6.- Características de las bases de datos objeto-relacionales.

Las BDOR las podemos ver como un híbrido de las BDR y las BDOO que intenta aunar los beneficios de ambos modelos, aunque por descontado, ello suponga renunciar a algunas características de ambos.

Los objetivos que persiguen estas bases de datos son:

- Mejorar la representación de los datos mediante la orientación a objetos.
- Simplificar el acceso a datos, manteniendo el sistema relacional.

En una BDOR se siguen almacenando tablas en filas y columnas, aunque la estructura de las filas no está restringida a contener escalares o valores atómicos, sino que las columnas pueden almacenar tipos estructurados (tipos compuestos como vectores, conjuntos, etc.) y las tablas pueden ser definidas en función de otras, que es lo que se denomina **herencia directa.**

Y eso, ¿cómo es posible?

Pues porque internamente tanto las tablas como las columnas son tratados como objetos, esto es, se realiza un mapeo objeto-relacional de manera transparente.

6.- Características de las bases de datos objeto-relacionales.

Como consecuencia de esto, aparecen **nuevas características**, entre las que podemos destacar las siguientes:

- **Tipos definidos por el usuario**. Se pueden crear nuevos tipos de datos definidos por el usuario, y que son compuestos o estructurados, esto es, será posible tener en una columna un atributo multivaluado (un tipo compuesto).
- **Tipos Objeto**. Posibilidad de creación de objetos como nuevo tipo de dato que permiten relaciones anidadas.
- **Reusabilidad**. Posibilidad de guardar esos tipos en el gestor de la BDOR, para reutilizarlos en tantas tablas como sea necesario
- **Creación de funciones**. Posibilidad de definir funciones y almacenarlas en el gestor. Las funciones pueden modelar el comportamiento de un tipo objeto, en este caso se llaman métodos.
- **Tablas anidadas**. Se pueden definir columnas como arrays o vectores multidimensionales, tanto de tipos básicos como de tipos estructurados, esto es, se pueden anidar tablas
- **Herencia** con subtipos y subtablas.

Estas y otras características de las bases de datos objeto-relacionales vienen recogidas en el estándar SQL 1999 .

6.1.- El estándar SQL99.

La norma ANSI SQL1999 (abreviadamente, SQL99) extiende el estándar SQL92 de las Bases de Datos Relacionales, y da cabida a nuevas características orientadas a objetos preservando los fundamentos relacionales.

Algunas extensiones que contempla este estándar y que están relacionadas directamente con la orientación a objetos son las siguientes:

- **Extensión de tipos de datos**.
  - Nuevos tipos de datos básicos para datos de caracteres de gran tamaño, y datos binarios de gran tamaño (Large Objects)
  - Tipos definidos por el usuario o tipos estructurados.
  - Tipos colección, como los arrays, set, bag y list.
  - Tipos fila y referencia
- **Extensión de la semántica de datos**.
  - Procedimientos y funciones definidos por el usuario, y almacenados en el gestor.
  - Un tipo estructurado puede tener métodos definidos sobre él.

6.1.- El estándar SQL99.

Por ejemplo, el siguiente segmento de SQL crea un nuevo tipo de dato, un tipo estructurado de nombre profesor y que incluye en su definición un método, el método **sueldo()** .

```
CREATE TYPE profesor AS (
id INTEGER,
nombre VARCHAR (20),
sueldo_base DECIMAL (9,2),
complementos DECIMAL (9,2),
INSTANTIABLE NOT FINAL

METHOD sueldo() RETURNS DECIMAL (9,2));
CREATE METHOD sueldo() FOR profesor
BEGIN
.....

END;
```

7.- Gestores de Bases de Datos Objeto-Relacionales.

Podemos decir que un sistema gestor de bases de datos objeto-relacional (SGBDOR) contiene dos tecnologías; la tecnología relacional y la tecnología de objetos, pero con ciertas restricciones.

A continuación te indicamos algunos ejemplos de **gestores objeto-relacionales**, tanto de código libre como propietario, todos ellos con soporte para Java:

- De código abierto:
  - **PostgreSQL**
  - **Apache Derby**
- De código propietario
  - **Oracle**
  - **First SQL**
  - **DB2 de IBM**

7.1.- Instalación del Gestor PostgreSQL.

