Unidad 3

Bases de datos relacionales

Acceso a datos

3

Índice

Introducción

En este tema estudiaremos diversos conceptos que necesitaremos para el empleo de las bases de datos.

Comenzaremos con el estudio de los conectores, API, y cuál es su función en el empleo de la base de datos. En este mismo punto conoceremos cuál es la función de los *drivers* y cómo se relaciona con los conectores que explicaremos.

Siguiendo esto estudiaremos JDBC y su relación con los *drivers*, para después desviarnos al estudio de SQL y sus variantes.

Se mostrarán diversas operaciones que podremos realizar para interactuar con nuestra base de datos y continuaremos, hasta el final de este tema, desglosando las operaciones, y las diferentes sentencias y métodos que se emplean en dichas operaciones, que podemos emplear para con nuestra base de datos.

Al finalizar esta unidad

- Habremos distinguido los distintos tipos de conectores junto con la función de API y de los *drivers*.
- Conoceremos detalles sobre las bases de datos relacionales y los problemas de desfase que se producen al intentar incluir objetos en ellas.
- Sabremos emplear distintas funciones de SQL y JDBC para la consulta de bases de datos.
- Habremos definidos cuáles son las funciones básicas de JDBC.
- Comprenderemos cómo emplear sentencias SQL.
- Habremos estudiado el empleo de transacciones y lotes.

3.1. Conectores

Los SGBD, sistemas gestores de bases de datos, poseen sus propios lenguajes para el almacenamiento en función de sus tipos, relacionales, XML, de objetos, etc., mientras que las aplicaciones suelen emplear un lenguaje más general como Java, por lo que para que puedan trabajar en conjunción requieren de un conector API.

Imagen 1. Ejemplo de comunicación entre aplicación y SGBD usando un conector API

3.2. Conectores para bases de datos relacionales

En las bases de datos relacionales, las más empleadas, se emplea SQL, ya sea como tal o en una de sus versiones como PL/SQL. Los *drivers* ayudan a los conectores a traducir las particularidades de cada base de datos:

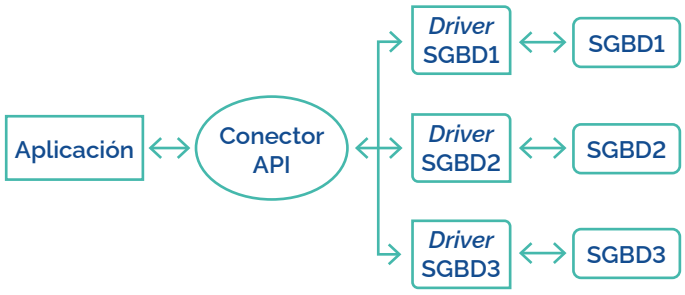


Imagen 2. Conexión mediante drivers

Se forma una arquitectura que entrelaza API y *drivers* en un sistema de traducción doble. Existen múltiples arquitecturas como la original ODBC, Open DataBase Connectivity, de Microsoft, que en la actualidad usan Linux y Unix, la cual introducía API que empleaban el lenguaje C. En la actualidad existen múltiples estructuras y *drivers* para cualquier necesidad. Java emplea una variante de ODBC llamada JDBC, que se encuentra en la mayoría de las bases de datos, y, las que no, cuentan con *drivers* de ODBC.

JDBC también posee *drivers* para elementos fuera de la base de datos, como son los ficheros CSV y XML.

Existen API específicas para ciertas bases de datos que no requieren de *drivers* Como es el hecho de PHP para Apache que permite el acceso a MySQL. Este intento de no usar *drivers* proviene del hecho de que, si bien estos junto con API nos dan acceso, a la vez crean una mayor complejidad y, por lo tanto, provocan un menor rendimiento en comparación con el empleo solamente de API.

3.3. Acceso a resultados de consultas sobre bases de datos relacionales mediante conectores

Los conectores, entre otras operaciones, permiten realizar consultas en las bases de datos relacionales.

