Tema 4

Manejo de conectores

4.1. Introducción.

BdD relacionales: las más extendidas.

BdOO usan estructuras mucho más cercanas a la POO sin embargo no han tenido tanto éxito:

- No representan bien relaciones.
- No existe un lenguaje con la potencia y versatilidad de SQL para tratar los datos.

Todo ello lleva a que su uso sea de "nicho".

4.1.1. El desfase objeto-relacional.

Problemas debidos a las diferencias entre el modelo de datos de la BdD relacional y el lenguaje de POO.

Las BdD relacionales no fueron pensadas para almacenar objetos -> mayor esfuerzo de programación.

Mejorado por el mapeo entre estructuras que veremos más adelante.

4.2. Protocolos de acceso a BdD.

Problema histórico: cada SGBD usaba su propia API.

Dos soluciones generales principales:

- **ODBC** (Open Database Connectivity): API de Microsoft creada para tener un estándar de acceso a BdD en Windows. Su complejidad ha hecho que casi no haya salido del ecosistema Windows.
- **JDBC** (Java Database Connectivity): API que pueden usar los programas Java para conectarse a SGBD relacionales. Sencillo pero permite mucha flexibilidad a los desarrolladores.

4.2.1. Arquitectura JDBC.

Soporta dos modelos de procesamiento:

- Dos capas: (Figura 4.1)
  - La aplicación se comunica directamente con la fuente de datos.
  - Se necesita un conector JDBC que pueda comunicar con la fuente de datos específica.
  - Los comandos o instrucciones del usuario se envían a la base de datos y los resultados se devuelven a la aplicación.

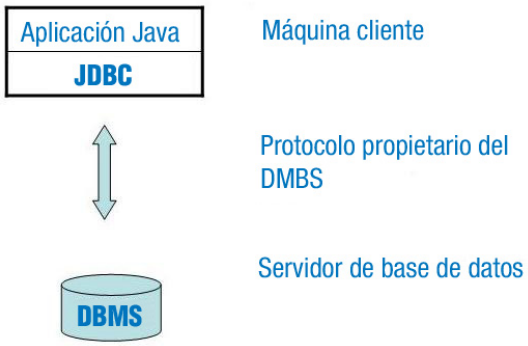

Figura 4.1: Modelo de dos capas. Imagen obtenida del curso Acceso a datos del MEFP.

- Tres capas: (Figura 4.2)
  - La aplicación se comunica con una capa intermedia (servidor de lógica de negocio) mediante algún protocolo generalmente estándar (por ejemplo HTTP).
  - La capa intermedia envía las órdenes necesarias a la fuente de datos.
  - La fuente de datos procesa los comandos y envía los resultados de vuelta la capa intermedia, desde la que luego se le envían a la aplicación.

El API JDBC viene distribuido en dos paquetes:

- `java.sql`

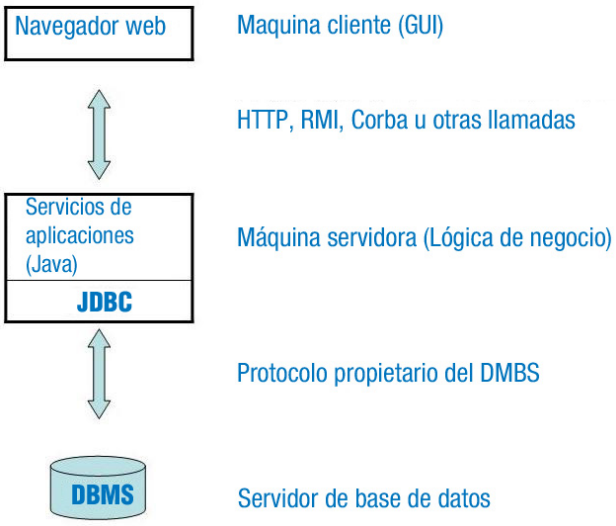

Figura 4.2: Modelo de tres capas. Imagen obtenida del curso Acceso a datos del MEFP.

- `javax.sql`

4.2.2. Conectores.

Conjunto de clases que implementan las interfaces de la API de acceso a BdD.

En Java suele ser un fichero `.jar`.

Al usar JBDC nuestro código no dependerá del conector (como nos ha pasado en los temas anteriores) ya que todos implementan las interfaces definidas. (Figura 4.3).

La parte de JDBC se puede dividir en tres: (Figura 4.4).

- La API JDBC.
- El gestor de drivers.
- La capa de drivers específicos de cada origen de datos.

4.3. Bases de datos.

Existen infinidad de opciones. Para nuestro propósito destaco dos paradigmas.

4.3.1. SGBD clásicos.

Funcionan en paradigma cliente/servidor.

Ejemplos:

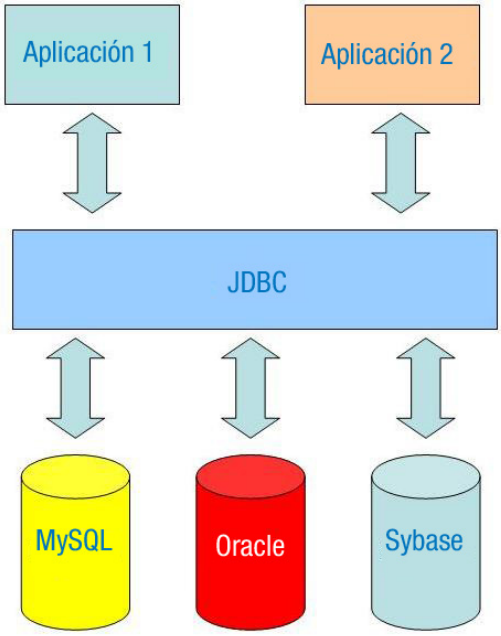

Figura 4.3: Independencia del conector con JDBC. Imagen obtenida del curso Acceso a datos del MEFP.