El código fuente de PostgreSQL está disponible bajo la licencia BSD. Esta licencia te da libertad para usar, modificar y distribuir PostgreSQL en cualquier forma, ya sea junto a código abierto o cerrado. Además PostgreSQL. incluye API para diferentes lenguajes de programación, entre ellos Java y .NET

Descarga del software

Accedemos a la web oficial de PostgreSQL **http://www.postgresql.org/**

Desde **Downloads**, hacemos click en el Sistema Operativo sobre el que vamos a realizar la instalación, en este caso **Windows 7**.

Click sobre **"Download the one click installer"** que nos llevará a la siguiente página.

Descarga del software

Descargamos la última versión para Windows que corresponda: **Win x86-32 o Win x86-64**(para Sistemas Operativos de 32 bits o 64 bits respectivamente). Guardamos el fichero y ejecutamos el instalador.

7.1.- Instalación del Gestor PostgreSQL.

Ejecución del Instalador

Al hacer **doble click sobre el instalador** comenzará la instalación de PostgreSQL. La primera pantalla es la de bienvenida. Pulsamos sobre el **botón siguiente**.

Directorio de Instalación

Indicamos el **directorio de instalación** y damos a siguiente. En este directorio se almacenará el sistema gestor de bases de datos.

7.2.- Tipos de datos: tipos básicos y tipos estructurados.

Como ya sabes, los SGBDOR incorporan un conjunto muy rico de tipos de datos. PostgreSQL no soporta herencia de tipos pero permite **definir nuevos tipos de datos mediante los mecanismos de extensión**.

Te vamos a comentar tres categorías de tipos de datos que encontramos en PostgreSQL:

- **Tipos básicos**: el equivalente a los tipos de columna usados en cualquier base de datos relacional.
- **Tipos compuestos**: un conjunto de valores definidos por el usuario con estructura de fila de tabla, y que como tal puede estar formada por tipos de datos distintos.
- **Tipos array**: un conjunto de valores distribuidos en un vector multidimensional, con la condición de que todos sean del mismo tipo de dato (básico o compuesto). Ofrece más funcionalidades que el array del estándar SQL99.

7.2.- Tipos de datos: tipos básicos y tipos estructurados.

Entre los **tipos básicos**, podemos destacar:

- **Tipos numéricos**. Aparte de valores enteros, números de coma flotantes, y números de precisión arbitraria, PostgreSQL incorpora también un tipo entero auto-incremental denominado serial.
- **Tipos de fecha y hora**. Además de los típicos valores de fecha, hora e instante, PostgreSQL incorpora el tipo interval para representar intervalos de tiempo.
- **Tipos de cadena de caracteres**. Prácticamente los mismos que en cualquier BDR.
- **Tipos largos**. Como por ejemplo, el tipo BLOB para representar objetos binarios. En la actualidad presentes en muchas BDR como MySQL.

Los **tipos compuestos** de PostgreSQL son el equivalente a los **tipos estructurados** definidos por el usuario del estándar SQL99. De hecho, son la base sobre la que se asienta el soporte a objetos.

7.2.- Tipos de datos: tipos básicos y tipos estructurados.

Por ejemplo, podemos crear un nuevo tipo, el tipo dirección a partir del nombre de la calle (**varchar**), del número de la calle (**integer**) y del código postal (**integer**), de la siguiente forma:

```
CREATE TYPE direccion AS (
calle varchar,
numero integer,
codigo_postal integer);
```

y luego definir la tabla de nombre afiliados, con una columna basada en el nuevo tipo:

```
CREATE TABLE afiliados AS(
afiliado_id serial,
nombre varchar(45),
apellidos varchar(45),
domicilio direccion);
```

7.3.- Conexión mediante JDBC.

La conexión de una aplicación Java con PostgreSQL se realiza mediante un conector tipo JDBC.

