UAX FP

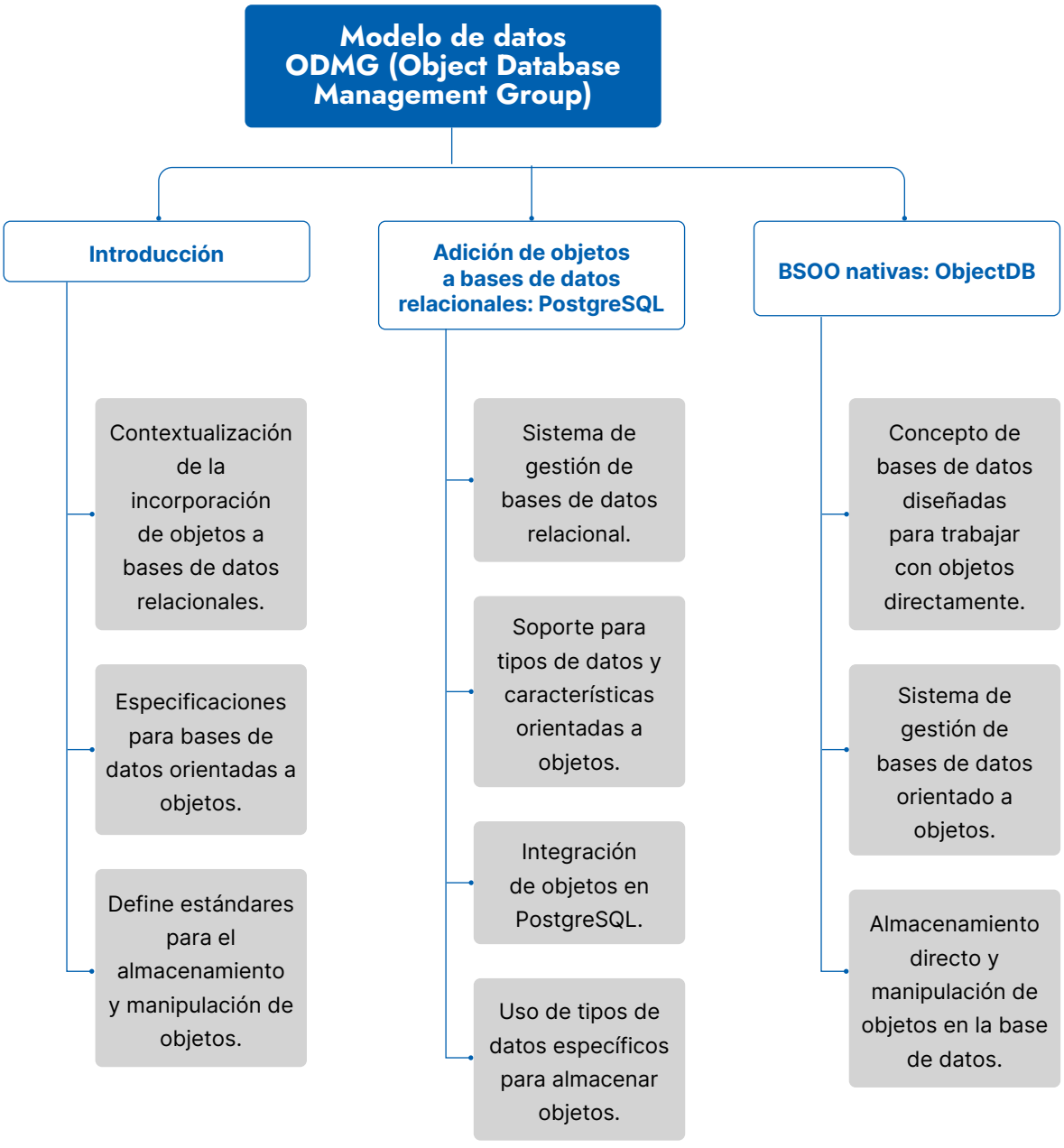
UF4

El modelo de datos ODMG

Acceso a datos

ÍNDICE

Mapa conceptual



01 Introducción. El modelo de datos ODMG

1.1. Evolución de los SGBD

DEFINICIÓN

Las **BDR** surgen a partir del modelo relacional, como un sistema que representa fielmente la realidad, con una gran solidez por la lógica relacional subyacente. Este modelo representa la perspectiva estática del modelado de la aplicación, y en él se desglosan todos los datos hasta niveles atómicos, ya que, no se permiten ni valores multivaluados ni compuestos.

El modelo relacional ha sufrido el **desfase objeto-relacional**, en el cual los lenguajes de programación realizaron una evolución de las estructuras, adoptando la metodología de la orientación a objetos. Esto provoca que diseñemos por un lado la base de datos, siguiendo el modelo relacional, y que también necesitemos, por otra parte, el diseño de clases de la aplicación. Dichos diseños, que no suelen coincidir, podemos «encajarlos» con las herramientas **ORM** para mitigar el desfase objeto-relacional.
De este desfase surge la necesidad de añadir un diseño más orientado a objetos dentro de la propia base de datos. Si analizamos un esquema orientado a objetos y tratamos de aplicarle la **teoría de la normalización**, nos encontraremos que los objetos se nos «descomponen» en varias entidades, lo que provoca que aparezcan muchas tablas.