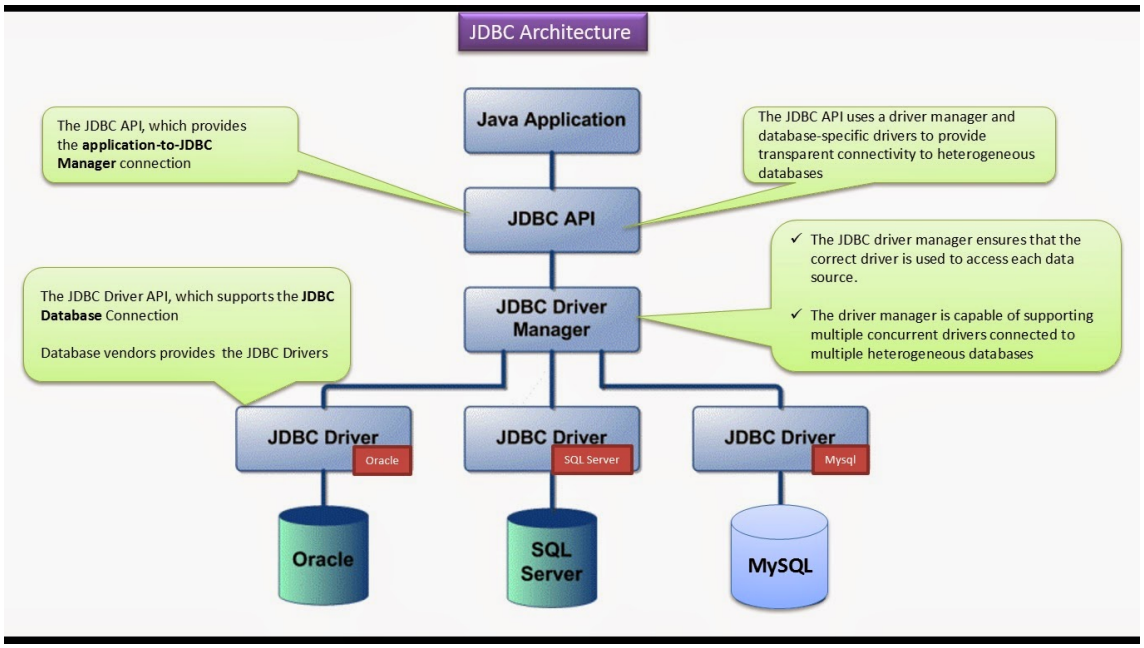

Figura 4.4: Estructura de JDBC. Imagen obtenida aquí.

- Oracle Database.
- MySQL.
- MariaDB.
- SQL Server.

4.3.2. BdD embebidas.

Para datos locales de aplicaciones móviles no necesitamos guardar mucha información, permitir conexiones concurrentes, etc.

BdD embebida: el motor está incrustado en la aplicación y es de acceso exclusivo desde ella.

BdD arranca cuando inicia la aplicación y para cuando esta se cierra.

Ejemplos:

- SQLite.
- Apache Derby: muy fácil integrarla en cualquier aplicación Java. Soporta también el paradigma Cliente/Servidor.
- HSQLDB (Hyperthreated Structured Query Language Database).
- H2.
- Firebird.
- SQL Server Compact.

4.4. Operando con JDBC.

JDBC consta de un conjunto de clases e interfaces que nos permite escribir el código Java en nuestras aplicaciones para realizar las siguientes operaciones:

- Abrir conexiones a la BdD.
- Enviar sentencias SQL a la BdD para que sean ejecutadas.
- Recuperar y procesar los resultados recibidos de la base de datos como respuesta a las sentencias.

Las clases e interfaces principales de JDBC son:

- `DriverManager`
- `Connection`
- `Statement`
- `Resultset`

El procedimiento habitual para acceder a una BdD en nuestra aplicación es: (Figura 4.5).

1. Importar las clases necesarias.
2. Cargar el *driver* JDBC.
3. Identificar el origen de datos.
4. Crear un objeto `Connection`.
5. Crear un objeto `Statement`.
6. Ejecutar una consulta con el objeto `Statement`.
7. Recuperar lo datos del objeto `ResultSet`.
8. Liberar el objeto `ResultSet`.
9. Liberar el objeto `Statement`.
10. Liberar el objeto `Connection`.

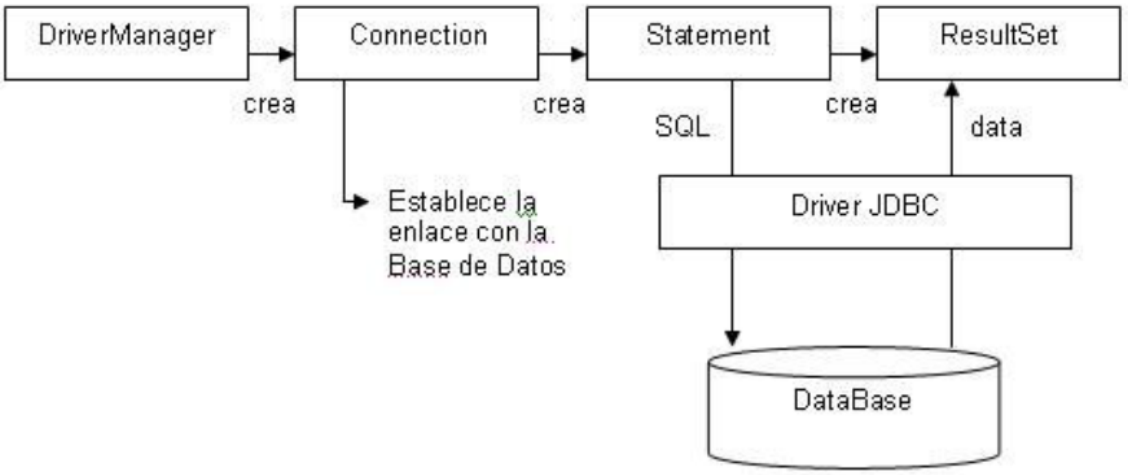

Figura 4.5: Procedimiento habitual con JDBC. Imagen obtenida del material de Carlos Tessier.

Una lista un poco más extensa de clases e interfaces útiles incluiría:

- `Driver`: permite conectarse a una base de datos. cada gestor de base de datos requiere un driver distinto.
- `DriverManager`: permite gestionar todos los drivers instalados en el sistema.
- `DriverPropertyInfo`: proporciona diversa información acerca de un driver.
- `Connection`: representa una conexión con una base de datos. Una aplicación puede abrir más de una conexión.
- `DatabaseMetadata`: proporciona información acerca de una BD, tablas que contiene, etc.
- `Statement`: permite ejecutar sentencias SQL sin parámetros.
- `PreparedStatement`: permite ejecutar sentencias SQL con parámetros de entrada.

- `CallableStatement`: permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados.
- `ResultSet`: contiene las filas resultantes de ejecutar una orden `SELECT`.
- `ResultSetMetadata`: permite obtener información sobre un ResultSet, como el número de columnas, sus nombres, etc.

4.5. Conexiones.

Dos aspectos que cambian al trabajar con JDBC según la fuente de datos:

- Cómo incluir el *driver*:
  - Descargar e importar en nuestro proyecto.
  - Incluirla con Maven.
  - Incluirla con Gradle.
- Cómo cargar el *driver*.
- Cómo pedirle al `DriverManager` que cree una conexión. Siempre será con:

  ```
  conexion = DriverManager.getConnection(url);
  ```

  o con

  ```
  conexion = DriverManager.getConnection(url, username, password);
  ```

  pero cambiará el formato de la url.

A modo de ejemplo vamos a ver algunos.

NOTA: la conexión debe cerrarse.

```
conexion.close();
```

4.5.1. SQLite.

Descarga.

