



ACCESO A DATOS

2º CFGS D.A.M.



UD. 3

BASES DE DATOS ORIENTADAS A OBJETO y NOSQL



Bases de Datos Objeto-Relacionales

El término Base de Datos Objeto Relacional (BDOR) se usa para describir una base de datos que ha evolucionado desde el modelo relacional hacia otra más amplia que incorpora conceptos del paradigma orientado a objetos. Por tanto, un Sistema de Gestión Objeto-Relacional (SGBDOR) contiene ambas tecnologías: relacional y de objetos.

Una idea básica de las BDOR es que el usuario pueda crear sus propios tipos de datos, para ser utilizados en aquella tecnología que permita la implementación de tipos de datos predefinidos. Además, las BDOR permiten crear métodos para esos tipos de datos. Con ello, este tipo de SGBD hace posible la creación de funciones miembro usando tipos de datos definidos por el usuario, lo que proporciona flexibilidad y seguridad.

Los SGBDOR permiten importantes mejoras en muchos aspectos con respecto a las BDR tradicionales. Estos sistemas gestionan tipos de datos complejos con un esfuerzo mínimo y albergan parte de la aplicación en el servidor de base de datos. Permiten almacenar datos complejos de una aplicación dentro de la BDOR sin necesidad de forzar los tipos de datos tradicionales. Son compatibles en sentido ascendente con las bases de datos relacionales tradicionales, tan familiares a multitud de usuarios. Es decir, se pueden pasar las aplicaciones sobre bases de datos relacionales al nuevo modelo sin tener que reescribirlas. Adicionalmente, se pueden ir adaptando las aplicaciones y bases de datos para que utilicen las funciones orientadas a objetos.

Las características más importantes de los SGBDOR son:

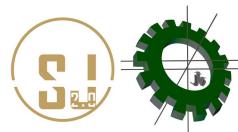
- Soporte de tipos de datos básicos y complejos. El usuario puede crear sus propios tipos de datos
- Soporte para crear métodos para esos tipos de datos. Se pueden crear funciones miembro usando tipos de datos definidos por el usuario.
- Gestión de tipos de datos complejos con un esfuerzo mínimo.
- Herencia
- Se pueden almacenar múltiples valores en una columna de una misma fila.
- Relaciones (tablas) anidadas
- Compatibilidad con las bases de datos existentes tradicionales. Es decir, se pueden pasar las aplicaciones sobre bases de datos relacionales al nuevo modelo sin tener que reescribirlas.

El inconveniente de las BDOR es que aumenta la complejidad del sistema, esto ocasiona un aumento del coste asociado.

Bases de Datos Orientadas a Objetos

Las BDOO (Bases de Datos Orientada a Objetos) son aquellas cuyo modelo de datos está orientado a objetos, soportan el paradigma orientado a objetos almacenando métodos y datos. Su origen se debe principalmente a la existencia de problemas para representar cierta información y modelar ciertos aspectos del mundo real. Las BDOO simplifican la programación orientada a objetos (POO) almacenando directamente los objetos en la BD y empleando las mismas estructuras y relaciones que los lenguajes POO.





Podemos decir que un Sistema Gestor de Base de Datos Orientada a Objetos (SGBDOO) es un sistema gestor de base de datos que almacena objetos.

Las bases de datos Orientada a Objetos (BDOO) posee las siguientes características:

- Almacena los datos como objetos
- Cada objeto se identifica mediante un identificador único u OID (Object Identifier), este identificador no es modificable por el usuario.
- Cada objeto define sus métodos y atributos y la interfaz mediante la cual se puede acceder a él, el usuario puede especificar qué atributos y métodos pueden ser usados desde fuera.
- Un SGBDOO debe cumplir las características de un SGBD, como son persistencia, concurrencia, recuperación ante fallos, gestión del almacenamiento secundario y facilidad de consultas; y las características de un sistema orientado a objetos (OO): encapsulación, identidad, herencia y polimorfismo.

Como ventajas podemos reseñar:

- Mayor capacidad de modelado. La utilización de objetos permite representar de una forma más natural los datos que se necesitan guardar.
- Extensibilidad. Se pueden construir nuevos tipos de datos a partir de tipos existentes.
- Existe una única interfaz entre el lenguaje de manipulación de datos y el de programación. Esto elimina el tener que incrustar un lenguaje declarativo como SQL en un lenguaje imperativo como Java o C.
- Lenguaje de consultas más expresivo. Éste lenguaje es navegacional de un objeto al siguiente, en contraste con el SQL.
- Soporte a transacciones largas, necesario para muchas aplicaciones de bases de datos avanzadas.
- Adecuación a aplicaciones avanzadas de bases de datos (CASE, CAD, sistemas multimedia, etc).

Como desventajas podemos reseñar:

- Falta de un modelo de datos universal.
- Falta de experiencia, siendo su uso todavía relativamente limitado
- Falta de estándares, no existe un lenguaje de consultas estándar como SQL, aunque existe el OQL de ODMG que se está convirtiendo en un estándar de facto.
- Competencia con los SGBDR y los SGBDOR.
- La optimización de consultas compromete la encapsulación
- Complejidad: el incremento de funcionalidad provisto por un SGBDOO lo hace más complejo que un SGBDR.
- Falta de soporte a las vistas: la mayoría de SGBDOO no proveen mecanismos de vistas.
- Falta de soporte a la seguridad.





El estándar ODMG

ODMG es un grupo de fabricantes de bases de datos con el objetivo de definir estándares para los SGBDOO. Uno de los estándares recibe el mismo nombre del grupo y especifica los elementos que se definirán y la manera en qué se hará , para la consecución de persistencia en las BDOO que soporten el estándar.

El estándar propone los siguientes componentes:

- Modelo de objetos.
- Lenguaje de definición de objetos (ODL, Object Definition Language)
- Lenguaje de consulta de objetos (OQL, Object Query Language)
- Conexión con los lenguajes C++, Smalltalk y Java.