IMPORTANTE

Esta cantidad elevada de tablas conduce, como consecuencia, a un aumento de las referencias entre estas, lo que incrementa considerablemente las relaciones entre ellas, y, por tanto, un mayor número de restricciones que controlar (**FOREIGN KEY**), lo cual supone un acrecentamiento en el número de comprobaciones que debe realizar el SGBD.

Así pues, las **BDOO (bases de datos orientadas a objetos)** permiten una definición de tipos complejos de datos, frente a los simples que incorpora el SGBDR, lo que posibilita la definición de tipos estructurados e incluso multivaluados. Con todo esto, las BDOO deberían simplemente permitir el diseño mediante objetos (de forma similar al UML), indicando los objetos que participan, sus atributos y métodos, y las relaciones que afectan a estos, ya sean participaciones o herencias.

Estas BDOO no terminan de despegar, y algunas alternativas que implementan las soluciones comerciales consisten en dotar al sistema de bases de datos relacionales de las capacidades semánticas de la orientación a objetos, con lo que aparecen las **BDOR (bases de datos objeto-relacionales)**.

!

Recuerda que estos valores estructurados y multivaluados violan la teoría de la normalización del modelo relacional.

Estos valores añadidos a los SGBD tradicionales minimizan el impacto de los ORM y acercan el modelo relacional al diseño orientado a objetos, por lo que los convierte en más cercanos al lenguaje de programación.

1.2. BDOO

IMPORTANTE

En estas bases de datos, todo lo que se almacena son objetos, es decir, entidades con un **estado** (determinado por el valor de sus atributos), que pueden modificarse mediante un **comportamiento** (determinado por las acciones que podemos hacer con ellos, sus métodos) y que posee un identificador único, que hace los objetos diferentes.

Las BDOO deben permitir, entre otras cuestiones:

Identificación de objetos (OID)	El sistema debe independizar su valor respecto de su estado.
Encapsulamiento	Los datos de implementación están en los objetos.
Navegabilidad	Permite acceder, a partir de objetos, a datos almacenados en otros objetos mediante referencias, evitando las uniones típicas del modelo relacional. Los tipos de datos admitidos se pueden dividir en atómicos, estructurados y colecciones, y estas últimas, a su vez, en ordenadas y desordenadas, y pueden contener o no elementos repetidos.
Herencia	Permite la creación de entidades a partir de entidades ya existentes previamente. Este paso tendrá estrecha relación con las especializaciones del modelo relacional, ya que la herencia es la sustituta natural de la especialización en los modelos OO.

Identidad	El sistema debe ofrecer un mecanismo de identificación de objetos, aparte de las claves principales.
Consultas sencillas	El sistema debe proveer un lenguaje sencillo de consultar. Normalmente se basan en SQL, y **OQL *(object query language)*** es la variante más reconocida.

1.3. Implementación del estándar ODMG

El **ODMG (Object Data Management Group)** representa a un conglomerado de compañías de la industria de las bases de datos, que propusieron en su momento un conjunto de características que debían implementar las BDOO.
La última versión es ODMG v3.0, publicada en el año 2000. Posteriormente, se ha transferido la tarea de seguir con las especificaciones a **Object Management Group (OMG)**.
En cuanto a **rasgos que deben implementar los SGBDOO**, según ODMG, son:

- Modelado de datos, según las especificaciones de OMG.
- Lenguaje de definición de objetos: ODL.
- Lenguaje de consulta de objetos: OQL.
- Enlazado con el lenguaje de programación, conocido como ***binding***, principalmente para C++ y Java.

1.4. BDOR

En el apartado anterior, hemos visto algunas características generales de las BDOO, las cuales se definen solo con objetos.

¿SABÍAS QUE...?
La norma ANSI SQL1999 (SQL99, y posteriormente la SQL2003, como continuación de SQL92, en la cual se adaptan las características del modelo relacional) permite **añadir características de orientación** a objetos a los BDR. Esto permite que SGBDR sólidos hayan «adoptado» e implementado dichas características requeridas por el ODMG.

Entre estas características destacan:

- Definición de nuevos tipos de datos por parte del usuario.
- Adaptación para dar cabida a tipos de datos binarios de gran tamaño, como imágenes y documentos.

- Posibilidad de almacenar elementos compuestos, como **arrays**.
- Almacenamiento directo de referencias a otras tablas.
- Definición de objetos y herencia.
- Definición de funciones que manejan las estructuras anteriormente definidas.

IMPORTANTE

Conclusión:

- Partiendo de los SGBDR, podemos añadir las capacidades de la orientación a objetos, para conseguir complementar el modelado que deseemos.
- Si partimos exclusivamente del modelo OO, y no necesitamos las características relacionales, entonces precisaremos el modelo de una BDOO.
- Aun así, ambos sistemas deben mantener las características de persistencia, gestión de transacciones y comunicación con los lenguajes de programación.
- Las estructuras de almacenamiento de la información siguen siendo las tablas, aunque el concepto de «fila», tal y como lo conocemos, puede variar, debido a la presencia de datos estructurados.