Dependencia Maven:

```xml
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.40.0.0</version>
</dependency>
```

Cadena de conexión:

```
String url = "jdbc:sqlite:test.db";
```

donde `test.db` es la base de datos.

Aquí puedes ver cómo incorporar SQLite a tu proyecto de IntelliJ.

4.5.2. Apache Derby.

Descarga.

Dependencia Maven:

```xml
<dependency>
    <groupId>org.apache.derby</groupId>
    <artifactId>derby</artifactId>
    <version>10.16.1.1</version>
</dependency>
```

Cargar el *driver*.

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

Cadena de conexión:

```
String url = "jdbc:derby://localhost:1527/test";
String username = "user";
String password = "password";
```

donde `test` es la base de datos.

4.5.3. MySQL.

Descarga (JDBC Driver).

Dependencia Maven:

```
<dependency>
   <groupId>com.mysql</groupId>
   <artifactId>mysql-connector-j</artifactId>
   <version>8.0.31</version>
</dependency>
```

Cargar el *driver*.

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Cadena de conexión:

```
   String url = "jdbc:mysql://hostname:port/test";
   String username = "user";
   String password = "password";
```

donde:

- `test` es la base de datos.
- `hostname` es el nombre DNS o dirección IP del SGBD.
- `port` es el puerto en el que escucha la BdD. Por defecto 3306.

4.6. Ejecución de sentencias.

La interfaz `Statement` permite utilizar métodos para ejecutar sentencias SQL y obtener los resultados:

```
Statement sentencia = conexion.createStatement();
```

NOTA: los `Statements` también se cierran.

Tiene varios métodos útiles:

- **`executeQuery(String)`**: para sentencias `SELECT` que devolverán datos en un único objeto `ResultSet`.
- **`executeUpdate(String)`**: para sentencias que como `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `DROP` y `ALTER` que devolverán el número de filas afectadas.
- **`execute(String)`**: para procedimientos almacenados.

4.7. Sentencias SELECT.

Un `ResultSet`:

- Cargará las filas que devuelva la ejecución de la sentencia.
- Permite acceder a los valores de cualquier columna de la fila actual.
- Podemos obtener el dato según el tipo con los métodos:
  - `getString(columna)`
  - `getBoolean(columna)`
  - `getInt(columna)`
  - `getLong(columna)`
  - `getFloat(columna)`
  - `getDouble(columna)`
  - `getDate(columna)`
  - `getTime(columna)`

Y puede recorrerse con `next()`:

```java
public static void consultarEmpleados() {
   // uso try-with-resources anidados.
   try(Connection conexion =
         DriverManager.getConnection(urlBaseDeDatos, usr, pwd)) {
      /* no añado un catch porque es la misma excepción por
         un motivo similar.*/
      try(Statement sentencia = conexion.createStatement()) {
         // Creamos la consulta SQL.
         String consulta =
            "SELECT nombre, apellido, fecha_contratacion, salario
               FROM empleados;";
         // La ejecutamos.
         try(ResultSet resultado = sentencia.executeQuery(consulta)) {
            // Recorremos los resultados.
            while (resultado.next()) {
               // Cargamos los distintos tipos de datos.
               String nombre = resultado.getString("nombre");
               String apellido = resultado.getString("apellido");
               // Conversión sencilla entre los tipos de fechas.
               Date fechaCon = resultado.getDate("fecha_contratacion");
               Calendar fechaContratacion =  new GregorianCalendar();
               fechaContratacion.setTime(fechaCon);
               float salario = resultado.getFloat("salario");

               // lo muestro por pantalla pero podría construir objetos.
               mostrarEmpleado(nombre, apellido, fechaContratacion,
                  salario);
            }
         } catch (SQLException ex) {
            System.out.println("ERROR: no se pudo ejecutar la
               consulta solicitada.");
         }
      }
   } catch (SQLException ex) {
      System.out.println("ERROR: no se pudo acceder a la base de
         datos Empleados.");
   }
}
```


4.8. Errores producidos en el gestor de bases de datos.

`SQLException` nos proporciona información de errores que se producen en el SGBD.

Principalmente usaremos tres métodos:

- `getMessage()`: mensaje del error producido.
- `getSQLState()`: código de estado definido por el estándar X/OPEN SQL.
- `getErrorCode()`: código de error dependiente del SGBD.

Si usamos en un `catch`:

```
System.out.println("Mensaje: " + ex.getMessage());
System.out.println("Estado SQL: " + ex.getSQLState());
System.out.println("Código de error: " + ex.getErrorCode());
```

Podríamos obtener por ejemplo:

```
Mensaje: Table 'empleados.empleadoRS' doesn't exist
Estado SQL: 42S02
Código de error: 1146
```

Ejercicios 4.1

1. Sobre la base de datos de "Empleados" (Apéndice E: Base de datos: empleados.) debes crear una clase que permita guardar los datos de un empleado de la tabla "Empleados", sin incluir el "id_director" y el "id_departamento". Debe llevar un método `toString()` apropiado.

   Lee todos los empleados del departamento "Ventas", cárgalos en objetos de la clase que has creado y muestra sus datos por pantalla.
2. A la clase anterior añádele lo necesario para incluir el nombre del departamento en el que trabaja cada empleado. Carga en un `List` los empleados de los departamentos "Ejecutivo" y "Marketing" y posteriormente muestra sus datos por pantalla.
3. Ahora queremos poder añadir a la clase los datos del director de cada empleado. Observa que es una relación reflexiva ("id_director" apunta a la tabla empleados). ¿Qué tipo de dato debes usar para guardar los datos del director? Aquí puedes observar cómo en BdD relacionales usamos claves ajenas mientras que en POO se utilizan referencias a objetos.

Modifica el método `toString()` para que en caso de que tenga director, muestre sus datos entre una etiquetas "——Datos director——" y "——Fin DD——". Ten en cuenta que el director también es un empleado por lo que puedes llamar a su propio método `toString()`.

Pruébalo, en dos ejecuciones distintas de tu programa, con los empleados del departamento "Ejecutivo" y luego con los del departamento "Administración".

4.9. Sentencias preparadas o parametrizadas.

Las sentencias preparadas[1], nos permiten modificar parámetros en la misma consulta.

Interfaz `PreparedStatement`.

Se usan *placeholders* que se marcan con un signo de cierre de interrogación (?).

Cada *placeholder* se identifica mediante un índice.

Los *placeholders* no se pueden usar para determinar nombres de columnas o tablas.

Mejor que concatenar *Strings*: se precompilan una vez y se pueden reutilizar cambiando los parámetros.

Los métodos son equivalentes a los de `Statement`.

NOTA: una diferencia importante es que al hacer `conexion.prepareStatement(consulta)` se asigna por lo que el método de ejecutar la consulta (depende del tipo, pero en nuestro caso `sentencia.executeQuery()`) no recibe la consulta como parámetro.