El modelo de objetos especifica las características de los objetos, cómo se relacionan y cómo se identifican, construcciones soportadas, etc. Las primitivas de modelado básicas son: los objetos caracterizados por un identificador único (OID) y los literales que son objetos que no tienen identificador, no pueden aparecer como objetos, están embebidos en ellos.

Los tipos de objetos son:

- Atómicos: boolean, short, long, float, doublé, char, String, enum, octect, unsigned long, unsigned short.
- Tipos estructurados: date, time, timestamp, interval.
- Colecciones <interfaceCollection>:
 - set<tipo>: grupo desordenado de objetos del mismo tipo que no admite duplicados.
 - bag<tipo>: grupo desordenado de objetos del mismo tipo que admite duplicados.
 - list<tipo>: grupo ordenado de objetos del mismo tipo del mismo tipo que admite duplicados.
 - array<tipo>: grupo ordenado de objetos del mismo tipo a los que se puede acceder por su posición. Tamaño dinámico
 - dictionary<clave, valor>: grupo de objetos del mismo tipo, cada valor está asociado a su clave.

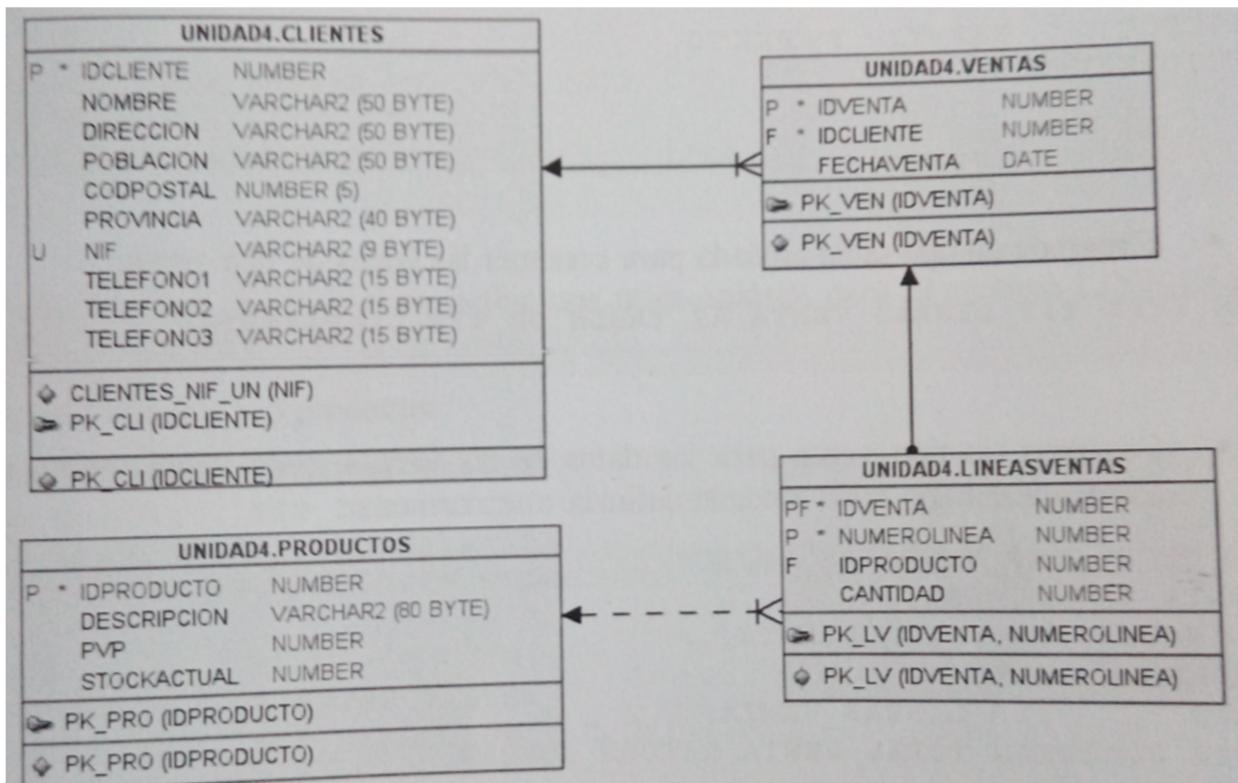
Mediante clases especificamos el estado y el comportamiento de un tipo de objeto, pueden incluir métodos. Son instanciables, por lo que a partir de ellas se pueden crear instancias de objetos individuales.

El lenguaje ODL es el equivalente al lenguaje de definición de datos (DDL) de los SGBD tradicionales. Define los atributos, las relaciones entre los tipos y especifica la firma de las operaciones. La sintaxis de ODL extiende el lenguaje de definición de interfaces. Algunas de las palabras reservadas para definir los objetos son:

- class: declaración del objeto, define el comportamiento y el estado de un tipo de objeto.
- extent: define la extensión, nombre para el actual conjunto e objetos de la clase. En las consultas se hace referencia al nombre definido en esta cláusula, no se hace referencia al nombre definido a la derecha de class.
- key[s]: declara la lista de claves para identificar las instancias.

- attribute: declara un atributo.
- set | list | bag | array: declara un tipo de colección.
- struct: declara un tipo estructurado
- enum: declara un tipo enumerado
- relationship: declara una relación.
- inverse: declara una relación inversa.
- extends: define la herencia simple.