En las bases relacionales el método empleado para almacenar información es la tabla, por lo que los conectores, ante una consulta, responden fila a fila empleando un objeto a modo de iterador o cursor. Por otro lado, una consulta sin un conector devolverá un conjunto de filas y *recordset*.

Dependiendo de la base de datos es posible que la consulta no se desarrolle de la misma forma, pero cuentan con una estructura común de cuatro pasos:

- Abrir una conexión.
- Realizar la consulta.
- Emplear el iterador o cursor.
- Cierre final

3.4. Desfase objeto-relacional

Llamamos desfase objeto-relacional a los problemas y dificultades que surgen al intentar almacenar un objeto en una base de datos relacional, ya que el formato de tabla no es apto para desempeñar esta labor. Estos problemas se agravan se el objeto que deseamos introducir es complejo; generalmente en estructura grafo, posee referencias a otros objetos o es una colección de objetos relacionales.

Con el fin de evitar estos problemas podemos emplear directamente una base de datos de objetos o emplear las herramientas ORM, correspondencia objeto-relacional.

3.5. Java Database Connectivity

JDBC cuenta con un API, que se puede localizar en java.sql y numerosos *drivers*, algunos de ellos no destinados a bases de datos, sino a otros conectores como el que existe para ODBC, aunque no existe uno específico para JDBC.

3.6. Operaciones básicas con JDBC

Empleando JDBC podemos llevar a cabo ciertas operaciones basadas en SQL. Al realizarse mediante SQL deberemos distinguir claramente entre las que se realizan en el sublenguaje DML, *data Manipulation language*, y las que lo hagan en DDL, *Data definition language*.

- En **DML** encontraremos las sentencias:
  - SELECT
    - Se realiza con `executeQuery()`, devuelve filas en `ResultSet`.
  - UPDATE
  - DELETE
  - INSERT
    - Estas tres se realizan con `executeUpdate`.
- En **DDL** Todas las sentencias se ejecutan con `execute()`.

Ejemplo de operaciones de JDBC

Código	Función
`Class.forName (Nombre del driver);`	Carga el *driver* no es necesario desde JDBC 4.0 o Java SE 6.
`Connection c = DriverManager.getConnection;`	Crea una conexión
`Statement s = c.createStatement();`	Crea una sentencia

`ResultSet rs= s.executeQuery(…Consulta realizada);`
`while(rs.next()){…}`
`Rs.close();`
	Ejecuta la consulta, obtiene los resultados y después la cierra.

`int res = s.executeUpdate; (DML)` `boolean res = s.execute; (DDL)`	
`s.close();`	Cierra la sentencia
`c.close();`	Cierra la conexión

3.6.1. Apertura y cierre de conexiones

En JDBC los *drivers* se almacenan en ficheros jar, a los que podemos conectarnos con la clase DriverManager empleando `getConnection(String URL de la conexión)`. La URL contiene identificadores de acceso, por ejemplo, en MySQL la URL es la siguiente:

Jdbc:mysql:host:puerto/basedatos

El puerto suele ser 3306. Si empleamos esta sentencia nos encontraremos cargando los *drivers* para JDBC en sus últimas versiones a partir de la 4.0 y podremos escoger el deseado. Devolverá Connection si la conexión se establece apropiadamente. Por ejemplo, el *driver* de MySQL 8.0 es com.mysql.cj.jdbc.Driver.

Si deseamos añadir un *driver* en JDBC deberemos añadir el fichero jar.

En el caso de que no se establezca conexión podremos emplear una sentencia más antigua, generalmente no usada, como es: Class.forName(Nombre de la clase);

Para conectarse a la base de datos con MySQL es necesario aportar en el código cierta información como es:

- **Servidor.**
  - Host.
  - Puerto.
- **Nombre de la base de datos.**
- **Autentificación.**
  - Usuario.
  - Contraseña.

Podemos abrir bloques de recursos con `try` con el fin de que se cierre correctamente al tiempo que se evita emplear el cierre `close()`.

3.6.2. La interfaz Statement

Para poder ejecutar sentencias SQL es necesario emplear Statement con el método getStatement.