02 Adición de objetos a bases de datos relacionales. PostgreSQL

Visto el contexto de los SGBD existentes, nos centramos en este apartado en **PostgreSQL**, como el sistema gestor de bases de datos objeto-relacionales de código libre que más repercusión ha tenido, y rival directo de la gran Oracle.

CLAVES Y CONSEJOS

Para una mejor comprensión de esta unidad, es importante disponer de un servidor Postgres. Puedes instalarlo en tu sistema de la manera que quieras o mediante contenedor Docker, como se ha visto en unidades anteriores.

2.1. Definición de tipos en PostgreSQL

En **PostgreSQL** aparecen los tipos habituales que hay en los **SGBD** modernos, pero cabe resaltar los habituales numéricos, textos y fechas, con los que no entraremos en detalles. Destacan, además, tipos especiales para almacenar direcciones de internet ***(network address types)***, **XML** y **JSON** para el guardado y procesado de dichos formatos, tipos propios y colecciones. Con esto, podemos fijarnos en los grandes avances de **PostgreSQL**, que, incluso, podríamos considerar como **objeto-relacional-documental**.

Introducción a los Tipos de Datos en PostgreSQL

En PostgreSQL, al igual que en otros sistemas de gestión de bases de datos (SGBD), los tipos de datos juegan un papel crucial en la definición de la estructura de nuestras tablas y en la forma en que almacenamos y manipulamos la información. Cada columna de una tabla en PostgreSQL tiene un tipo de dato asociado que determina el tipo de valores que puede contener.

Los tipos de datos en PostgreSQL son herramientas fundamentales para definir la estructura y el comportamiento de nuestras bases de datos. Al comprender estos tipos y cómo se utilizan, podemos diseñar esquemas de base de datos efectivos y garantizar la integridad y consistencia de nuestros datos. Además, la variedad de tipos especiales en PostgreSQL nos permite manejar una amplia gama de datos, desde simples números hasta estructuras complejas como XML y JSON.

Tipos de Datos Comunes:

1. **Numéricos:** Estos tipos de datos se utilizan para almacenar números enteros y decimales.
2. **Texto:** Los tipos de datos de texto se utilizan para almacenar cadenas de caracteres y texto.
3. **Fecha y Hora:** Estos tipos de datos se utilizan para almacenar información relacionada con fechas y horas.
4. **Booleanos:** Un tipo especial que solo puede tener dos valores: verdadero (true) o falso (false).
5. **Tipos Especiales:** PostgreSQL también proporciona tipos de datos especiales para manejar información específica, como direcciones de Internet, XML, JSON, etc.

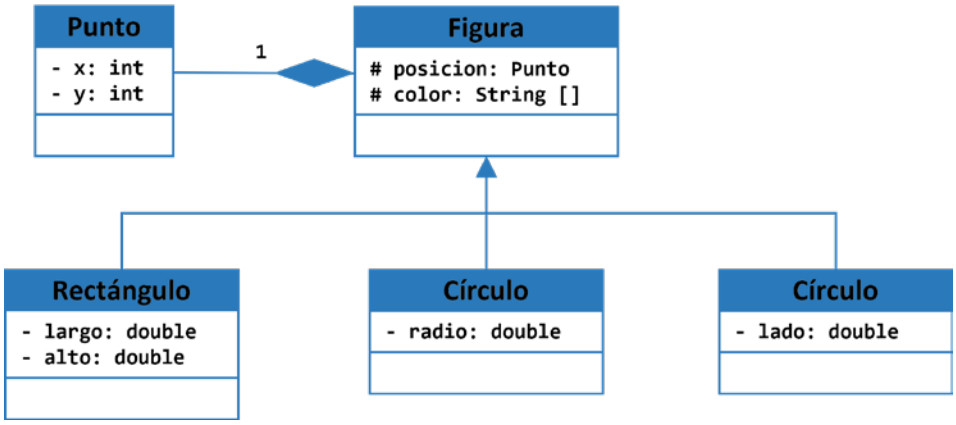
Resumen en Tabla:

TIPO DE DATO	DESCRIPCIÓN
INTEGER	Número entero de tamaño normal.
BIGINT	Número entero de tamaño grande.
SMALLINT	Número entero de tamaño pequeño.
NUMERIC/DECIMAL	Número decimal con precisión variable.
REAL	Número de coma flotante de precisión simple.
DOUBLE PRECISION	Número de coma flotante de precisión doble.
CHAR(n)	Cadena de caracteres de longitud fija.
VARCHAR(n)	Cadena de caracteres de longitud variable.
TEXT	Cadena de texto de longitud variable sin límite.
DATE	Tipo de dato para almacenar fechas.
TIME	Tipo de dato para almacenar horas.
TIMESTAMP	Tipo de dato para almacenar fecha y hora.
BOOLEAN	Tipo de dato para almacenar valores verdadero/falso.

EJEMPLO

ESQUEMA DE TRABAJO

Para trabajar en los ejemplos siguientes, partiremos del siguiente esquema UML, el cual incluye cuatro elementos principales: PUNTO, FIGURA, CÍRCULO y RECTÁNGULO. Estos elementos representan entidades o clases que pueden ser utilizadas para modelar diversos conceptos en un sistema, como puntos en un plano, formas geométricas, etc. Cada uno de estos elementos tiene sus propias características y puede relacionarse de diversas formas en el contexto de la aplicación que estemos desarrollando."