Ejemplo de creación del siguiente modelo de datos:



```

class Cliente (extent Clientes key NIF)
{
    /*Definición de atributos*/
    attribute struct Nombre_Persona {
        string apellidos,
        string nombre} nombre;
    attribute string NIF;
    attribute date fecha_nac;
    attribute enum Genero{H,M} sexo;
    attribute struct Direccion {
        string calle,
        string poblac} direc;
    attribute set<string> telefonos;
    /*Definición de operaciones*/
    short edad();
}

```



```
class Producto (extent Productos key IDPRODUCTO)
{
    /*Definición de atributos*/
    attribute short IDPRODUCTO;
    attribute string descripción;
    attribute float pvp;
    attribute short stockactual;
}
```

```
class LineaVenta (extent Lineaventas)
{
    /*Definición de atributos*/
    attribute short numerolinea;
    attribute Producto product;
    attribute short cantidad;

    /*Definición de operaciones*/
    float importe();
}
```

```
class Venta (extent Ventas key IDVENTA)
{
    /*Definición de atributos*/
    attribute short IDVENTA;
    attribute date fecha_venta;
    attribute set <LineaVenta> lineas;

    /*Definición de relaciones*/
    relationship Cliente pertenece_a_cliente inverse
        Cliente::tiene_venta

    /*Definición de operaciones*/
    float total_venta();
}
```

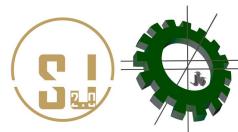
El lenguaje de consulta OQL

El Object Query Language es el lenguaje estándar de consultas de BDOO y posee las siguientes características:

- Es orientado a objetos y está basado en el modelo ODMG.
- Es un lenguaje declarativo del tipo SQL, con una sintaxis muy similar.
- Acceso declarativo a los objetos de la base de datos (propiedades y métodos).
- Semántica formal bien definida.
- No incluye operaciones de actualización (sólo de consultas). Las modificaciones se realizan mediante los métodos que los objetos poseen.
- Dispone de operadores sobre colecciones (max, min, count, avg, ...) y cuantificadores (for all, exists).

Posee la siguiente sintaxis:





```
SELECT <lista de valores>
FROM <lista de colecciones y miembros típicos> [WHERE <condición>]
```

Donde las colecciones en FROM pueden ser extensiones (los nombres que aparecen a la derecha de extent) o expresiones que evalúan una colección. Se suele utilizar una variable iterador que vaya tomando valores de los objetos de la colección. Las variables iterador se pueden especificar de varias formas utilizando las cláusulas IN o AS:

```
FROM Clientes x
FROM x IN Clientes
FROM Clientes AS x
```

Para acceder a los atributos y objetos relacionados se utilizan expresiones de camino, que empieza normalmente con un nombre de objeto o una variable iterador seguida de atributos conectados mediante un punto o nombre de relaciones. Es decir, si queremos obtener el nombre de los clientes que son mujeres, podemos escribir:

```
SELECT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo="M"
SELECT x.nombre.nombreper FROM Clientes x WHERE x.sexo="M"
SELECT x.nombre.nombreper FROM Clientes AS x WHERE x.sexo="M"
```

Otras expresiones son:

- v.IDVENTA es el identificador de venta del objeto v.
- v.fecha_venta es la fecha de venta del objeto v.
- v.total_venta() obtiene el total venta del objeto v.
- v.pertenece_a_cliente es un puntero al cliente mencionado en v.
- v.pertenece_a_cliente.direc es la dirección del cliente mencionado en v.
- v.lineas es una colección de objetos del tipo LineaVenta
- El uso de v.lineas.numerolinea NO es correcto porque v.lineas es una colección de objetos y no un objeto simple.
- Cuando tenemos una colección como en v.lineas, para poder acceder a los atributos de la colección podemos usar la orden FROM.

Por ejemplo para obtener los datos del cliente cuyo IDVENTA=1 podríamos realizar la siguiente consulta:

```
SELECT v.pertenece_a_cliente.nombre, v.pertenece_a_cliente.direc, v.fecha_venta, v.total_venta()
FROM Ventas v WHERE v.IDVENTA=1;
```

O si queremos obtener las líneas de veenta del IDVENTA=1, podemos usar el objeto v para definir la segunda colección v.lineas:

```
SELECT lin.numerolinea, lin.product.descripcion,lin.cantidad,lin.importe()
FROM Ventas v, v.lineas lin WHERE v.IDVENTA=1;
```

El resultado de una consulta OQL puede ser de cualquier tipo soportado por el modelo. Por ejemplo, la consulta anterior devuelve un conjunto de estructuras del tipo short, string, y float; el resultado es del tipo colección: bag (struct(numerolinea:short,descripción:string,cantidad:short,importe:float)).



Pero si la consulta es: `SELECT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo="M"`, devuelve un conjunto de nombres; el tipo devuelto es `bag(string)`.

En las consultas podemos usar alias. Por ejemplo, la consulta anteriormente vista podemos usar los siguientes alias:

```
SELECT nlin:lin.numerolinea,  
       dpro:lin.product.descripcion,  
       can:lin.cantidad,  
       imp:lin.importe()
```

Y el tipo devuelto en este caso: `bag(struct(nlin:short, dpro:string, can:short,
imp:float))`

Para obtener un set de estructuras (que no admite duplicados) podemos usar `DISTINCT` a continuación de `SELECT`:

```
SELECT DISTINCT x.nombre.nombreper FROM x IN Clientes WHERE x.sexo="M"  
(devuelve set(string))  
SELECT nlin:lin.numerolinea,  
       dpro:lin.product.descripcion,  
       can:lin.cantidad,  
       imp:lin.importe()  
  
FROM Ventas v, v.lineas lin WHERE v.IDVENTA=1 ORDER BY nlin ASC; (devuelve un  
list(struct(nlin:short,dpro:string,can:short,imp:float)))
```

Como en cualquier lenguaje, podemos hacer uso de operadores para comparar los valores (`=, >, >=, <, <=, i=, IN, LIKE, _ o ?, * o %`)

Además, mediante el uso de cuantificadores podemos comprobar si todos los miembros, al menos un miembro o algunos miembros, etc. satisfacen la condición:

Todos los miembros	<code>FOR ALL x IN colección:condición</code>
Al menos uno:	<code>EXISTS x IN colección:condición</code> <code>EXISTS x</code>
Solo uno:	<code>UNIQUE x</code>
Algunos/cualquier:	<code>Colección comparación SOME/ANY condición</code> <i>Donde comparación puede ser: <, >, <=, >=, =</i>

```
SELECT v.IDVENTA, v.fecha_venta, v.total_venta() FROM Ventas v
```

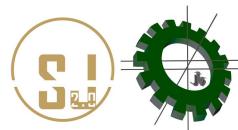
```
WHERE EXISTS x IN  
v.lineas:x.product.descripcion="PNY Pendrive 16 GB";
```

Obtiene todas las ventas que tengan líneas de venta cuya descripción del producto sea "PNY Pendrive 16 GB"

```
SELECT v.IDVENTA, v.fecha_venta, v.total_venta() FROM Ventas v
```

```
WHERE FOR ALL x IN  
v.lineas:x.product.descripcion="PNY Pendrive 16 GB";
```

Obtiene las ventas que sólo tienen líneas de venta cuya descripción del producto sea "PNY Pendrive 16 GB"



Los operadores AVG, SUM, MIN, MAX, COUNT, se pueden aplicar a cualquier colección, siempre y cuando tengan sentido para el tipo de elemento. Por ejemplo para calcular la media del total venta de todas las ventas necesitaríamos asignar el valor devuelto a una variable.

Media = AVG(SELECT v.total_venta() FROM Ventas v)	devuelve
bag(struct(total_venta:float)	

Ejemplo de Base de Datos Orientada a Objetos

Vamos a ver una base de datos sencilla de utilizar que posee una API simple que no requiere técnicas de mapeo. Dicha base de datos es NeoDatis Object Database. Esta base de datos es de código abierto que funciona con Java, .Net, y Android, además de Groovy. Los objetos se pueden almacenar y recuperar con una sola línea de código, sin necesidad de tener que mapearlos a tablas.

Para ello, nos descargamos la última versión desde <https://sourceforge.net/projects/neodatis-odb/>. El fichero neodatis-odb-1.9.30.689.jar lo añadiremos a nuestro proyecto.

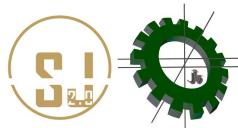
```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;

//Clase Jugadores
class Jugadores {
    private String nombre;
    private String deporte;
    private String ciudad;
    private int edad;

    // constructores
    public Jugadores() {
    }

    public Jugadores(String nombre, String deporte, String ciudad, int edad) {
        this.nombre = nombre;
        this.deporte = deporte;
        this.ciudad = ciudad;
        this.edad = edad;
    }

    // getters y setters
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getDeporte() {
        return deporte;
    }
    public void setDeporte(String deporte) {
        this.deporte = deporte;
    }
}
```



```
public String getCiudad() {
    return ciudad;
}
public void setCiudad(String ciudad) {
    this.ciudad = ciudad;
}
public int getEdad() {
    return edad;
}
public void setEdad(int edad) {
    this.edad = edad;
}

public class EjemploNeodatis {

    public static void main(String[] args) {

        // Crear instancias para almacenar en BD
        Jugadores j1 = new Jugadores("Maria", "voleibol", "Madrid", 14);
        Jugadores j2 = new Jugadores("Miguel", "tenis", "Madrid", 15);
        Jugadores j3 = new Jugadores("Mario", "baloncesto", "Guadalajara",
        15);
        Jugadores j4 = new Jugadores("Alicia", "tenis", "Madrid", 14);

        ODB odb = ODBFactory.open("neodatis.test"); // Abrir BD

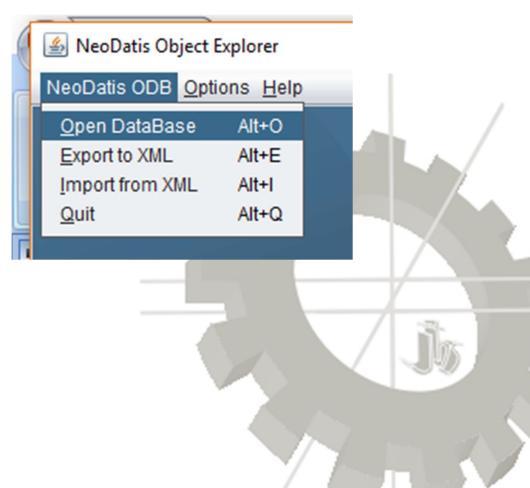
        // Almacenamos objetos
        odb.store(j1);
        odb.store(j2);
        odb.store(j3);
        odb.store(j4);

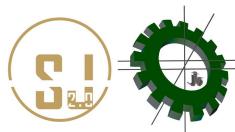
        // recuperamos todos los objetos
        Objects<Jugadores> objects = odb.getObjects(Jugadores.class);
        System.out.printf("%d Jugadores: %n", objects.size());

        int i = 1;
        // visualizar los objetos
        while (objects.hasNext()) {
            Jugadores jug = objects.next();
            System.out.printf("%d: %s, %s, %s %n", i++, jug.getNombre(),
                jug.getDeporte(), jug.getCiudad(), jug.getEdad());
        }
        odb.close(); // Cerrar BD
    }
}
```