Métodos de STATEMENT	
Método	Funcionalidad
`ResultSet executeQuery (String sql)`	Ejecutar una consulta, SELECT, devolviendo un ResultSet para acceder a los resultados.
`ResultSet getResultSet()`	Conjunto de resultados de SELECT, u otra sentencia, a modo de procedimientos almacenados.
`Int executeUpdate(String sql)`	Operaciones de modificación, INSER, UPDATE y DELETE, devolviendo el número de filas afectadas.
`Boolean execute(String sql)`	Para cualquier consulta, en especial para las DDL donde nos devolverá el valor FALSE. Podemos emplear también `executeUpdate` para DDL, devolviendo 0.
`Void close()`	Cierre de Statement

3.6.3. Ejecución de sentencias para modificar contenidos de la base de datos

DDL es un lenguaje de definición de datos con múltiples sentencias para modificar datos que se ejecuta a través del método `execute()`.

```
s.execute("CREATE TABLE USUARIOS(NOMBRE CHAR(30) NOT NULL, CP CHAR(5) NOT NULL.)
```

Las más empleadas son INSER, UPDATE o DELETE.

Igualmente podemos emplear las sentencias anteriores mediante el método `executeUpdate()`.

```
Int TablaFilas = s.executeUpdate(
    "INSERT INTO Usuarios (CP, NUMUSER) VALUES"
    + "('57381','552'),"
    + "('57381','552'),");
```

3.6.4. Ejecución de consultas y manejo de ResultSet

La ejecución de `executeQuery()` devolverá como respuesta `ResultSet` devolverá un conjunto de filas, aunque existen métodos para que la respuesta también muestre distintas columnas señalando estas por su posición o nombre.

En el caso de emplear `sentence` tendremos que pasar las columnas, una a una mediante `next()`, con `ResultSet` tenemos diferentes métodos.

- `boolean next()`: mueve una columna hacia adelante y devuelve TRUE, en caso de encontrarse ya en la última columna devuelve el valor de FALSE.
- `getXXX(int)` o `getXXX(String)`: devuelve el valor de la columna especificada, algunas de las funciones son las siguientes:
  - getInt(), getString(), getDate(), etc.
- `close()`: cierra ResultSet.

También nos podemos encontrar resultados *scrollable* de ResultSet empleando parámetros especiales con Statement o PreparedStatement:

Parámetros `tipo`:

- `ResultSet.TYPE_FORWARD_ONLY`: tipo por defecto.
- `ResultSet.TYPE_SCROLL_INSENSITIVE`: resultados *scrollable* en el que no se reflejan los cambios que ocurren.
- `ResultSet.TYPE_SCROLL_SENSITIVE`: resultados *scrollable* en el que se reflejan los cambios que ocurren.

Existen otros métodos de seleccionar distintas columnas:

Ejemplos de métodos de desplazamientos entre columnas

Método	Funcionalidad

`Boolean next()`
`boolean previous()`
`boolean first()`
`boolean last()`
`void beforeFirst()`
`void afterLast()`
`boolean absolute(int pos)`
`boolean isFirst()`
`boolean isLast()`
`boolean isBeforeFirst()`
`boolean isAfterLast()`
`int getRow()`
	Permite moverse entre las columnas o desplazarse a columnas especificas.

`getXXX(int)` `getXXX(String):`	Devuelve el valor de la columna especificada, algunas de las funciones son getInt(), getString(), getDate(), etc.

3.7. Sentencias preparadas

Podemos simplificar las sentencias mediante expresiones con variables, pero esto generaría sus propios problemas, como los de seguridad y rendimiento, ya que sería un punto de falla explotable por técnicas de inyección de código y en cuanto al rendimiento este tipo de consulta conllevará una compilación de cada consulta para la creación de planes de ejecución semejantes.

Con el fin de evitar este problema podemos emplear sentencias preparadas las cuales, mediante marcadores, o *placeholders*, generalmente "?", realizan una precompilación usable múltiples veces con tan solo modificar los valores y volver a ejecutar.