A. Creación de tipos enumerados

VÍDEO LAB

Para consultar el siguiente vídeo sobre Acceso a datos en PostgreSQL, escanea el código QR o pulsa aquí.

DEFINICIÓN

Los **tipos enumerados** son valores tipos que permiten solo unos valores concretos, habitualmente conocidos también como dominios.

En PostgreSQL podemos crearlos de la siguiente manera:

```
CREATE TYPE nombre_enumerado AS ENUM
    ( [ 'valor' [, ... ] ] );
```

Como ejemplo, podemos crear los posibles colores básicos para pintar unas figuras o tipos de calle para una futura dirección.

```
create type  colores_basicos as
enum('#FF0000','#00FF00','#0000FF');
create type TipoCalle as enum ('Calle','Avenida',','Plaza');
```

Esta manera de definir este tipo nos evitará las comprobaciones de valores (cláusulas **CHECK**) existentes en algunos SGBD relacionales.

B. Creación de tipos estructurados

Los **tipos estructurados** son los precursores de los objetos propiamente dichos. En la programación estructurada, a partir de los tipos básicos, podíamos crear estructuras de datos donde todos los elementos fueran iguales (vectores, arrays y colecciones), o estructuras donde sus elementos podían ser de distintos tipos.
Estas estructuras evolucionaron a los actuales objetos al añadirles comportamiento y otras características más.
En el modelo relacional, dado que debemos respetar la atomicidad de los datos, no podíamos generar dichas estructuras. En **Postgres** podemos crear estos **nuevos tipos de datos estructurados** con la siguiente sintaxis, muy parecida a la creación de una tabla:

» Definición de tipos

```
CREATE TYPE nombre_tipo AS
    ( nombre_atributo tipo_de_dato,
        [, ...]     -- uno o varios
    );
```

» Ejemplos de definición de tipos

```
CREATE TYPE Punto AS (
    x integer,
    y integer
);
```

» Reutilizando los tipos

```
CREATE TYPE Direccion AS (
    tipo TipoCalle,
    calle varchar,
    numero int
);
```

» Utilizando los tipos en nuestras tablas

```
CREATE TABLE persona (
    idPersona serial,
    nombre varchar,
    direccion Direccion
);
```

» Inserción de datos

```
INSERT INTO persona(nombre) VALUES ('Joange');
INSERT INTO persona(nombre,direccion) VALUES ('Joange', null);
INSERT INTO persona(nombre,direccion)
VALUES ('Joange', ROW('Calle','Calvario',1));
```

» Consultas

```
SELECT direccion FROM persona p;
SELECT (direccion).calle FROM persona p;
```

Y, además, en el segundo caso, reutilizando los tipos:

```
create type Direccion as (
  tipo TipoCalle
,  calle varchar,
  numero int
);
```

Ya podemos utilizar estos tipos en nuestras tablas, por ejemplo:

```
create table persona(
  idPersona serial,
  nombre varchar,
  direccion Direccion
);

insert into persona(nombre) values('Joange');
insert into persona(nombre,direccion) values('Joange',null);
insert into persona(nombre,direccion)
values('Joange',('Calle','Calvario',1));

select direccion from persona p;
select (direccion).calle from persona p;
```

A los campos a los que pertenecen los tipos creados no se les puede aplicar restricciones de **NOT**, **NULL**, **DEFAULT** ni **CHECK**.

- La creación de tipos tiene sentido en datos que no tienen existencia por sí mismos, que necesitan ser embebidos en otras estructuras o tablas.
- Al usarse dentro de una tabla y manipular la inserción, esta se hará en bloque, entre paréntesis, ya que determina una estructura.
- Para seleccionar un subtipo, deberemos encerrar el tipo general entre paréntesis, ya que, si no, Postgres lo confunde con una tabla y genera un error.

2.2. Definición de «clases»

Vamos a crear una **clase Figura**, que será el punto inicial de una herencia del modelo presentado al principio de la unidad. Fijémonos en lo que incorpora respecto a las implementaciones del modelo relacional.
La **Figura** contiene una clave principal, e incluirá un **Punto** para ubicarla en el plano. Además, contiene una colección de colores, para realizar posibles degradados, guardada como un array. Guardar colecciones es también una capacidad añadida que no admite el modelo relacional, dada la ausencia de multivaluados.

IMPORTANTE

Una clase se crea con la misma sintaxis de una tabla, ya que, a efectos prácticos, es lo mismo desde el punto de vista estructural. Posteriormente, la **herencia** sí que distingue que una tabla «hereda» de otra.

```
-- Creación de tabla Figura
CREATE TABLE Figura (
    fID serial primary key, -- identificador
    posicion Punto, -- posición que ocupa
    color TEXT[] -- color(es) de la figura
```

Para **insertar nuevos registros**, hay que tener en cuenta que:

- Los elementos de tipo **Punto** deben almacenarse mediante un constructor, que crea una «fila» abstracta, denominada **ROW**, o los paréntesis.
- Para los arrays, también necesitamos un constructor denominado **ARRAY**, con una lista de elementos.

```
insert into Figura(posicion,color) values
(row(0,0),array['FFFFFF','#00CC00']);
```