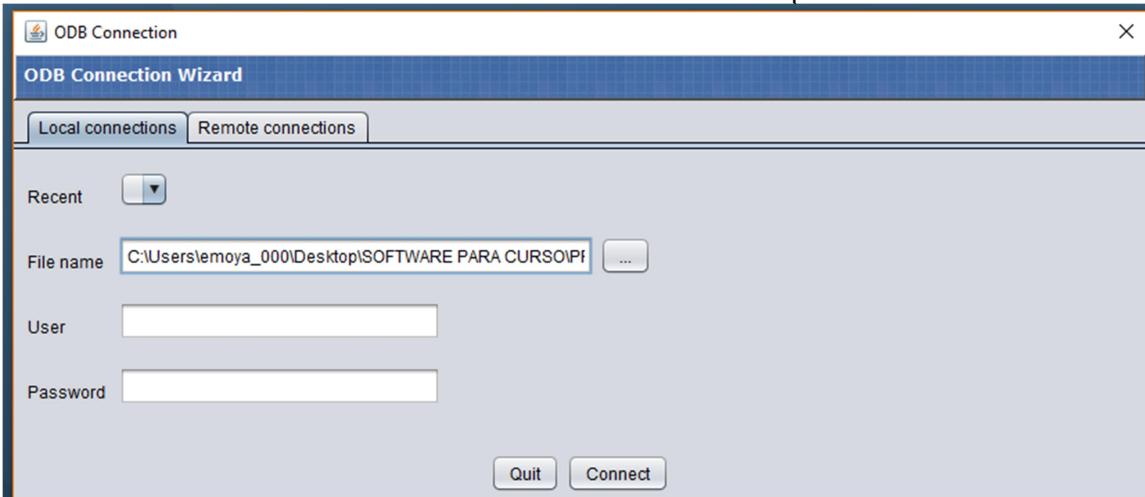
NeoDatis posee un explorador que nos permite navegar a través de los objetos. Para ejecutarlo hacemos un doble clic en el fichero odb-explorer.bat (sistemas Windows) u odb-explorer.sh (Linux). Es necesario abrir la base de datos por la que vamos a navegar, pulsamos en el menú NeoDatis ODB->Open Database

Y en Local connections, localizamos el fichero neodatis.test, pulsando el botón Connect a continuación.

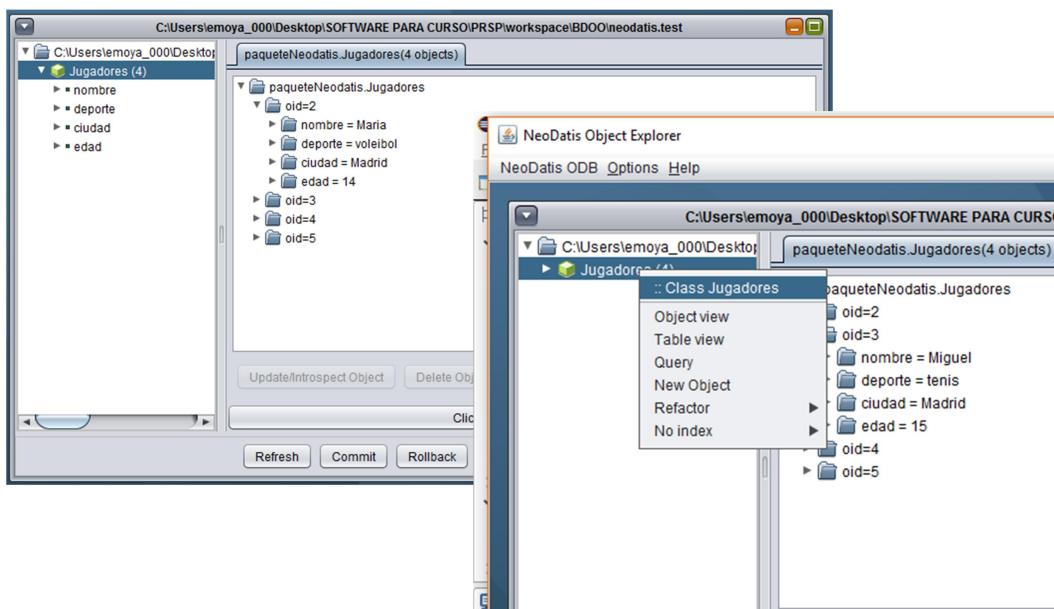




Normalmente se debe encontrar en el workspace de nuestro IDE.



Desde el navegador, podemos realizar consultas, modificaciones, eliminaciones, inserciones y todo ello validado con el commit.



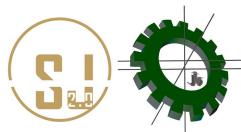
Podemos acceder a los datos mediante su OID (Object Identifier) implementando mediante java su código correspondiente:

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.OID;
import org.neodatis.odb.core.oid.OIDFactory;

public class ejemploOID {

    public static void main(String[] args) {
        ODB odb = ODBFactory.open("neodatis.test"); //Abro la BD
        OID oid = OIDFactory.buildObjectOID(3); //Obtengo el objeto con el OID=3

        Jugadores jug = (Jugadores) odb.getObjectFromId(oid);
```



```
System.out.println(jug.getNombre()+"*"+jug.getDeporte()+"*"+jug.getCiudad()+"*"+jug.getEdad());
    odb.close(); //Cierro la BD
}
}
```

Podemos realizar consultas sencillas, usando la clase CriteriaQuery, donde especificaremos la clase y el criterio para realizar la consulta. Para ello necesitaremos una serie de paquetes a importar.