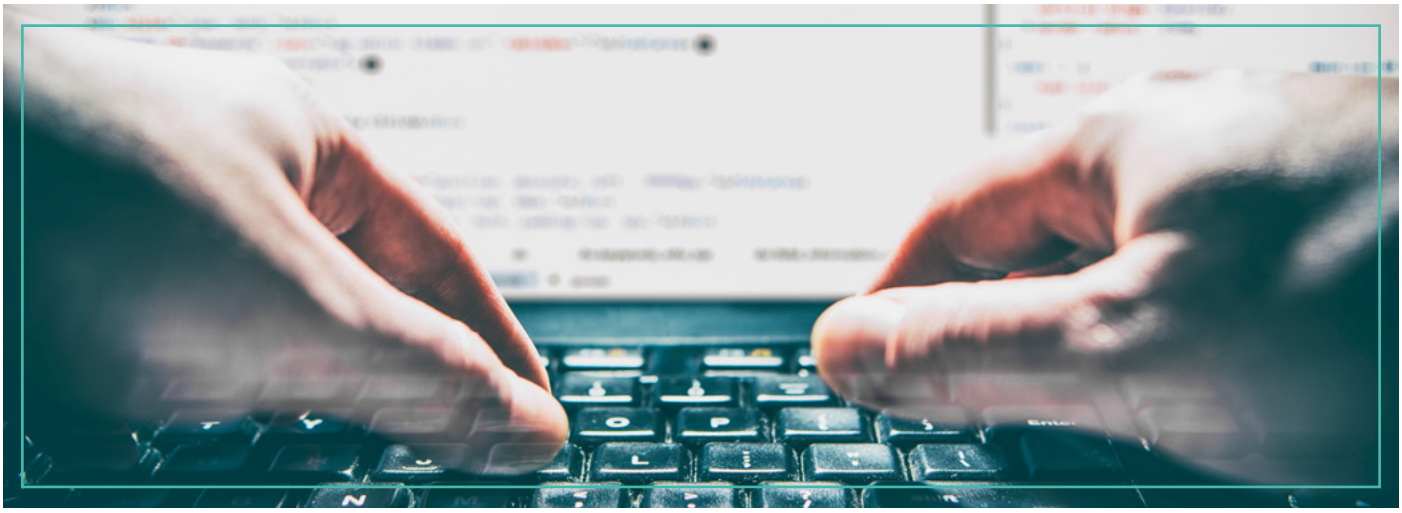
Las consultas se realizan con PreparedStatement mediante el método getPreparedStatement de Connection. A este método añadimos el marcador "?".

PreparedStatement comparte muchos métodos de Statement, pero también posee algunos propios:

Métodos de PreparedStatement	
Método	**Funcionalidad**

`ResultSet executeQuery()`
`int executeUpdate()`
`boolean execute()`
	No poseen parámetros de consulta.

`setXXX(int pos, XXX valor)`	Asigna un valor a un marcador, 1 por defecto para el primer marcador.
`setNull(int pos, int XXX)`	Asigna NULL a una columna, se debe indicar uno de los tipos de java.sql.Types.



3.8. Transacciones

En ocasiones es necesario llevar a cabo más de una operación de manera simultánea, en estos casos se emplean las transacciones. Las transacciones nos permiten establecer diversas operaciones simultaneas de manera completa, o en caso de error, eliminar cualquier modificación, de modo que se complete el conjunto completo o, en su defecto, no se realice ningún cambio a la base de datos. Las transacciones siguen las características ACID (*atomic, consistent, isolated, durable*).

Debido a su amplio uso todos los SGBD relacionales poseen estructuras para las transacciones, aunque con sus particulares en cada uno. JDBC posee su propia manera, pero admite la realización empleada por SQL con la siguiente sintaxis:

```
> START TRANSACTION
> Operación 1
> Operación 2
...
COMMIT
```

Si se emplea la sentencia `ROLLBACK` podemos anular la transacción y revertir los posibles cambios. Es posible, en raros casos, que `COMMIT` falle, en estos casos la sentencia se anularía y se revertirán los fallos. Podemos mediante los métodos de `Connection` afectar a las transacciones.