A partir de ahí, crearemos nuevas clases para representar los círculos, cuadrados y rectángulos a partir de la **Figura**, mediante la herencia.
La sintaxis es la siguiente:

```
create table tabla_heredera(
  -- definición de atributos de la tabla
) inherits (super_tabla);
```

Como podemos observar, simplemente añadimos **inherits** para crear la relación de herencia. Para el diseño que teníamos anteriormente:

```
create table Rectangulo(
        alto int,
        ancho int
) inherits (Figura);
```

```
create table Cuadrado(
        lado int
) inherits (Figura);

create table Circulo(
        radio int
) inherits (Figura);
```

Insertamos algunas filas, fijándonos en que tenemos que incluir también los atributos de la superclase.

```
insert into Cuadrado(posicion,color,lado) values
 (row(10,10),array['#00BBCC','#BBCC00'],40); insert into
 Cuadrado(posicion,color,lado) values
 (row(10,15),array['#AA6633','#CCFF00'],27); insert into
Circulo(posicion,color,radio) values
 (row(30,25),array['#BBCC','#CCCC00'],20); insert into
Circulo(posicion,color,radio) values (row(10,-
10),array['#00BBCC','#CCCC00'],20); insert into
Rectangulo(posicion,color,alto,ancho) values
(row(10,5),array['#00BBCC','#CCCC00'],20,50); insert
into Rectangulo(posicion,color,alto,ancho) values
(row(30,-10),array['#00BBCC','#CCCC00'],20,50);
```

Si visualizamos las inserciones con DBeaver, observamos las agrupaciones de tipos:

figura | Enter a SQL expression to filter results (use Ctrl+Space)

Grilla / Texto

	fid	posicion		color
		x	y	
1	1	2	3	{#AABBCC,#FFCC00}
2	2	0	0	{FFFFFF,#00CC00}
3	3	-2	7	{#AABB00,#FFCCCC}
4	7	10	5	{#00BBCC,#CCCC00}

rectangulo | Enter a SQL expression to filter results (use Ctrl+Space)

Grilla / Texto

	123 fid	posicion		color	123 alto	123 ancho
		123 x	123 y			
1	7	10	5	{#00BBCC,#CCCC00}	20	50
2	8	30	-10	{#00BBCC,#CCCC00}	20	50
3	9	-10	15	{#00BBCC,#CCCC00}	20	50
4	10	15	5	{#00BBCC,#CCCC00}	20	50

cuadrado | Enter a SQL expression to filter results (use Ctrl+Space)

Grilla / Texto

	123 fid	posicion		color	123 lado
		123 x	123 y		
1	4	10	5	{#00BBCC,#CCCC00}	20
2	11	10	10	{#00BBCC,#BBCC00}	40
3	12	10	15	{#AA6633,#CCFF00}	27

circulo | Enter a SQL expression to filter results (use Ctrl+Space)

Grilla / Texto

	123 fid	posicion		color	123 radio
		123 x	123 y		
1	5	30	25	{#BBCC,#CCCC00}	20
2	6	10	-10	{#00BBCC,#CCCC00}	20

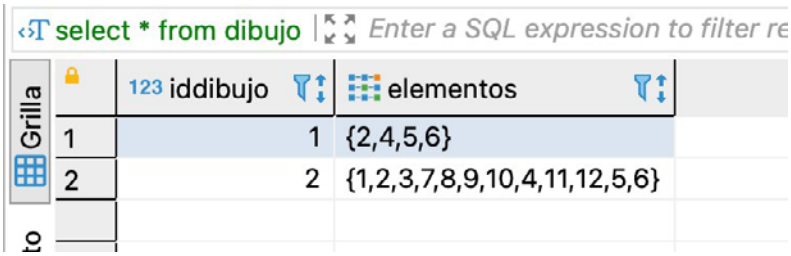
Como es lógico, cabe pensar que al seleccionar datos de la tabla general **`(Select * from Figura)`** aparecerán todos los elementos de las subclases. Si quisiéramos seleccionar solo las que son Figura, podríamos hacerlo con **`(Select * from ONLY Figura)`**.
Vamos a completar el ejemplo creando un dibujo con todas las figuras que tenemos almacenadas. El dibujo lo almacenaremos en una nueva clase que contiene el identificador del dibujo y nos guardaremos una colección con los identificadores de las figuras que forman el dibujo.

```
create table Dibujo(
 idDibujo serial primary key,
 elementos int[]
);

insert into Dibujo (elementos) values(ARRAY[2,4,5,6]);
insert into Dibujo (elementos) values(ARRAY (select fid from figura));
```

