar*. Esta BD ya está disponible para su uso mediante ODBC.

Para este primer ejercicio utilizad el fichero *Consulta.java*. Debéis rellenar los métodos descritos. Capturar también las excepciones.

2. Haciendo uso de la clase anterior nos disponemos a realizar una clase que se conecte a la BD y realice un listado de todos los vuelos y usuarios (con toda la información para cada registro). Podéis utilizar la plantilla del fichero *EjJDBC1.java* para realizar el ejercicio. Observad que no sólo debéis rellenad los métodos, también debéis incluir la clase **Conecta** y llamar a los métodos correspondientes. Realizad, por último, un nuevo método que liste la tabla *Disponibilidad* ordenada por precio.

3. Ahora vamos a practicar con las consultas en varias tablas. Vais a realizar los siguientes métodos, incorporándolos a la clase desarrollada:

- Un método que le pasemos como parámetro una cadena que se corresponderá con el primer apellido de un cliente y devolverá un **ResultSet** como resultado de realizar una consulta a la BD preguntando por todos los vuelos reservados por dicho cliente.

- Un método que reciba dos parámetros de entrada, el aeropuerto de partida y el final. El método devolverá el resultado de consultar la BD diciéndonos la disponibilidad de vuelos para ese par de aeropuertos. El método también listará los siguientes datos: una primera línea con el aeropuerto inicio, aeropuerto fin, hora salida y hora fin; en las siguientes líneas y ordenado por el campo *clase*, la clase, el número de plazas disponibles, el número máximo de plazas y el precio.

- Por último, un método que devuelva y muestre por pantalla todos los posibles recorridos a realizar entre un par de aeropuertos. Pasando como parámetros dos aeropuertos, el método irá interrogando a la BD para comprobar qué posibles combinaciones se pueden realizar. Tened en cuenta que las horas de salida y llegada han de tenerse en cuenta para aceptar ese tramo.

Movimientos y restricciones en la consulta

Vamos a realizar una clase que nos permita movernos y actualizar el **ResultSet.** Para ello, realizaremos un menú que permita consultar, reservar y anular vuelos. Haced uso de la plantilla *EjJDBC3.java*. Los pasos a seguir serán los siguientes:

- Lo primero a realizar es la consulta. Implementad el método *consultaDisp*. Este método va a mostrar los datos de los vuelos junto con la disponibilidad de plazas. Debéis crear un menú que permita ir avanzando de uno en uno por los vuelos. El menú debe permitir avanzar, retroceder, ir al último registro e ir al primero. La creación del objeto **Statement** debe configurarse para que permita el movimiento por el **ResultSet**. Una vez leído un registro, si se intenta volver a leer dará una excepción: debemos realizar un *next* seguido de un *previous* (por ejemplo). La información a mostrar será: número de vuelo, aeropuerto inicio y destino, fecha y hora de inicio y final, la clase y las plazas disponibles.

- En el método *muestraTray* se pedirá al usuario que introduzca el apellido del cliente. A continuación mostrará todos los trayectos de ese usuario, mostrando para cada trayecto los vuelos parciales reservados. Realizaréis también un menú como el anterior para moverse por todos los registros. Aquí debéis realizar dos consultas. La primera para sacar todos los trayectos del cliente. Esta consulta será por la que nos movamos. La otra consulta será para mostrar los vuelos parciales de ese trayecto (legs). La información a mostrar será: Nombre y apellidos del cliente, una línea por cada vuelo parcial, mostrando el aeropuerto de inicio, el de destino y la fecha y hora de salida.

- El método *modificaCliente* primero hace una consulta a la tabla *usuario* mostrando todos los datos del primer cliente. Tenéis que hacer un menú para moveros por el **ResultSet** y permitir modificar los datos del cliente que se muestra en ese momento. Si seleccionamos modificar los datos del cliente nos irá pidiendo los datos y por último se actualizará la fila correspondiente

- El último método a implementar, *anulaTray*, mostrará la información de trayectos de un usuario particular. Para ello podemos hacer uso del método anterior, añadiendo una nueva opción que será la de anular reserva. Esta opción se añadirá en el menú y cuando la seleccionemos borrará todos los vuelos parciales (legs) que se correspondan con el trayecto actual. Debéis realizar dos cosas: eliminar los legs y actualizar la tabla disponibilidad, pues al eliminar un leg dispondremos de una plaza más.