Métodos de Connection para transacciones

Método	Funcionalidad
`Void setAutoCommit(boolean autoCommit)`	Con `autoCommit=false` inicia la transacción como START TRANSACTION
`Void commit()`	Equivale a COMMIT, para ejecutar más sentencias SQL no en transacción se requiere `setAutoCommit(true)`
`Void rolldack()`	Descarta los cambios realizados, generalmente como respuesta a `SQLException`

`SQLException` se produce al fallar una sentencia SQL, y podremos abortarla mediante la sentencia `rollback()`.

3.9. Valores de claves autogeneradas

Ciertos elementos, como las facturas o solicitudes, requieren de un código único, el cual no es rentable introducir manualmente. En estos casos podemos emplear claves autoincrementables, llamadas autogeneradas en JDBC, las cuales se generan mediante una tabla con un código inicial que aumenta con cada fila generada, siendo, por tanto, un código numérico único para cada una de las filas sin posibilidad de coincidencia.

Esta tabla se puede realizar en casi todos los lenguajes con mínimas variaciones. Oracle se distingue por emplear secuencias para la generación de los códigos.

Para poder introducir las claves autogeneradas empleando JDBC podemos emplear métodos de Statement y Connection:

Métodos para la inserción de claves autogeneradas

Método	Funcionalidad
`Int executeUpdate( Strin sql, Int autoGeneratedKeys )`	Valores de `autoGenerated`: `Statement. RETURN_GENERATED_KEYS` `Statement.NO_GENERATED_KEYS` No son compatibles con `PreparedStatement`
`ResultSet getGeneratedKeys()`	Devuelve una clave autogenerada como `ResultSet`. Se puede emplear con `PreparedStatement`
`PreparedStatement prepareStatement( String sql, Int autoGeneratedKeys)`	Prepara la clave autogenerada y se recupera con el valor `PreparedStatement.RETURN_GENERATED_KEYS` para `autoGeneratedKeys`.

3.10. Llamadas a procedimientos y funciones almacenados

La escritura en el código de los bloques no es la única forma de llevar a cabo acciones, con el fin de evitar un exceso de repetición de la escritura se implementaron las llamadas o invocaciones de procedimiento o función. Estas invocaciones nos permiten desarrollar una función tan solo con su nombre, evitándonos la tediosa tarea de escribirlo. Podemos emplear, con el fin de modificar estos procedimientos podemos emplear diversos valores denominados parámetros.

La creación de los procedimientos necesarios nos permite reutilizar código y por tanto ahorrar tiempo y evitar errores. También podemos crear procedimientos especiales llamados *Triggers*, que cuentan con la particularidad de invocarse a sí mismos cuando ocurre un evento específico a modo de respuesta.

La incorporación de invocaciones y *Triggers*, permite un código mucho más ligero y ordenado, evitando la repetición excesiva de un mismo código. Como segunda ventaja la reutilización implica un código más corto y, por tanto, menos errores y una codificación y depuración más sencilla. Emplearemos PL/SQL.

3.10.1. Procedimientos

El procedimiento permite a que un bloque de código pueda ser llamado con una invocación, su sintaxis se caracteriza por el inicio con la siguiente frase.

```
CREATE (OR REPLACE) PROCEDURE Nombre del proceso
Parámetros (con IN | OUT | IN OUT)
IS | AS
Declaraciones
BEGIN
...
```

Imagen 3. Ejemplo de sintaxis de un procedimiento

Esta sintaxis permite la creación del procedimiento, el inicio en `CREATE PROCEDURE` o `CREATE OR REPLACE PROCEDURE` es indispensable, ya que iniciará la creación o la creación y sustitución de un procedimiento. Tras esto se debe incluir el nombre de dicho procedimiento, sin ningún tipo de palabra clave o variable para que sea válido. Tras esto se añadirán los parámetros entre paréntesis y una de las siguientes palabras clave, `AS` o `IS`.