[1] Prepared Statements


```java
public static void consultarEmpleados() {
   try(Connection conexion =
         DriverManager.getConnection(URL_BASE_DATOS, USR, PWD)) {

      // Consulta con un único placeholder.
      String consulta = "SELECT nombre, apellido, salario "
         + "FROM empleados WHERE id_departamento = ?";

      try(PreparedStatement sentencia =
            conexion.prepareStatement(consulta)) {
         // asignamos parámetros a los placeholders.
         sentencia.setInt(1, 10);

         // La ejecutamos SIN PASAR LA CONSULTA COMO PARÁMETRO.
         try(ResultSet resultado = sentencia.executeQuery()) {
            while (resultado.next()) {
               String nombre = resultado.getString("nombre");
               String apellido = resultado.getString("apellido");
               float salario = resultado.getFloat("salario");

               mostrarEmpleado(nombre, apellido, salario);
            }
         } catch (SQLException ex) {
            System.out.println("ERROR: no se pudo ejecutar la "
               + "consulta solicitada.");
         }
      }
   } catch (SQLException ex) {
      System.out.println("ERROR: no se pudo acceder a la base de "
         + "datos Empleados.");
   }
}
```

En el siguiente ejemplo vemos cómo se puede reutilizar la sentencia solo cambiando los parámetros:


```java
public static void consultarEmpleadosVariosDepartamentos() {
   int[] departamentos = {10, 20, 30};

   try(Connection conexion =
      DriverManager.getConnection(URL_BASE_DATOS, USR, PWD)) {

      String consulta = "SELECT nombre, apellido, salario "
         + "FROM empleados WHERE id_departamento = ?";

      try(PreparedStatement sentencia =
            conexion.prepareStatement(consulta)) {
         for(int departamento: departamentos) {
            // asignamos parámetros a los placeholders.
            sentencia.setInt(1, departamento);

            try(ResultSet resultado = sentencia.executeQuery()) {
               System.out.println("\n***Departamento: "
                  +departamento+"***\n");
               while (resultado.next()) {
                  // Cargamos los distintos tipos de datos.
                  String nombre = resultado.getString("nombre");
                  String apellido = resultado.getString("apellido");
                  float salario = resultado.getFloat("salario");

                  mostrarEmpleado(nombre, apellido, salario);
               }
            } catch (SQLException ex) {
               System.out.println("ERROR: no se pudo ejecutar la "
                  + "consulta solicitada.");
            }
         }
      }
   } catch (SQLException ex) {
      System.out.println("ERROR: no se pudo acceder a la base de "
         + "datos Empleados.");
   }
}
```


4.10. Sentencias que modifiquen la información.

Podemos usar `INSERT`, `UPDATE` y `DELETE`.

Utilizamos `executeUpdate(String)` (o `executeUpdate()` con sentencias preparadas).

Devuelve un entero con el número de filas afectadas.

```
public static void modificarEmpleado() {
   try(Connection conexion =
   DriverManager.getConnection(URL_BASE_DATOS, USR, PWD)) {

      // Consulta con varios placeholders.
      String actualizacion = "UPDATE empleados " +
      "SET apellido=?, email=?, telefono=? " +
      "WHERE id_empleado=?;";

      try(PreparedStatement sentencia =
      conexion.prepareStatement(actualizacion)) {
         // asignamos parámetros a los placeholders.
         sentencia.setString(1, "Márquez");
         sentencia.setString(2, "am@scv.test");
         sentencia.setString(3, "612646765");
         sentencia.setInt(4, 601);

         int resultado = sentencia.executeUpdate();

         if(resultado≠1) {
            // aquí deberías introducir una gestión de errores adecuada.
            System.out.println("No se ha podido actualizar.");
         }
         else {
            System.out.println("Actualizado con éxito.");
         }
      }
   } catch (SQLException ex) {
      System.out.println("ERROR: no se pudo acceder a la base de "
      + "datos Empleados.");
   }
}
```


Ejercicios 4.2

1. Refactoriza el ejercicio 4.1.3 para que se haga con sentencias preparadas. . La consulta debe mostrar solo los empleados de un departamento determinado. El departamento se filtrará por nombre, no por identificador.
2. Crea un método para insertar un nuevo empleado a partir del código del ejercicio 4.2.1. Tendrás un objeto de la clase que creaste y ese es el que debes insertar. Ten en cuenta que necesitarás datos como el id del director, etc.
3. Crea un método que permita borrar un empleado dado su identificador. Ten en cuenta que puede haber condiciones que impidan que se elimine como por ejemplo que sea director de otro empleado.

4.11. Opciones sobre un `ResultSet`.

Hemos creado `Statements` que nos iban a devolver `ResultSets` básicos.

Podemos ver que hay tres opciones:

- createStatement()
- createStatement(int resultSetType, int resultSetConcurrency)
- createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)

Lo que nos muestra que hay tres modificadores que podemos indicar:

- Tipo:
  - `ResultSet.TYPE_FORWARD_ONLY`: solo podrá recorrerse hacia adelante.
  - `ResultSet.TYPE_SCROLL_INSENSITIVE`: el cursor puede moverse hacia adelante, hacia atrás o situarse en una posición concreta. **NO** refleja los cambios producidos en los datos en la fuente (BdD) desde que se cargaron en el RS.
  - `ResultSet.TYPE_SCROLL_SENSITIVE`: el cursor puede moverse hacia adelante, hacia atrás o situarse en una posición concreta. **SÍ** refleja los cambios producidos en los datos en la fuente (BdD) desde que se cargaron en el RS.
- Concurrencia:
  - `ResultSet.CONCUR_READ_ONLY`: los datos en el RS **no** se pueden modificar.
  - `ResultSet.CONCUR_UPDATABLE`: los datos en el RS **sí** se pueden modificar.
- Comportamiento ante `COMMIT`:

- `ResultSet.HOLD_CURSORS_OVER_COMMIT`: los objetos que estén abiertos en este RS se **mantendrán abiertos** tras un `COMMIT`.
- `ResultSet.CLOSE_CURSORS_AT_COMMIT`: los objetos que estén abiertos en este RS se **cerrarán** tras un `COMMIT`.

4.11.1. Navegar por el RS.

Según lo indicado al crear el `Statement` podremos movernos por el RS con los siguientes métodos:

- `next()`: mueve el cursor una fila hacia delante. Devuelve `true` si se posiciona en una fila y `false` si después de la última fila. (El único que se puede utilizar en cualquier tipo de RS).
- `previous()`: mueve el cursor una fila hacia atrás. Devuelve `true` si se posiciona en una fila y `false` antes de la última fila.
- `first()`: mueve el cursor a la primera fila. Devuelve `true` si se posiciona en la primera fila y `false` si no hay filas en el `ResultSet`.
- `last()`: mueve el cursor a la última fila. Devuelve `true` si se posiciona en la última fila y `false` si no hay filas en el `ResultSet`.
- `beforeFirst()`: mueve el cursor antes de la primera fila. Sin efecto si el `ResultSet` no tiene filas.
- `afterLast()`: mueve el cursor después de la última fila. Sin efecto si el `ResultSet` no tiene filas.
- `relative(int rows)`: mueve el cursor relativa a su posición actual.
- `absolute(int row)`: posiciona el cursor en la fila especificada de forma absoluta.