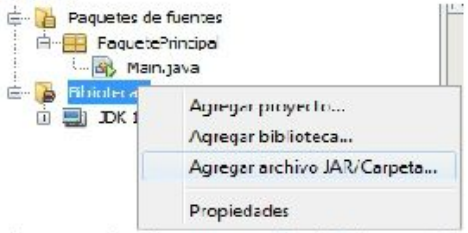
Descarga del driver JDBC de PostgreSQL

- Accedemos a la web oficial del driver JDBC para PostgreSQL http://jdbc.postgresql.org
- En la pestaña **Home**, seleccionamos el menú **Download** dentro de **About**; o bien, vamos directamente a la página de descargas http://jdbc.postgresql.org/download.html

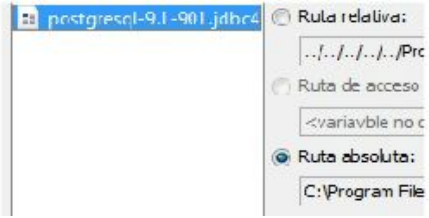
- En la sección **Current Version**, seleccionamos el driver más adecuado para el Java que tengamos instalada (normalmente, la última disponible para el **JDBC4 Postgresql driver**)
- Guardamos el fichero **.jar** en un directorio, y anotamos la ruta para referencia posterior.

Integración en un proyecto del Netbeans

- Para integrar el driver en un proyecto del **Netbeans**, clic con el botón derecho en **Bibliotecas** para ejecutar el comando **Agregar archivo JAR/Carpeta**



- Se introduce la ruta donde se guardó el fichero **.jar** descargado, y listo.



7.3.- Conexión mediante JDBC.

import java.sql.*

Recuerda que en JDBC, una base de datos está representada por una URL. La cadena correspondiente tiene una de las tres formas siguientes:

- jdbc: postgresql: base de datos
- jdbc: postgresql: //host/base de datos
- jdbc: postgresql: //host: puerto/base de datos

El nombre de host por defecto, del servidor PostgreSQL, será **localhost**, y el puerto por el que escucha el 5432. Como ves, esta cadena es idéntica a la empleada por otros sistemas gestores de bases de datos.

Para conectar con la base de datos, utilizaremos el método **DriverManager.getConnection()** que devuelve un objeto **Connection** (la conexión con la base de datos). **Una de las posibles sintaxis de este método es:**

Connection conn = DriverManager.getConnection(url, username, password);

Una vez abierta, la conexión se mantendrá operativa hasta que se llame a su método **close()** para efectuar la desconexión. Si al intentar conectar con la base de datos ésta no existe, se generará una excepción del tipo **PSQLException** "FATAL: no existe la base de datos ...". En cualquier caso se requiere un bloque **try-catch** .

7.3.- Conexión mediante JDBC.

```
//cadena url de la base de datos anaconda en el servidor local (no hay
//que indicar el puerto si es el por defecto)
String url = "jdbc:postgresql://localhost/anaconda";

//conexión con la base de datos
Connection conn = null;

try {
  //abre la conexión con la base de datos a la que apunta el url
  //mediante la contraseña del usuario postgres
  conn = DriverManager.getConnection(url, "postgres", "1234");

} catch (SQLException ex) {
  //imprime la excepción
  System.out.println(ex.toString());
} finally {

  //cierra la conexión
  conn.close();
}
```

7.4.- Consulta y actualización de tipos básicos

PostgreSQL implementa los objetos como filas, las clases como tablas, y los atributos como columnas. Hablaremos por tanto de tablas, filas y columnas, tal y como lo hace PostgreSQL.

Para interactuar con PostgreSQL desde Java, vía JDBC, debemos enviar sentencias SQL a la base de datos mediante el uso de comandos.

Por tanto, si nuestra conexión es **conn**, para enviar un comando **Statement** haríamos lo siguiente:

- Crear la sentencia, por ejemplo.: Statement sta = conn.createStatement();
- Ejecutar la sentencia:
  - sta.executeQuery(string sentenciaSQL); si sentenciaSQL es una consulta (SELECT)
  - sta.executeUpdate(string sentenciaSQL); si sentenciaSQL es una actualización (INSERT, UPDATE O DELETE)
  - sta.execute(string sentenciaSQL); si sentenciaSQL es un CREATE, DROP, o un ALTER

Como ves, en PostgreSQL se utilizan estos comandos como en cualquier otra BDR.