Tras esta introducción se añadirá el bloque que se desea que se produzca normalmente.

```
CREATE OR REPLACE PROCEDURE EJEMPLO AS
BEGIN
FOR NUM IN 0 .. 100 LOOP
DBMS_OUTPUT.PUT_LINE(NUM)
END LOOP;
END;
```

Imagen 4. Ejemplo de creación de un procedimiento

Parámetros en procedimientos

Los parámetros nos permitirán modificar el procedimiento al llamarlo, lo cual nos dará versatilidad sin tener que escribir el código de nuevo.

Todos los parámetros deben precederse, para su identificación de `IN`, `OUT`, `IN OUT`. Tras estos también se deben incluir los tipos de datos empleados.

Para poder entender las explicaciones de estos primero debemos conocer los conceptos de parámetros formales y actuales:

- **Parámetro formal:** parámetro definido por la función como una variable local de este, pero se inician con el valor actual que posean.
- **Parámetro actual:** valores de los parámetros al llamar la función. Se asignan a los parámetros formales en la invocación.

La diferencia entre `IN`, `OUT`, `IN OUT` es la siguiente.

- `IN=` Un parámetro de entrada, no modifica el resto de los parámetros, ni a sí mismo fuera del parámetro.
- `OUT=` Un parámetro de salida, el valor asignado afecta al parámetro, pero no emplea un procedimiento formal para introducir un valor.
- `IN OUT=` Un parámetro de entrada y salida, de modo que su valor inicial no es igual que su valor de salida, modificando este, al iniciar P1:=G1, al terminar el procedimiento G1:=P1.

Uno de los procedimientos comúnmente empleados es el de intercambio, permitiendo intercambiar dos valores entre sí. Con el fin de llevar a cabo este proceso podemos emplear variables no existentes como auxiliares, en este caso se denominarán AX. INT designa el tipo de dato, en este caso número entero.

```
CREATE OR REPLACE PROCEDURE INTERCAMBIO
(VALORA IN OUT INT, VALORB IN OUT INT) AS
AX INT;
BEGIN
      AX:=VALORA
      VALORA:=VALORB
      VALORB:=AX

END;
/

DECLARE
NUM1 INT:= 1
NUM2 INT:=0
BEGIN
      DBMS_OUTPUT.PUT_LINE('NUM1: ' || NUM1 || 'NUM2: ' ||NUM2);
            INTERCAMBIO (NUM1,NUM2);
      DBMS_OUTPUT.PUT_LINE('NUM1: ' || NUM1 || 'NUM2: ' ||NUM2);
END;
/
```

Imagen 5. Ejemplo de creación de procedimiento en invocación

3.10.2. Funciones

A diferencia de los procedimientos, las funciones tienen como final devolver un resultado concreto tras su invocación. Es imprescindible indicar el tipo de dato de la devolución. Su sintaxis es la siguiente:

```
CREATE OR REPLACE FUNCTION Nombre
RETURN Tipos de Dato
IS || AS
Declaraciones
BEGIN
…
END;
/
```

Imagen 6. Ejemplo de sintaxis de una función

Las líneas de creación y `RETURN` son imprescindibles.

Una función de permite introducir parámetros y obtener los resultados deseados.

```
CREATE OR REPLACE FUNCTION
EJEMPLO (PRE IN NUMBER, DIV IN NUMBER)
RETURN NUMBER AS
AX NUMBER:=0;
RES NUMBER:=0;
BEGIN
PRE/100:=AX;
AX*DIV:=RES;
RETURN(RES);
END;
/
```

Imagen 7. Ejemplo de una función para cálculo porcentajes

En este ejemplo al introducir los valores PRE (precio) y DIV (divisor) podremos obtener el cálculo de un porcentaje obteniendo como resultado el porcentaje DIV de PRE, por ejemplo, el 20% de 50 obtendríamos el resultado 10, lo cual obtendríamos al añadir los valores 50 y 20, en ese orden.

Llamadas a funciones y procedimientos

Dentro de un bloque anónimo, tanto para un procedimiento como para una función, solo es necesario escribir el nombre se estos y sus parámetros.