```
import org.neodatis.odb.ODB;
import org.neodatis.odb.ODBFactory;
import org.neodatis.odb.Objects;
import org.neodatis.odb.core.query.IQuery;
import org.neodatis.odb.core.query.criteria.Where;
import org.neodatis.odb.impl.core.query.criteria.CriteriaQuery;

public class EjemploConsulta {

    public static void main(String[] args) {

        ODB odb = ODBFactory.open("neodatis.test"); // Abrir BD

        IQuery query = new CriteriaQuery(Jugadores.class, Where.equal("deporte", "tenis"));
        query.orderByAsc("nombre,edad"); //ordena ascendemente por nombre y edad

        // recuperamos todos los objetos
        Objects<Jugadores> objects = odb.getObjects(query);
        System.out.printf("%d Jugadores: %n", objects.size());

        int i = 1;
        // visualizar los objetos
        while (objects.hasNext()) {
            Jugadores jug = objects.next();
            System.out.printf("%d: %s, %s, %s %n", i++, jug.getNombre(),
                jug.getDeporte(), jug.getCiudad(), jug.getEdad());
        }
        odb.close(); // Cerrar BD
    }
}
```

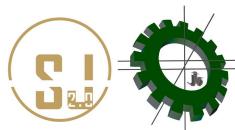
La modificación de un objeto, es muy similar a todo lo visto hasta ahora. Por ello, deberemos cargar el objeto modificarlo usando los métodos setter del objeto y a continuación lo actualizaremos con el método store(). No olvidemos que para validar los cambios es necesario cerrar la BD.

```
IQuery query = new CriteriaQuery(Jugadores.class, Where.equal("nombre", "Maria"));
Objects<Jugadores> objects = odb.getObjects(query);
//Obtiene sólo el primer objeto encontrado.
Jugadores jug = (Jugadores) objects.getFirst();

jug.setDeporte("voley-playa"); //Cambia el deporte
odb.store(jug); //actualiza el objeto.
```

La eliminación es idéntica en procedimiento a lo anterior, ya que sustituimos el store por el delete y no es necesario usar el método setter correspondiente.





```
odb.delete(jug);
```

También podemos usar una interfaz para construir el criterio de la consulta. Esta interfaz es la ICriterion y debemos importar el paquete correspondiente.

```
import org.neodatis.odb.core.query.criteria.ICriterion;
```

Con este interfaz podemos crear consultas del tipo: jugadores que tengan 14 años, mayores de 14 años o cuyo nombre comiencen por M.

```
ICriterion criterio = Where.equal("edad", 14);
CriteriaQuery query = new CriteriaQuery(Jugadores.class, criterio);

// recuperamos todos los objetos
Objects<Jugadores> objects = odb.getObjects(query);
```

```
ICriterion criterio = Where.like("nombre", "M%");
```

```
ICriterion criterio = Where.gt("edad", 14);
```

También podemos usar Where.ge para Mayor o igual, lt para menor y le para menor o igual.

Para comprobar si un array o una colección contiene un valor determinado usamos Where.contains

Y si es un valor nulo o no nulo Where.isNull Where.isNotNull();

La construcción de expresiones lógicas también es posible, pero debemos importar paquetes.

```
import org.neodatis.odb.core.query.criteria.And;
import org.neodatis.odb.core.query.criteria.Or;
import org.neodatis.odb.core.query.criteria.Not;
```

Un ejemplo podría ser:

```
ICriterion criterio = new And().add(Where.equal("ciudad", "Madrid"))
.add(Where.equal("edad", 15));
```

De igual forma, en el ejemplo anterior podemos utilizar el Or() y el not.

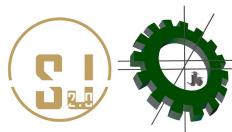
```
ICriterion criterio = Where.not(Where.like("nombre", "M%"));
```

O combinarlo, en dos criterios:

```
ICriterion criterio1 = Where.like("nombre", "M%");
ICriterion criterio=Where.not(criterio1);
```

Se pueden realizar consultas más complejas, agrupándolas, y usando funciones SUM, MAX, MIN, AVG, COUNT... y para ello deberemos usar la API Object Values de Neodatis, pero este apartado te lo dejo para ti si quieras profundizar más en el tema de las bases de datos Orientada a Objetos.





Bases de Datos NoSQL

Las bases de datos NoSQL son aquellas que no siguen el modelo clásico del sistema de gestión de bases de datos relacionales. Se caracterizan porque no usan el SQL como lenguaje principal de consultas, y además, en el almacenamiento de los datos no se utilizan estructuras fijas de almacenamiento.

Este término surge a raíz de la web 2.0, ya que hasta ese momento sólo subían contenidos a la red aquellas empresas que tenían un portal, pero con la llegada de las redes sociales en las que el usuario interactúa en la web, cualquier usuario podía subir contenido, provocando así el crecimiento exponencial de los datos.

El problema que surge a raíz de ello es cómo gestionar y acceder a toda esa información almacenada en bases de datos relacionales. La solución de poner más máquinas era costosa y no solucionaba el problema. Otra solución fue crear nuevos sistemas gestores de datos pensados para un uso específico, que con el paso del tiempo han dado paso a la filosofía NoSQL.

NoSQL son estructuras que nos permiten almacenar información en aquellas situaciones en las que las bases de datos relacionales generan ciertos problemas (de escalabilidad, rendimiento, concurrencia,...). Estas bases de datos intentan resolver problemas de almacenamiento masivo, alto desempeño, procesamiento masivo de transacciones, persistencia, etc.

El principio ACID

Este tipo de transacciones, enfocan su interés en la fiabilidad de las transacciones bajo el conocido principio ACID (**A**tómica **C**onsistente **I**slada – **D**urabilidad en inglés- y **D**urable)

- Atomicidad: Asegurar de que la transacción se complete o no, sin quedarse a medias ante fallos.
- Consistencia: Asegurar el estado de validez de los datos en todo momento
- Aislada –Isolation–: Asegurar la independencia entre transacciones
- Durabilidad: Asegurar la persistencia de la transacción ante cualquier fallo.

Este principio, aporta robustez pero colisiona con el rendimiento y operatividad a medida que los volúmenes de datos crecen.

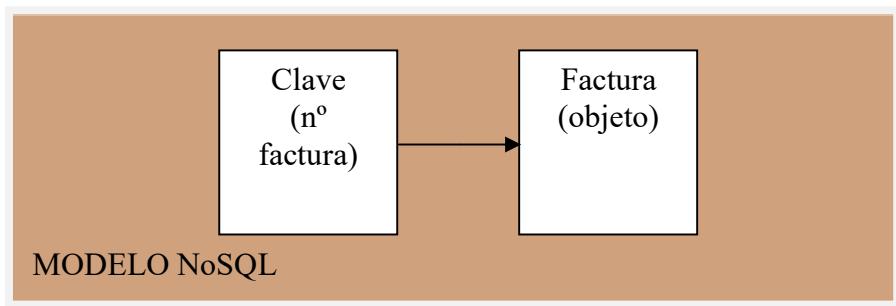
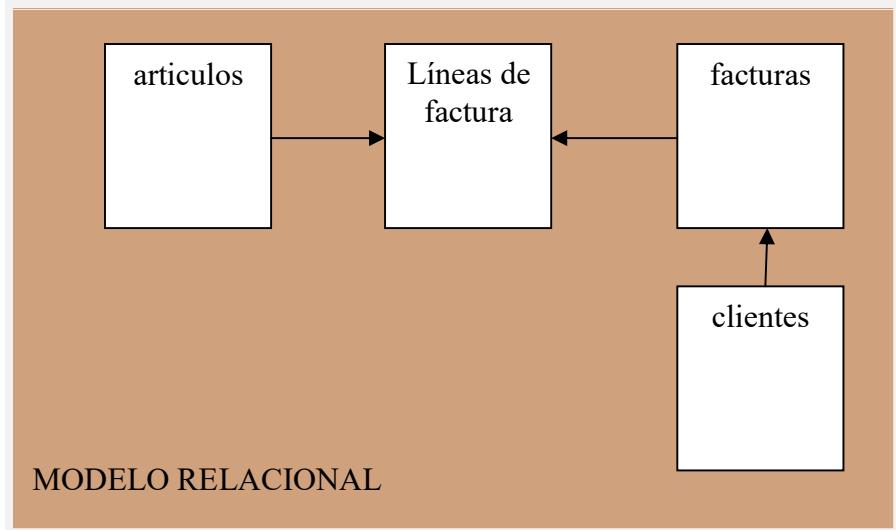
Cuando la magnitud y el dinamismo de datos cobran importancia, este principio queda en segundo plano en los modelos relacionales frente al rendimiento, disponibilidad y escalabilidad.

Otro principio es el conocido como BASE (**B**asic **A**vailability **S**oft **S**tate **E**ventually **c**onsistency). Disponibilidad como prioridad, la consistencia de datos se delega a gestión externa al motor de la base de datos e intentar lograr la convergencia hacia un estado consistente (se asume que inconsistencias temporales progresen a un estado final estable). Es este principio al que se ajustan la mayoría de sistemas de datos en internet.



Ventajas de los sistemas NoSQL

La gran diferencia de estas bases de datos es cómo almacenan los datos.



El tipo de almacenamiento de información, ofrece ciertas ventajas sobre los modelos relacionales como son:

- Se ejecutan en máquinas con pocos recursos: estos sistemas no requieren mucha programación, por lo que se pueden instalar en máquinas de un coste más reducido.
- Escalabilidad horizontal: para mejorar el rendimiento de estos sistemas simplemente se consigue añadiendo más nodos, con la única operación de indicar al sistema cuáles son los nodos que están disponibles.
- Pueden manejar gran cantidad de datos: esto es debido a que utiliza una estructura distribuida, en muchos casos mediante tablas Hash.
- No generan cuellos de botella: el principal problema de los sistemas SQL es que necesitan transcribir cada sentencia para poder ser ejecutada, y cada sentencia compleja requiere, además, de un nivel de ejecución aún más complejo, lo que constituye un punto de entrada en común, que ante muchas peticiones ralentiza el sistema.

Diferencia con los sistemas SQL

Ya podemos deducir algunas diferencias con los sistemas SQL, pero a continuación, las detallamos:





- No utilizan SQL como lenguaje de consulta. Aunque pueden usarlo como lenguaje de apoyo, evitan el uso de los mismos.
- No utilizan estructuras fijas como tablas de almacenamiento de datos. Permiten hacer uso de otros tipos de modelos como sistemas clave-valor (hash), objetos o grafos.
- No suelen permitir operaciones JOIN. Al disponer de un volumen de datos tan extremadamente grande suele resultar deseable evitar los JOIN. Esto se debe a que cuando la operación no es la búsqueda de una clave, la sobrecarga puede llegar a ser muy costosa. Las soluciones más directas consisten en desnormalizar los datos, o bien, realizar el JOIN mediante software en la capa de aplicación.