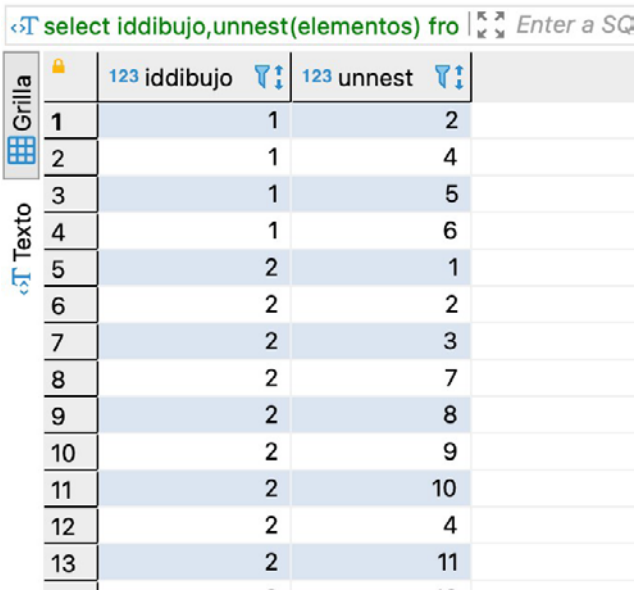
Hay que comentar que la selección de los identificadores de Figura puede ser **directa**, o bien seleccionando aquellas que deseemos, mediante una consulta embebida dentro del constructor de **`ARRAY`**. Esto puede realizarse cuando la **`select`** devuelve una sola columna.

Si queremos obtener los dibujos que hemos insertado **`(select * from dibujo)`**, se nos muestran los elementos como una colección de figuras (un **array**).



Podemos deconstruir el vector, para así poder acceder a cada una de las figuras que lo componen, mediante la función **`unnest`**.

```
select iddibujo,unnest(elementos) from dibujo ;
```



Usamos JDBC (Java Database Connectivity) para ejecutar consultas y realizar operaciones CRUD (Crear, Leer, Actualizar, Borrar) en la base de datos. A continuación, proporcionaré un ejemplo de cómo insertar filas en las tablas Cuadrado, Circulo y Rectangulo utilizando JDBC.

Primero, asumiremos que hemos configurado una conexión JDBC a nuestra base de datos PostgreSQL. Luego, podemos ejecutar consultas SQL utilizando esta conexión.

Aquí hay un ejemplo de inserción de filas en las tablas Cuadrado, Circulo y Rectangulo:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class Main {
```

```
    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost:5432/nombre_bd";
        String usuario = "usuario";
        String contraseña = "contraseña";

          try (Connection conexion = DriverManager.getConnection(url,
usuario, contraseña)) {
            // Inserción en la tabla Cuadrado
              String insertCuadrado = "INSERT INTO Cuadrado (posicion,
color, lado) VALUES (ROW(10, 10), ARRAY['#00BBCC','#BBCC00'], 40)";
                        try (PreparedStatement statement = conexion.
prepareStatement(insertCuadrado)) {
                statement.executeUpdate();
            }

            // Inserción en la tabla Circulo
          String insertCirculo = "INSERT INTO Circulo (posicion, color,
radio) VALUES (ROW(30, 25), ARRAY['#BBCC','#CCCC00'], 20)";
                        try (PreparedStatement statement = conexion.
prepareStatement(insertCirculo)) {
                statement.executeUpdate();
            }

            // Inserción en la tabla Rectangulo
           String insertRectangulo = "INSERT INTO Rectangulo (posicion,
color, alto, ancho) VALUES (ROW(10, 5), ARRAY['#00BBCC','#CCCC00'], 20,
50)";
                        try (PreparedStatement statement = conexion.
prepareStatement(insertRectangulo)) {
                statement.executeUpdate();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Explicación paso a paso del proceso:

1. **Establecer la conexión:** Se define la URL de conexión JDBC, que incluye la dirección del servidor de la base de datos, el puerto y el nombre de la base de datos. Además, se proporciona el nombre de usuario y la contraseña para autenticarse en la base de datos.

2. **Establecer la conexión:** Se utiliza el método `DriverManager.getConnection()` para establecer una conexión con la base de datos PostgreSQL utilizando la URL, el nombre de usuario y la contraseña proporcionados.

3. **Inserción en la tabla Cuadrado:** Se define la consulta SQL para insertar una fila en la tabla `Cuadrado`. La consulta incluye valores para los atributos `posicion`, `color` y `lado`. Se utiliza un `PreparedStatement` para ejecutar la consulta, lo que nos permite precompilar la consulta y evitar posibles problemas de seguridad como la inyección SQL.

4. **Inserción en la tabla Circulo:** Se realiza un proceso similar al paso anterior, pero esta vez para la tabla `Circulo`. Se define la consulta SQL para insertar una fila en la tabla `Circulo`, con valores para los atributos `posicion`, `color` y `radio`.

5. **Inserción en la tabla Rectangulo:** Se repite el proceso de inserción, pero esta vez para la tabla `Rectangulo`. Se define la consulta SQL para insertar una fila en la tabla `Rectangulo`, con valores para los atributos `posicion`, `color`, `alto` y `ancho`.

6. **Gestión de excepciones:** Se utiliza un bloque try-catch para capturar y manejar cualquier excepción que pueda ocurrir durante el proceso de conexión a la base de datos o de ejecución de consultas SQL. En caso de una excepción, se imprime el rastreo de la pila para ayudar en la depuración del problema.

03 BSOO nativas. ObjectDB

En este apartado, como **SGBDOO** se ha optado por **ObjectDB**, ya que es muy versátil, es gratuito e incluso permite embeberse dentro de nuestros proyectos Java, lo que proporciona una gran simplicidad para el desarrollo de aplicaciones pequeñas, debido a la eliminación de un servidor.

