Información de la BD y optimización en consultas

Índice

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.getInt("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");
ResultSetMetaData rsmd = resul.getMetaData();
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.

2. Optimización de sentencias

Cuando ejecutamos una sentencia SQL, esta se compila y se manda al SGBD. Si la vamos a invocar 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 alumnos
    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 alumnos
    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. Llamadas a procedimientos

Los procedimientos (PROCEDURES) (también llamados funciones en algunos SGBD) 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.

A continuación se muestra un ejemplo de función en PostGres:

```
create or replace function devhoralleg(int)
    returns time language 'plpgsql' as '
    declare aux vuelo%rowtype;
    begin
        select * into aux from vuelo where numero=($1);
        return aux.horalleg;
    end;'
```

Esta función recibe como parámetro un entero indicando en número de vuelo y devuelve la hora de llegada del vuelo, en formato Time. Antes de crear esta función, debéis ejecutar los siguientes comandos dentro de PostGres, para definir el lenguaje PlPgsql:

```
create function plpgsql_call_handler ()
    returns language_handler as '$libdir/plpgsql' language C;
create trusted procedural language plpgsql
handler plpgsql_call_handler;
```

Para hacer uso de esta función, PostGres no utiliza el procedimiento detallado en esta sección, sino que se basa en la llamada a una sentencia *select* tal como se muestra:

```
ResultSet rs=stmt.executeQuery("select devhoralleg(1)");
if (rs.next()) System.out.println("hola "+rs.getTime(1));
```