Fuera de un bloque podemos llamar un procedimiento mediante CALL o EXEC y una función en una expresión CALL DBMS_OUTPUT.PUT_LINE. Ejemplos:

```
BEGIN
EJEMPLO (100,20);
END;
/
```

```
CALL INTERCAMBIO (100,20);
EXEC INTERCAMBIO (100,20);
```

```
CALL DBMS_OUTPUT.PUT_LINE(EJEMPLO (100,20));
```

Imagen 8. Ejemplos de los tipos de llamadas

JDBC por su parte, para realizar las llamadas, emplea `CallableStatement`.

	Sintaxis de llamada	
	Con parámetros	Sin parámetros
Procedimiento	`{call Pro(?,?,…)}`	`{call Pro}`
Función	`{?=call Fun(?,?,…)}`	`{call Fun}`

Los métodos de CallableStatement para llamadas y resultados son los siguientes.

Métodos de CallableStatement para procedimientos y funciones	
Método	Funcionalidad
`setXX(int pos, Xx valor)`	Como PreparedStatement, asignan valores a parámetros predeterminados.
`Void registerOutParameter(int pos, int sqlType)`	Registra un parámetro de salida y el valor que devuelve.
`getXX(int pos)`	Obtiene el parámetro de salida.
`ResultSet getResultSet()`	Obtiene el resultado con ResultSet.

3.11. Actualizaciones sobre los resultados de una consulta

ResultSet, aunque destinada a consulta también es capaz de realizar cambios en la base de datos como insertar, actualizar o eliminar filas de una tabla. Esto se realiza mediante ResultSet actualizables o empleando, más comúnmente, sentencias UPDATE,DELETE e INSERT o clausulas WHERE con UPDATE y DELETE.

ResultSet posee un parámetro tipo *scrollable* con dos posibles valores:

- `ResultSet.CONCUR_READ_ONLY`: no es actualizable por lo que los cambios en él no pasan a la base de datos.
- `ResultSet.CONCUR.UPDATABLE`: es actualizable por lo que es lo posible gravar en la base de datos los cambios realizados.

Métodos con ResultSet actualizables para la actualización

Método	Funcionalidad
`void updateXX(int Pos, Xx valor)` `void updateXX(String Nombre, Xx valor)`	Asigna un valor a la columna indicada.
`void updateNull(int pos)` `void updateNull(String Nombre)`	Indicando una columna y le asigna el valor NULL.
`void updateRow()`	Copia el valor actual de `ResultSet` en la base de datos.
`deleteRow()`	Borra el valor actual de `ResultSet` en la base de datos.
`moveToInsertRow()`	Mueve a la fila de inserción el cursor. Introducimos el valor con `updateXX()` y lo insertamos con `insertRow()`. Finalmente salimos con `moveToCurrentRow()`
`Void insertRow()`	Inserta en ResultSet y la base de datos el valor de la fila de inserción

3.12. Ejecución de scripts

Una secuencia de sentencias de SQL separadas por ";", es un *script* de SQL y para poder ejecutarlo en MySQL es necesario establecerse una conexión con la opción `allowMultiQueries=true` en la URL. De esta forma, como una sentencia SQL individual, podemos ejecutar un *script* mediante un Statement.

Este tipo de *script* de SQL se emplean para crear y actualizar la base de datos o para la instalación de aplicaciones. Empleando un conjunto de estos, normalmente en un único fichero, se crean la base de datos y su estructura.

3.13. Ejecución de sentencias por lotes

Para evitar problemas al ejecutar un gran número de sentencias de SQL, es posible emplear lotes, conjuntos de sentencias SQL que se ejecutan juntas.

Los lotes se crean empleando el método `addBatch(String sql)` de `Statement y addBatch()` y `preparedStatement`. Statement permites lotes con sentencias del mismo tipo, mientras que con `PreparedStatement` las sentencias deben de ser del mismo tipo. Los lotes se ejecutan con `executeBatch()`, devuelve el número de filas afectadas mediante `int()`. También existen los métodos `executeLargeBatch()` que devuelven un *array* `long[]` y `CallableStatement`.

www.universae.com