Metadatos y sentencias preparadas

En el primer ejercicio de esta sesión vamos a realizar una clase que nos permita copiar la estructura de tablas y los datos de un SGBD a otro (en nuestro caso, de MySQL a PostGres). Para ello vamos a seguir los siguientes pasos:

- Cread una nueva clase de conexión que nos permita conectarnos con PostGres. Podéis basaros en la realizada en la anterior sesión, modificando los datos de conexión.

- Cread una clase que copie la estructura de tablas desde MySQL a PostGres. La BD ya está creada en PostGres con el nombre Viajes. La clase a desarrollar contendrá un método *crear* que obtendrá los metadatos de MySQL y ejecutará un comando CREATE TABLE para cada tabla.

- Una vez creadas todas las tablas procedemos a copiar los datos. Para ello, creamos un nuevo método dentro de la anterior clase que realizará una consulta a la BD en MySQL e insertará dichos datos a la correspondiente en PostGres. Para simplificar os fijáis en los tipos de los campos. En este método lo correcto sería utilizar el método **executeBatch** con el correspondiente **addBatch**, tal como hemos visto en teoría. Sin embargo, el driver de PostGres no tiene implementados estos métodos. Realizad un **executeUpdate** por cada INSERT.

- Una vez copiados comprobad si la clase desarrollada en la sesión anterior funciona con PostGres. Simplemente cambiad la clase de conexión a la nueva creada para PostGres, compiláis y ejecutáis.

En el siguiente ejercicio se trata de hacer uso de las sentencias preparadas. Utilizad el ejercicio desarrollado en la sesión anterior. Modificad el método *MuestraTray* para que trabaje con sentencias preparadas. En este método se realizan dos consultas. La segunda consulta se repite muchas veces cambiando sólo los datos de la consulta. Este tipo de consulta son las ideales para hacerlas preparadas. Cread la sentencia preparada en el constructor de la clase y asignar los parámetros tal como hemos visto en teoría.

Transacciones y RowSet

La primera parte de este bloque de ejercicios consiste en comprobar el funcionamiento de las transacciones. Para ello vamos a realizar dos clases que accederán a la misma tabla de forma concurrente y así comprobaremos el funcionamiento en las transacciones. Las dos clases accederán a la tabla *vuelo*. De la tabla *vuelo* consultarán el registro del vuelo número 9 y 10, y actualizarán el valor del aeropuerto de inicio.

Ambas clases tendrán un método *cambio* que primero realizará una consulta del vuelo número 9 y mostrará el valor de su aeropuerto de inicio. A continuación actualizará el valor de su aeropuerto de inicio. La sentencia SQL puede ser la siguiente:

```
Update vuelo set aeroinic='ALC2' where numero=9
```

A continuación se quedará esperando que el usuario pulse una tecla. Podéis utilizar el siguiente código:

```
int pp = System.in.read();
```

Por último modificará el valor del aeropuerto inicio del registro número 10 a MAD2. Recordad que antes de la primera actualización se debe llamar al método **setAutoCommit** y después de la última **commit** para que surtan efecto los cambios. También debéis llamar a **rollback** si se produce una excepción. Para probar este ejercicio se llamarán a las dos clases desde sesiones distintas. Probad primero con PostGres. La primera clase actualizará el valor de la fila y se quedará esperando que pulsemos una tecla. En ese momento ejecutamos la segunda. Comprobad que la segunda se queda esperando, pero aunque pulsemos no continúa su ejecución, pues está esperando que se libere el bloqueo producido por la primera. Cuando pulsamos en la primera aplicación las dos finalizan.

Otra prueba a realizar es el interbloqueo. Para ello simplemente cambiáis el orden de una de las clases, que primero acceda a la fila con número 10 y después a la 9. Veréis que PostGres detecta el interbloqueo.

Cambiad ahora y comprobad el funcionamiento con MySQL. Con el primer ejemplo, cambiad el final de la primera clase para que no haga **commit** sino **rollback**. El sistema debe dar un error.

En la segunda parte de la sesión vamos a utilizar los RowSet. Descargad la implementación de RowSet que ha realizado Sun. Se encuentra disponible en la página de recursos. Modificad el ejercicio desarrollado en la sesión 2 para que trabaje con un objeto de la clase *CachedRowSet* (cread una nueva clase). Cread un nuevo método en la misma clase que utilice un *WebRowSet* y que tenga la misma funcionalidad que *MuestraTray*, pero que el resultado lo envíe a un fichero en formato XML. Comprobad el resultado que se obtiene. No es posible leer el fichero XML de vuelta, pues el DTD no es correcto.