Por ejemplo:


```
try(Connection conexion =
      DriverManager.getConnection(URL_BASE_DATOS, USR, PWD)) {
   // Creamos la sentencia con las opciones indicadas.
   try(Statement sentencia = conexion.createStatement(
         ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY)) {
      // Creamos la consulta SQL.
      String consulta =
         "SELECT nombre, apellido, fecha_contratacion, salario "
         + "FROM empleados;";
      // La ejecutamos.
      try(ResultSet resultado = sentencia.executeQuery(consulta)) {
         // Nos situamos al final.
         resultado.afterLast();
         // Recorremos los resultados al revés.
         while (resultado.previous()) {
            String nombre = resultado.getString("nombre");
            String apellido = resultado.getString("apellido");
            // Conversión sencilla entre los tipos de fechas.
            Date fechaCon = resultado.getDate("fecha_contratacion");
            Calendar fechaContratacion =  new GregorianCalendar();
            fechaContratacion.setTime(fechaCon);
            float salario = resultado.getFloat("salario");

            mostrarEmpleado(nombre, apellido, fechaContratacion,
               salario);
         }
      } catch (SQLException ex) {
         System.out.println("ERR: no se pudo ejecutar la consulta.");
      }
   }
} catch (SQLException ex) {
   System.out.println("ERR: no se pudo acceder a la base de datos.");
}
```

4.11.2. Actualizar el RS.

Podemos modificar los datos de un RS.

Existen métodos *update* para los ocho tipos primitivos, para `String`, `Object`, `URL`, y los tipos de datos de SQL que hay en el paquete `java.sql`.

Para cada tipo existen dos versiones:

- Por nombre de la columna.
- Por posición (índice) de la columna.

Además el RS debe cumplir unas condiciones determinadas:

- Haber creado el `Statement` con `ResultSet.CONCUR_UPDATABLE`.
- La consulta solo puede implicar una única tabla.
- La consulta no puede usar funciones.
- La consulta debe incluir todas los atributos que forman parte de la clave primaria.

Por ejemplo:

- `updateString(String columnName, String s)`
- `updateString(int columnIndex, String s)`

Se puede actualizar el RS sin modificar los datos de la BdD.

Esto solo se va a ver en los RS de algunos que utilizan algunas implementaciones de JDBC.

No es el caso de MySQL como podemos ver al ejecutar el siguiente código:

```
try(Connection conexion =
      DriverManager.getConnection(URL_BASE_DATOS, USR, PWD)) {
   // Obtenemos el objeto para sacar la metainformación.
   DatabaseMetaData dbmd = conexion.getMetaData();

   System.out.println("Updates Visibles: " +
      dbmd.ownUpdatesAreVisible(ResultSet.TYPE_SCROLL_SENSITIVE));
   System.out.println("Inserts Visibles: " +
      dbmd.ownInsertsAreVisible(ResultSet.TYPE_SCROLL_SENSITIVE));
   System.out.println("Deletes Visibles: " +
      dbmd.ownDeletesAreVisible(ResultSet.TYPE_SCROLL_SENSITIVE));

} catch (SQLException ex) {
   System.out.println("ERROR: no se pudo acceder a la base de datos "
   + "Empleados.");
}
```

Que muestra:

```
Updates Visibles: false
Inserts Visibles: false
Deletes Visibles: false
```

De todas formas, muestro un ejemplo de cómo actualizar los datos:


```
float porcentajeSubida = (float)1.2;

try(Connection conexion =
      DriverManager.getConnection(URL_BASE_DATOS, USR, PWD)) {
   // no añado un catch por misma excepción por un motivo similar.
   try(Statement sentencia = conexion.createStatement(
         ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
      // Creamos la consulta SQL.
      String consulta =
         "SELECT id_empleado, nombre, apellido, fecha_contratacion, "
         + "salario FROM empleados;";
      // La ejecutamos.
      try(ResultSet resultado = sentencia.executeQuery(consulta)) {
         while (resultado.next()) {
            float salario = resultado.getFloat("salario");
            resultado.updateFloat("salario",
               salario * (float)porcentajeSubida);
         }

         resultado.beforeFirst();
         // Recorremos los resultados.
         while (resultado.next()) {
            String nombre = resultado.getString("nombre");
            String apellido = resultado.getString("apellido");
            // Conversión sencilla entre los tipos de fechas.
            Date fechaCon = resultado.getDate("fecha_contratacion");
            Calendar fechaContratacion =  new GregorianCalendar();
            fechaContratacion.setTime(fechaCon);
            float salario = resultado.getFloat("salario");

            mostrarEmpleado(nombre, apellido, fechaContratacion, salario);
         }
      } catch (SQLException ex) {
         System.out.println("ERR: no se pudo ejecutar la consulta.");
      }
   }
} catch (SQLException ex) {
   System.out.println("ERR: no se pudo acceder a la base de datos.");
}
```

4.11.3. Modificar el RS trasladando los cambios a la BdD.

Sin embargo, si se reflejan al trasladar los cambios a la BdD.

Ten en cuenta que creamos el `Statement` con opción de que reflejara los cambios sobre los datos en la BdD.

```
conexion.createStatement(
   ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)
```

Así que para reflejar los cambios, usaremos `updateRow()`.

Por lo que si modificamos el código anterior para que lo incluya:

```
    ...
while (resultado.next()) {
   float salario = resultado.getFloat("salario");
   resultado.updateFloat("salario",
   salario * (float)porcentajeSubida);
   resultado.updateRow();
}
    ...
```

4.11.4. Otras operaciones sobre los datos del RS.

Podemos también realizar inserciones y borrados:

- Borrado de una fila: `deleteRow()` -> elimina la fila en la que nos encontremos.
- Inserciones:
  1. Desplazar el cursor hasta el punto donde se puede insertar una nueva fila:

     ```
     rs.moveToInsertRow();
     ```

  2. Luego debes "crear" la fila en el RS: usando los métodos updateXX vistos, por ejemplo:

     ```
     rs.updateInt("id_empleado", "675");
     rs.updateString("nombre", "Pedro");
     ...
     ```


3. Ejecutar la inserción:

```
rs.insertRow();
```

Ejercicios 4.3

En algún ejercicio es posible que tengas que desactivar las actualizaciones y borrados seguros tal y como explico en el apéndice Base de datos: empleados.