3.1. Instalación y acceso a ObjectDB

ObjectDB no requiere instalación como tal, ya que todo su código va integrado en una API de acceso a la base de datos, empaquetado en un fichero **JAR**.
Desde la web oficial podemos descargarnos el **ObjectDB Development Kit**. Este kit contiene, entre otros:

- Dependencias para proyectos Java.
- Utilidades para la visualización y consulta de la BD.
- Servidor para aplicaciones distribuidas.
- Documentación.

Una vez descomprimido, solo necesitaremos la máquina virtual de Java instalada en nuestro sistema para ejecutar todos los elementos.
Para utilizar **ObjectDB** en nuestro proyecto, deberemos añadir el archivo **objectdb.jar** a las dependencias de nuestro proyecto, o bien realizarlo mediante el gestor de dependencia de **maven** o **gradle**.
En este momento, ya podemos conectarnos a la base de datos; la centralización de la conexión a la base de datos se realiza mediante una instancia de un objeto **EntityManagerFactory**, a partir del cual podemos obtener varias instancias de un **EntityManager**.
A partir del **EntityManager**, ya podremos realizar las típicas operaciones CRUD, teniendo en cuenta que, siempre que existan modificaciones en esta, deberemos realizar la operación dentro de una transacción para evitar situaciones inconsistentes en ella. Aquí vemos una posible clase con el establecimiento de la conexión y la obtención de un `EntityManager`.

3.2. Creación y persistencia de objetos

A. Persistir clases

Para **persistir un objeto**, necesitaremos:

- Anotar su clase y marcarla como **@Entity**.
- Definir un campo como identificador **@Id**, y, opcionalmente, que sea autoincremental con **@GeneratedValue**.
- El resto de atributos de la entidad, por defecto, se persisten automáticamente sin ningún tipo de anotación. En caso de no querer persistir alguno, podemos indicarlo con **@transient**.

```
@Entity
public class Alumno {
@Id @GeneratedValue
private Long id;
private String nombre;
}
```

Para almacenar un Alumno, bastará con crear un Alumno y persistirlo en la base de datos, como se ve a continuación, suponiendo el objeto con de tipo **conexionODB** visto en el fichero adjunto.

```
EntityManager em= con.getEM();
em.getTransaction().begin(); Alumno alu=new
Alumno("Antonio Ramos"); em.persist(alu);
em.getTransaction().commit();
```

B. Clases embebidas o componentes

En Java existen en ocasiones **clases que no tienen existencia propia**, a menos que existan dentro de otra clase, como, por ejemplo, una clase Dirección. No tiene sentido crear un objeto Dirección ***ad hoc***; en cambio, sí que tiene sentido crearlo de manera que exista una Dirección, por ejemplo, dentro de un Alumno.
Estas clases (débiles) que existen embebidas dentro de otras clases debemos declararlas como **embebibles o incrustadas** mediante la anotación **@Embeddable** y marcarlas como embebidas (**@Embedded**) dentro de la clase en que existen.

```
@Embeddable
public class Direccion {
        ...
}
```

```
@Entity
public class Alumno {
    ...
    @Embedded
    private Direccion direccion;
}
```

```
// en main
Alumno alu= new Alumno("Joan Gerard");
Direccion d= new Direccion("C/ del calvario");
alu.setDireccion(d);
em.persist(alu);
```

En la base de datos se almacenará una entidad Alumno, pero la dirección no existe como objeto en sí mismo.

3.3. Relaciones

A. Uno a uno

La más sencilla es la 1-1, en la cual un objeto contiene a otro objeto. Lo marcaremos, como ya hacíamos en Hibernate, con el modificador **@OneToOne**, indicando que el guardado sea en cascada **(cascade=CascadeType.PERSIST)**.
A partir de este momento, al guardar una instancia de un objeto, se guardará una instancia propia del objeto relacionado y se enlazará. El objeto enlazado tendrá existencia en sí mismo (en caso de estar marcado como **@Entity**).
Un ejemplo básico es que una clase tiene un único Tutor; basándonos en el caso en que una Clase (de un instituto) tiene asociado un Tutor, el ejemplo será el siguiente:

```
@Entity
public class Profesor  {
    ...
}
```

```
@Entity
public class Clase{
    ...
    @OneToOne(cascade=CascadeType.PERSIST)
    private Profesor elTutor;
}
```

```
Profesor p=new Profesor("Pepe");
Clase c= new Clase("2DAM");
c.setTutor(p);
em.persist(c);  //al guardar la clase se guarda el tutor
```

B. Uno a muchos

Vamos a referirnos ahora a una **relación clásica**, en la que un Profesor es el tutor de varios Alumnos. Estas relaciones pueden ser unidireccionales o bidireccionales. En este ejemplo, la veremos bidireccional, de manera que, dado un Alumno, podemos saber quién es su Profesor tutor, y, dado un Profesor, podemos obtener los alumnos que tutoriza:

```
@Entity
public class Alumno {
            ...
    @ManyToOne(cascade=CascadeType.PERSIST)
    private Profesor elTutor;
}
@Entity
public class Profesor  {
          ...
    @OneToMany(cascade=CascadeType.PERSIST,fetch=FetchType.EAGER)
    private List<Alumno> losAlumnos;
}
```

Hay que fijarse en que es muy similar a Hibernate, tanto en el **CascadeType** como en el **FetchType**. Los usos son iguales que en **Hibernate**; **Eager** es el modo de carga anticipada o «ansiosa», y **Lazy**, la carga en diferido o «vaga».