7.5.- Consulta y actualización de tipos estructurados.

Imaginemos que tenemos el tipo estructurado dirección:

```
CREATE TYPE direccion AS (
calle varchar, numero integer, codigo_postal varchar);
```

y la tabla afiliados, con una columna basada en el nuevo tipo:

```
CREATE TABLE afiliados(afiliado_id serial, nombre varchar, apellidos varchar, domicilio direccion);
```

Insertamos valores en una tabla con un tipo estructurado Se puede hacer de dos formas:

Pasando el valor del campo estructurado entre comillas simples (*lo que obliga a encerrar entre comillas dobles cualquier valor de cadena dentro*), y paréntesis para encerrar los subvalores separados por comas:

```
INSERT INTO afiliados (nombre, apellidos, domicilio)
 VALUES ('Onorato', 'Maestre Toledo', '(“Calle de Rufino”, 56, 98080)');
```

Mediante la función ROW que permite dar valor a un tipo compuesto o estructurado.

```
INSERT INTO afiliados (nombre, apellidos, direccion)
VALUES ('Onorato', 'Maestre Toledo', ROW('Calle de Rufino', 56, 98080));
```

7.5.- Consulta y actualización de tipos estructurados.

Referenciar una subcolumna de un tipo estructurado

Se emplea la notación punto, '.' , tras el nombre de la columna entre paréntesis, (tanto en consultas de selección, como de modificación) . Por ejemplo:
SELECT (domicilio).calle FROM afiliados WHERE (domicilio).codigo_postal=98080

devolvería el nombre de la calle Maestre Toledo. Los paréntesis son necesarios para que el gestor no confunda el nombre del campo compuesto con el de una tabla.

Eliminar el tipo estructurado

Se elimina con DROP TYPE, por ejemplo DROP TYPE direccion;

7.6.- Consulta y actualización de tipos array..

PostgreSQL permite especificar **vectores multidimensionales como tipo de dato para las columnas** de una tabla. La única condición es que todos sus elementos sean del mismo tipo.

Por ejemplo:

- Declaración de una columna de tipo vector: nombre_columna tipo_dato[]
- Declaración de una columna de tipo matriz multidimensional: nombre_columna tipo_dato[][]

donde como se ve, sólo hay que agregar uno o más corchetes '[]' al tipo de dato.
Aunque PostgreSQL permite especificar el tamaño de cada dimensión en la declaración, y acepta escribir:

nombre_columna tipo_dato[5] o bien nombre_columna tipo_dato[2][3] , las versiones actuales ignoran estos valores en la práctica, de manera que los **array** declarados de esta forma tienen la misma funcionalidad que los del ejemplo anterior.

En realidad ninguna de estas declaraciones es conforme al estándar SQL99, que sólo contempla el tipo **array** como columnas de vectores unidimensionales declarados con la palabra reservada **array**:

7.6.- Consulta y actualización de tipos array..

En el siguiente ejemplo puedes ver la creación de una tabla con una columna de tipo **array** de **varchar** y como se insertan y consultan valores:

```
//comando
sta = conn.createStatement();
sta.execute("DROP TABLE IF EXISTS tareas");
//crea una tabla con una columna matricial de tipo varchar
sta.execute("CREATE TABLE tareas(comercial_id integer,"
        + "agenda varchar[][])");
//inserta un registro de dos tareas por día para el comercial número 3
//durante dos días (día 1: [0][0],[0][1]; día 2:[1][0],[1][1])
sta.executeUpdate("INSERT INTO tareas VALUES(3,"
        + "'{{\"reunión 9:30\",\"comida 14:30\"},"
        + "{\"reunión 8:30\",\"cena 22:30\"}}')");
//consulta todas las tarea del segundo día del comercial número 3
ResultSet rst = sta.executeQuery("SELECT agenda[2:2] "
        + "FROM tareas WHERE comercial_id=3");
//muestra el resultado
while (rst.next()) {
  System.out.println(rst.getString(1));
}
//consulta la segunda tarea del primer día del comercial número 3
rst = sta.executeQuery("SELECT agenda[1][2] "
        + "FROM tareas WHERE comercial_id=3");
```