1. Crea un programa que cargue todos los empleados en un `ResultSet`, ordenados por número de departamento.

   Posteriormente debes recorrer el RS para determinar donde empiezan y terminan los empleados del departamento 30.

   Una vez tengas las posiciones, muéstralos en orden inverso.
2. Crea un programa que cargue todos los empleados en un RS y modifique sus salarios: el de los pares lo multiplicará por 2 y el de los impares lo dividirá entre 2.

   Posteriormente debes volver a recorrer el RS para comprobar si los salarios se encuentran entre los valores indicados para cada trabajo. En caso contrario los actualizará:

   - Si no llega al mínimo, se le asignará el mínimo.
   - Si se pasa del máximo, se le asignará el máximo.
3. Crea un programa que cargue las localizaciones en un RS y los departamentos en otro.

   Debe borrar aquellas localizaciones en las que no haya ningún departamento.

   Si es necesario, puedes modificar los datos de la BdD para probar tu programa.

4.12. Transacciones.

Transacción: conjunto de operaciones que se realizan en conjunto o no se realizan.

Se lleva a cabo mediante `commit` y `rollback`:

- Todas las operaciones realizadas correctamente: `commit`.
- Si alguna operación falla: `rollback`.

IMPORTANTE:

Realmente todas las operaciones de bases de datos se realizan como transacciones pero los SGBD suelen tener un modo *autocommit* que hace que cada instrucción que enviamos se englobe en una transacción.
Para aplicaciones con trabajo intensivo con BdD, desactivar *autocommit* y controlar nosotros las transacciones, incluyendo varias operaciones cada vez, puede mejorar el rendimiento sensiblemente.

En java:

- Desactivar el *autocommit* (Por defecto cualquier sentencia SQL que ejecutemos tiene efecto automático en la BD) `conection.setAutoCommit(false);`
- Realizar un `commit` después del éxito de todas las operaciones de la transacción. `conection.commit();`
- Realizar un `rollback` (volver al estado inicial) si se produce algún fallo. `conection.rollback();`

Por ejemplo:


```
try(Connection conexion =
      DriverManager.getConnection(URL_BASE_DATOS, USR, PWD)) {

   conexion.setAutoCommit(false);

   String insercion = "INSERT INTO departamentos (id_departamento, "
      + "nombre_departamento, id_director, id_localizacion) "
      + "VALUES (?, ?, NULL, 1000);";

   try(PreparedStatement sentencia
         = conexion.prepareStatement(insercion)) {
      try {
         // Preparamos la sentencia y la ejecutamos. Operación 1
         sentencia.setInt(1, 100);
         sentencia.setString(2, "Venta de helados");
         sentencia.executeUpdate();

         // Preparamos la sentencia y la ejecutamos. Operación 2
         sentencia.setInt(1, 101);
         sentencia.setString(2, "Magia chachi");
         sentencia.executeUpdate();

         conexion.commit();
      } catch (SQLException ex) {
         System.out.println("Error al procesar la transacción.");
         System.out.println("Revirtiendo al estado anterior.");
         // Volvemos al estado anterior.
         conexion.rollback();
      }
   }  finally {
      // dejamos el estado anterior.
      conexion.setAutoCommit(true);
   }
} catch (SQLException ex) {
   System.out.println("ERROR: no se pudo acceder a la BdD.");
}
```

IMPORTANTE:

Devolver *autocommit* a `true` depende de la aplicación. Si vamos a hacer un uso intensivo de las transacciones no tendríamos por qué hacerlo pero si el uso de transacciones es esporádico, sería preferible hacerlo al terminar de ejecutar una tal y cómo se hace en el ejemplo.

Sin embargo si intentamos insertar una PK duplicada y mostramos los detalles (solo muestro la parte que cambia):

```
try {
   // Preparamos la sentencia y la ejecutamos. Operación 1
   sentencia.setInt(1, 102);
   sentencia.setString(2, "Montaje de arcoiris");
   sentencia.executeUpdate();

   // Intentar insertar un id repetido fallará.
   sentencia.setInt(1, 102);
   sentencia.setString(2, "Fin de fiesta");
   sentencia.executeUpdate();

   conexion.commit();
} catch (SQLException ex) {
   // Volvemos al estado anterior.
   conexion.rollback();
   System.out.println("Error al procesar la transacción.");
   System.out.println("Revirtiendo al estado anterior.");
   /* Información para el desarrollaror */
   System.out.println("Mensaje: " + ex.getMessage());
   System.out.println("Estado SQL: " + ex.getSQLState());
   System.out.println("Código de error: " + ex.getErrorCode());
}
```

Obtendríamos el siguiente resultado y podríamos comprobar que no se ha actualizado tampoco la primera de las inserciones que no produce ningún error:


```
Error al procesar la transacción.
Revirtiendo al estado anterior.
Mensaje: Duplicate entry '102' for key 'departamentos.PRIMARY'
Estado SQL: 23000
Código de error: 1062
```

4.12.1. Puntos intermedios (*Savepoints.*)

Se pueden establecer puntos intermedios y hacer *rollbacks* a ellos (solo se muestra la parte que cambia):

```java
try {
   Savepoint spInicial = conexion.setSavepoint();
   // Preparamos la sentencia y la ejecutamos. Operación 1
   sentencia.setInt(1, 200);
   sentencia.setString(2, "Nubes de algodón dulce");
   sentencia.executeUpdate();

   // Preparamos la sentencia y la ejecutamos. Operación 2
   int id2 = 201;
   sentencia.setInt(1, id2);
   sentencia.setString(2, "Unicornios rosas");
   sentencia.executeUpdate();

   Savepoint spIntermedio = conexion.setSavepoint();
   // Preparamos la sentencia y la ejecutamos. Operación 3
   int id3 = 202;
   sentencia.setInt(1, id3);
   sentencia.setString(2, "Elefantes voladores");
   sentencia.executeUpdate();

   if(id2 < id3) {
      conexion.rollback(spIntermedio);
   }

   conexion.commit();
} catch (SQLException ex) {
   System.out.println("Error al procesar la transacción.");
   System.out.println("Revirtiendo al estado anterior.");
   // Volvemos al estado anterior.
   conexion.rollback();
}
```

4.13. Ejecutar procedimientos almacenados.

Lenguaje dependiente del SGBD.

Por ejemplo:

- MySQL: parámetros de entrada, salida o E/S.
- Oracle: parámetros de salida o entrada/salida.

Interfaz `CallableStatement`: llamar desde Java a los procedimientos almacenados.

El objeto se obtiene mediante el método `prepareCall(String sql)` de `Connection`.