C. Muchos a muchos

Las relaciones muchos a muchos podemos enfocarlas de varios modos. Pongamos el ejemplo de la docencia entre Profesores y Alumnos. Si simplemente queremos **indicar quién da clase a quién**, bastaría guardar una colección de profesores en cada alumno (los profesores que le dan clase a dicho alumno), y, de manera simétrica, en cada profesor una colección de alumnos (los alumnos a los que imparte clase).
En este caso, sería una **bidireccional**, ya que desde una clase podemos navegar hasta la otra, lo que queda de este modo:

```
@Entity
public class Alumno {
            ...
    @ManyToMany(cascade=CascadeType.PERSIST, fetch=FetchType.LAZY)
    private Set<Profesor> profesores=new HashSet<>();
}
```

```
@Entity
public class Profesor {
        ...
    @ManyToMany(cascade=CascadeType.PERSIST, fetch=FetchType.LAZY)
    private Set<Alumno> losAlumnos=new HashSet<>();;
}
```

Si necesitamos **guardar dentro de dicha relación alguna información**, como, por ejemplo, las notas que ha recibido el alumno, o las incidencias puestas, entonces debemos crear una nueva clase, que incorporará los atributos propios de la relación, y realizar relaciones 1 a M desde cada entidad (Alumno/Profesor) hacia la nueva entidad (Docencia). Este supuesto es el famoso «Las relaciones N-M generan tabla con los atributos que poseen» del modelo relacional.

3.4. Consultas

Vamos a revisar cómo podemos **cargar los datos que hemos guardado previamente en la base de datos**. Supongamos que tenemos una clase Alumno, mapeada con entidad y con identificador `(idAlumno)`. La manera más sencilla de cargar un Alumno, conocido su ID, es el método `find(class,id)`:

```
Alumno a=em.find(Alumno.class, 2);
```

Busca en aquellas entidades de dicha clase la que tiene dicho identificador.

3.5. Queries y TypedQueries

El resto de cargas debemos realizarlas mediante **consultas**, en un lenguaje **JPQL *(Java persistance query language)***, que, de nuevo, es similar al HQL de Hibernate. Lo que cambian son los tipos de datos para montar dichas consultas.

IMPORTANTE

Existen dos clases **Query** y **TipedQuery** (la segunda hereda de la primera), que habitualmente se usan, en el primer caso, cuando desconocemos el resultado de la consulta, y en el segundo, cuando sabemos el resultado. La primera es polimórfica; por tanto, hará un enlace dinámico de resultados, y la segunda verifica el resultado con la clase actual. En la documentación oficial se recomienda el uso de la segunda, **TypedQuery**, para consultas. El **Query** lo utilizaremos, sin embargo, para actualizaciones y borrados.

CREACIÓN DEL QUERY (Q) O DEL TYPEDQUERY (TQ)

`Object q.getSingleResult();` `T tq.getSingleResult();`	Queries que devuelven un solo objeto. Genérico o concreto.
`List q.getResultList();` `List<T> tq.getResultList();`	Queries con múltiples resultados. List genérico o List concreto.
`q.executeUpdate();`	Para borrados y actualizaciones.

EJEMPLO

```
TypedQuery<Alumno> tq=em.createQuery("Select a from Alumno a where
a.ampa=true", Alumno.class);
List<Alumno> alumnosAmpa=tq.getResultList();
```

Para evitar consultas ***hard-coded***, podemos parametrizarlas de manera nominal:

```
TypedQuery<Alumno> tq=em.createQuery("Select a from Alumno a where
a.ampa= :ampa", Alumno.class);
tq.setParameter("ampa", true);
List<Alumno> alumnosAmpa=tq.getResultList();
```

Podemos también reaprovechar las consultas, creando consultas etiquetadas. Se define una consulta y se etiqueta. Posteriormente, se puede invocar indicando la etiqueta y la clase. Pueden contener parámetros.

```
@NamedQueries({
     @NamedQuery(query = "Select a from Alumno a where a.nombre =
:name", name = "find alu by name"),
     @NamedQuery(query = "Select a from Alumno a", name = "find all
alu")
})
…
TypedQuery<Alumno>  tq=em.createNamedQuery("get  all  alu",Alumno.
class);
List<Alumno> alumnosAmpa=tq.getResultList();
```

3.6. Borrado y actualización

Para terminar, vamos a revisar las operaciones CRUD que nos quedan. **Las actualizaciones son totalmente transparentes al usuario**, ya que cualquier modificación que se realice dentro del contexto de una transacción será automáticamente guardada al cerrarla con un `commit()`. Además, pueden realizarse `Queries` de actualización.

IMPORTANTE

Para los borrados, si el objeto ha sido recuperado de la base de datos, y por tanto está en la transacción actual, puede eliminarse con `em.remove(objeto)`. Se borrará de la base de datos al realizarse el `commit`. Si dicho objeto está referenciado por alguna relación, deberá revisarse el `CascadeType`, para no producir eliminaciones en cascada indeseadas.

UAX FP