4.13.1. MySQL.

Si tenemos el procedimiento:

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `empleados`.`getNombreEmple` $$
CREATE PROCEDURE `empleados`.`getNombreEmple`
   (IN emp_id INT, OUT emp_nom VARCHAR(255))
BEGIN
   SELECT nombre INTO emp_nom
   FROM empleados
   WHERE id_empleado = emp_id;
END $$

DELIMITER ;
```

Lo podríamos ejecutar con:


```
try(Connection conexion =
      DriverManager.getConnection(URL_BASE_DATOS, USR, PWD)) {

   String procedimiento = "{call getNombreEmple (?, ?)}";

   try(CallableStatement  sentencia =
         conexion.prepareCall(procedimiento)) {
      try {
         // asignamos un valor al parámetro de entrada.
         sentencia.setInt(1, 100);
         // Registramos el parámetro de salida e indicamos su tipo.
         sentencia.registerOutParameter(2, java.sql.Types.VARCHAR);

         sentencia.execute();

         // recuperamos el dato.
         String nombreEmple = sentencia.getString(2);
         System.out.println(nombreEmple);
      } catch (SQLException ex) {
         System.out.println("Error al ejecutar el procedimiento.");
      }
   } finally {
      // dejamos el estado anterior.
      conexion.setAutoCommit(true);
   }
} catch (SQLException ex) {
   System.out.println("ERROR: no se pudo acceder a la BdD.");
}
```

4.13.2. Oracle.

Para ejecutar el procedimiento:

```
create or replace function sum4 (n1 in number,n2 in number)
return number is temp number(8);
begin
   temp :=n1+n2;
   return temp;
end;
```


Lo haríamos:

```
try (Connection con=DriverManager.getConnection(
      "jdbc:oracle:thin:@localhost:1521:xe","system","oracle")){
   CallableStatement stmt=con.prepareCall("{?= call sum4(?,?)}");
   stmt.setInt(2,10);
   stmt.setInt(3,43);
   stmt.registerOutParameter(1,Types.INTEGER);
   stmt.execute();
   System.out.println(stmt.getInt(1));
} catch (SQLException e) { }
```

4.14. Modelo–vista–controlador.

Patrón de diseño utilizado como guía de diseño de arquitecturas software con fuerte interacción con el usuario.

Se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

Propone la construcción de tres componentes distintos: (Figura 4.6).

- **Controlador**:
  - Responde a eventos (generalmente producidos por el usuario).
  - Genera peticiones para el **modelo** a partir de dichos eventos.
  - También puede interactuar con la **vista** si se solicitan cambios en cómo se muestra la información (*scroll, cambio de orden en una columna, etc.*).
- **Modelo**: representación de la información con la que trabaja el sistema:
  - Gestiona la manipulación de la misma (consultas, actualizaciones, etc.) y reglas de negocio.
  - Implementación y control de privilegios.
  - Envía a la **vista** la información que haya solicitado el usuario en cada caso.
  - El resto de componentes no deben preocuparse de la fuente de la que proviene la información (BdD, XML, JSON, etc.).
- **Vista**: muestra representa el modelo de forma gráfica para interactuar con el usuario.

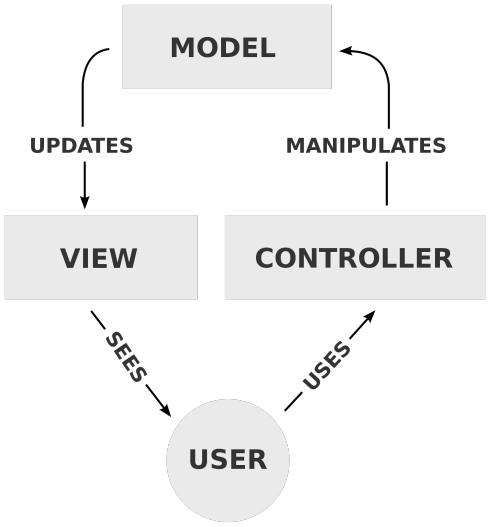

Figura 4.6: MVC. Imagen obtenida de aquí.

1. El usuario interactúa con la interfaz.
2. El **controlador** gestiona los eventos y recibe la operación que ha solicitado el usuario, generalmente a través de un gestor de eventos.
3. El **controlador** accede al modelo para manipularlo y posiblemente modificarlo, como por ejemplo un carro de la compra.
4. La función de mostrar los datos es responsabilidad de la **vista**.

Aunque esté en JavaScript, vamos a ver este ejemplo para que nos quede más claro.

También podemos consultar este otro en Java, del que puedes descargar el proyecto para NetBeans.

Nota: En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista. Por ejemplo en el MVC usado por *Apple* en su *framework* Cocoa. Suele citarse como Modelo-Interfaz-Controlador.

La interacción sería iniciada por el usuario con la interfaz (p.ej: navegador): (Figura 4.7).

1. El controlador examina la solicitud y decide que regla de negocio aplicar: usa el modelo adecuado.
2. El modelo contiene las reglas de negocio que procesan la solicitud y que dan lugar al acceso a los datos que necesita el usuario.
3. El controlador toma los datos que devuelve el modelo y selecciona la vista en la que se van a presentar esos datos al usuario.

4. El controlador devuelve los resultados a la vista tras procesar la solicitud.
5. La vista muestra al usuario el resultado.

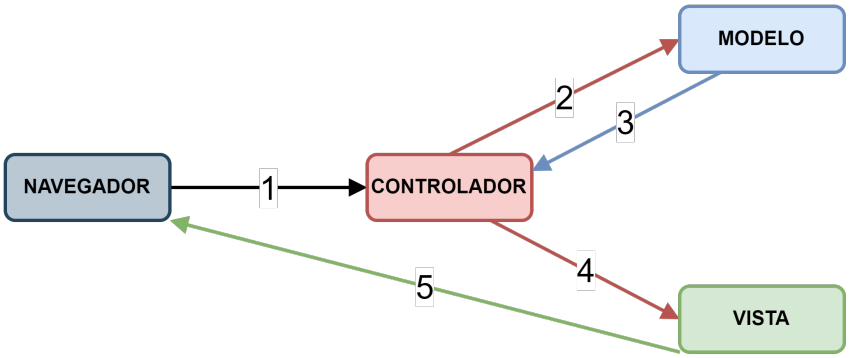

Figura 4.7: MIC.

Ventajas:

- Separa los datos de la representación visual de los mismos.
- Diseño de aplicaciones modulares.
- Reutilización de código.
- Facilidad a la hora de probar cada parte.
- Facilita el mantenimiento y la detección de errores.

Desventajas:

- Diseño más complejo.
- Aumenta la cantidad de ficheros (pero los hace más manejables, por lo que podría ser una ventaja también).

4.15. Arquitectura en capas.

Se definen:

- Capa de negocio:
  - Resuelve las funcionalidades para las que se ha diseñado la aplicación ofreciéndolas como servicios: clases con métodos que implementan estas funciones (p.ej: un servicio web).
  - Necesitamos objetos que representen las entidades de nuestra aplicación.

  - Accede a la capa de persistencia y envía/recibe datos de la capa de presentación siendo independiente de ambas.
- Capa de persistencia:
  - La persistencia de datos puede implementarse con múltiples y diferentes tecnologías: ficheros, gestores de BD relacionales diferentes (Oracle, mysql, Sqlite), de documentos (MongoDB...)
  - La implementación de la persistencia tiene que ser "transparente" para las capas de negocio y presentación.
  - Se necesita independencia del sistema de persistencia:
    - Distintas fuentes: BdD relacional, BdD OO, BdD de documentos, XML, JSON, etc.
    - Puede que se necesite acceder a varios sistemas desde la misma aplicación.
    - Las APIS de cada sistema serán diferentes.
- Capa de presentación: es la que ve y con la que interactúa el usuario.

4.15.1. *Data Access Object* (DAO).

Proporciona una interfaz única de acceso a los datos independientemente de cómo estén almacenados.

Permite que se independice la lógica del negocio del acceso a datos.

Habitualmente se crea un objeto DAO por cada elemento (entidad en E/R) del modelo.

Objetos de acceso a datos: clases DAO con JDBC y SQL.

Frameworks de persistencia mapeo O/R (objeto/relación): Hibernate, Spring, JPA (Java Persistence API)...

Ejercicios 4.4

Vamos a trabajar con la BdD de Pokemon obtenida de aquí.

En el aula virtual encontrarás el *script* para crearla en MySQL.

el diagrama relacional correspondiente es (Figura 4.8).

1. En la base de datos de Pokemon, la tabla `pokemon` y la tabla `pokemon_type` no están relacionadas, por ello, cuando se introduce un nuevo pokemon y se debe meter también su tipo. Cada uno se hará con un `INSERT` por lo que ambas operaciones se deben realizar mediante una transacción.

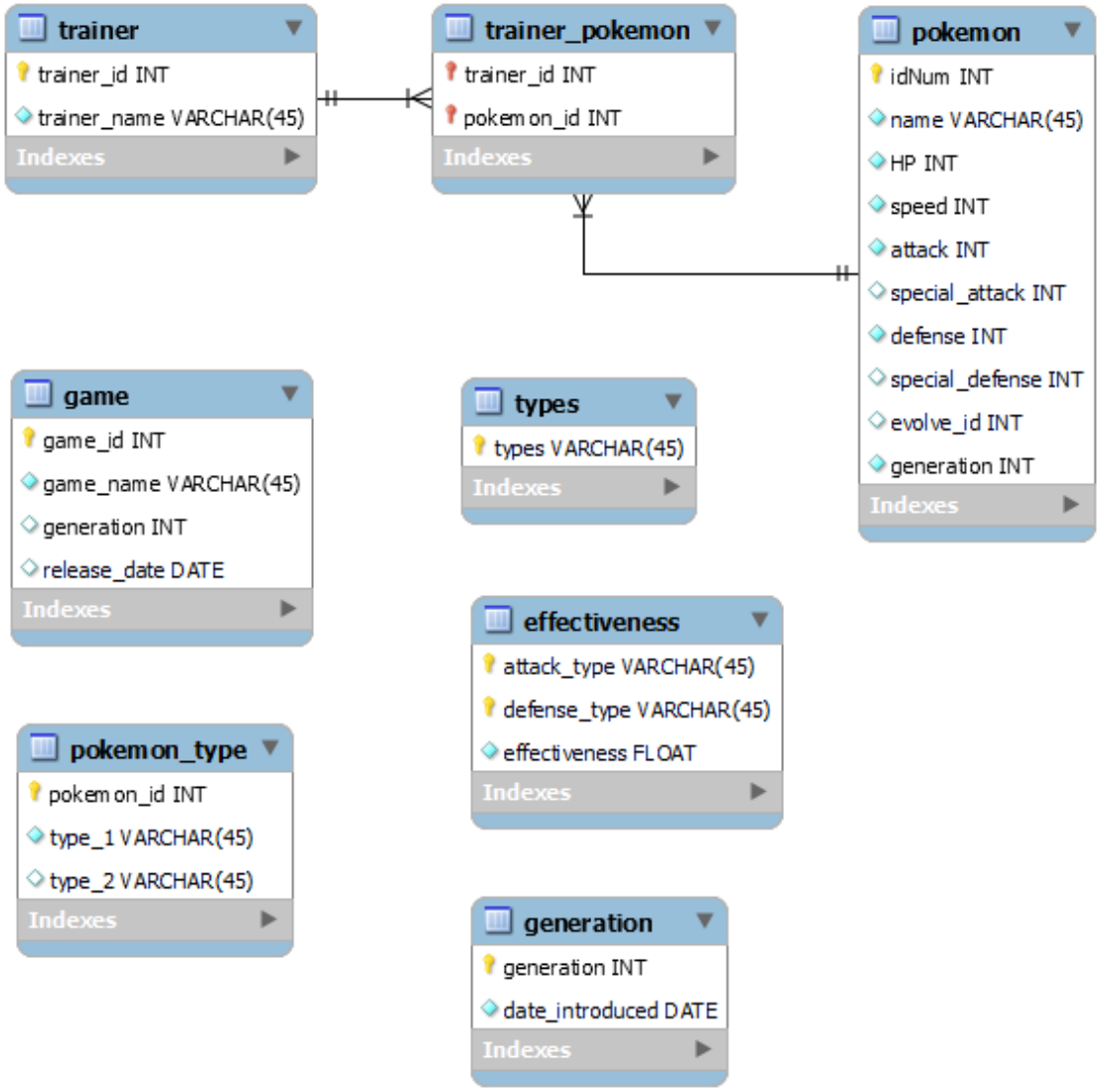

Figura 4.8: Esquema relacional en pata de gallo de la BdD de Pokemon.

Crea el código necesario para hacerlo.

2. Se pide que crees una implementación de `PokemonDAO` para interactuar con la BdD de Pokemon. Añade el código necesario para probar lo que has hecho.
3. Implementa un DAO para SQLite de la interfaz que creaste para los personajes de Star Wars.

   Presta especial atención a si tienes que realizar algún cambio a la definición que realizaste en su momento y apúntalo. No es grave ya que estamos aprendiendo pero ten en cuenta que si te sucediera en un proyecto real, tendrías que modificar todo el código.

   En el Aula Virtual tienes un ejemplo de proyecto con SQLite.

4.16. Bibliografía.

- Carlos Tessier Fernández (2021). Curso Acceso a Datos.
- Curso MEFP-IFCS02-AD. Ministerio de Educación y Formación Profesional.
- Documentación de la API de Java.
- Java & MySQL - Updating a ResultSet. Tutorialspoint.
- Using Transactions. The Java™ Tutorials. Modelo–vista–controlador. Wikipedia.