- Arquitectura distribuida. Las bases de datos relacionales suelen estar centralizadas, en NoSQL la información puede estar compartida en varias máquinas mediante mecanismos de tablas hash distribuidas.

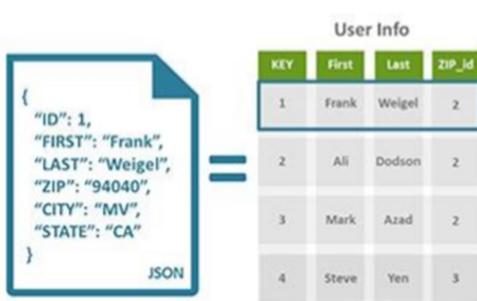
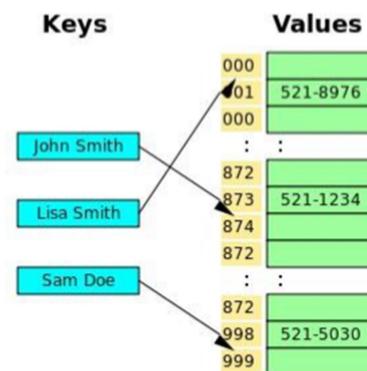
Tipos de Bases de Datos NoSQL

Según el tipo o modelo escogido para almacenar los datos, las bases de datos NoSQL se agrupan en cuatro categoría principales:

1.- Bases de datos clave – valor

Son el modelo de base de datos NoSQL más popular, además de ser la más sencilla en cuanto a funcionalidad. En este tipo de sistema, cada elemento está identificado por una llave única, lo que permite la recuperación de la información de forma muy rápida, información que habitualmente está almacenada como un objeto binario (BLOB). Se caracterizan por ser muy eficientes tanto para las lecturas como para las escrituras.

Algunos ejemplos de este tipo son Cassandra, BigTable o HBase.



2.- Bases de datos documentales

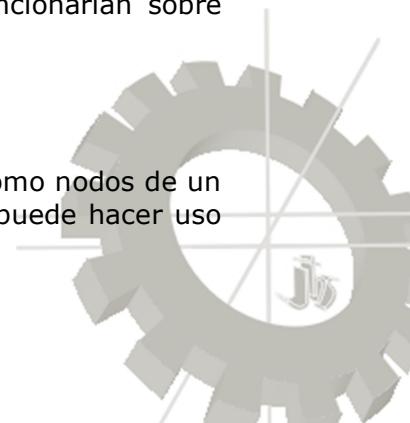
Este tipo almacena la información como un documento, generalmente utilizando para ello una estructura simple como JSON o XML y donde se utiliza una clave única para cada registro. Este tipo de implementación permite, además de realizar búsquedas por clave-valor, realizar consultas más avanzadas sobre el contenido del documento.

Son las bases de datos NoSQL más versátiles. Se pueden utilizar en gran cantidad de proyectos, incluyendo muchos que tradicionalmente funcionarían sobre bases de datos relacionales.

Algunos ejemplos de este tipo son **MongoDB** o **CouchDB**.

3.- Bases de datos en grafo

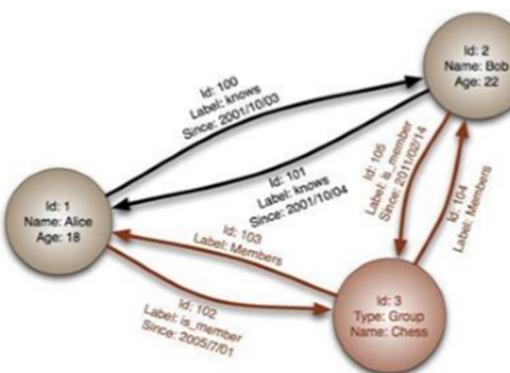
En este tipo de bases de datos, la información se representa como nodos de un grafo y sus relaciones con las aristas del mismo, de manera que se puede hacer uso



de la teoría de grafos para recorrerla. Para sacar el máximo rendimiento a este tipo de bases de datos, su estructura debe estar totalmente normalizada, de forma que cada tabla tenga una sola columna y cada relación dos.

Este tipo de bases de datos ofrece una navegación más eficiente entre relaciones que en un modelo relacional.

Algunos ejemplos de este tipo son Neo4j, InfoGrid o Virtuoso.



4.- Bases de datos en columnas

Parecido al modelo clave/valor pero la clave se basa en una combinación de columna, fila y marca de tiempo que se utiliza para referenciar conjuntos de columnas (familias). Es la implementación más parecida a bases de datos relacionales. Ejemplos: Cassandra, BigTable, Hadoop/HBase. Compañías como Twitter o Adobe usan este modelo.

MongoDB

MongoDB es un gestor de datos **NoSQL distribuido de tipo documental** que almacena documentos en un formato similar a JSON (para ser más exactos internamente usa BSON). Está escrita en C++ y es **multi-plataforma, Open Source y gratuito**.

El proyecto nació a finales del año 2007 como un proyecto interno de una empresa llamada **10Gen** para usarlo en una aplicación de Internet que estaban desarrollando, pero en 2009 decidieron liberarlo como Open Source y dedicarse íntegramente a él, ofreciendo soporte comercial y servicios relacionados.

Su nombre proviene de la palabra en inglés **Humongous**, que significa literalmente "algo realmente grande", y se refiere a su capacidad de gestionar cantidades enormes de datos.

Sus principales características son:

- Basado en el motor V8 de Google Chrome para JavaScript. Facilidad de aprendizaje por basarse en este lenguaje.
- Almacenamiento flexible basado en JSON sin necesidad de definir esquemas previamente.
- Soporte para creación de índices a partir de cualquier atributo.
- Alto rendimiento para consultas y actualizaciones.
- Consultas flexibles basadas en documentos.
- Alta capacidad de crecimiento, replicación y escalabilidad: puedes escalar horizontalmente simplemente añadiendo máquinas baratas sin ver afectado el rendimiento ni complicar la gestión.
- Soporte para almacenamiento independiente de archivos de cualquier tamaño basado en GridFS.

<http://www.campusmvp.es/recursos/post/Fundamentos-de-bases-de-datos-NoSQL-MongoDB.aspx>



Podemos ver la analogía entre las terminologías relacionales y de documento de MongoDB:

MODELO RELACIONAL	MONGODB
Base de datos	Base de datos
Tabla	Colección
Fila	Documento
Columna	Campo
Índice	Índice
Join	Documento embebido o referencia

Con el modelo de MongoDB se pasa de un modelo de datos rígido basado en estructuras de datos bidimensionales, formado por tablas, filas y columnas a un modelo de datos de documentos dinámicos con subdocumentos y matrices embebidas. Con MongoDB podemos crear colecciones sin definir su estructura también se puede alterar la estructura de los documentos simplemente añadiendo nuevos campos o borrando los ya existentes.

MongoDB almacena documentos JSON (JavaScript Object Notation) en una representación binaria llamada BSON (Binary JSON). BSON es una serialización codificada en binario de documentos JSON, comporta todas las características de JSON e incluye los tipos de datos int, long, float o arrays. El documento (registro en modelo relacional) representa la unidad básica de datos en MongoDB.

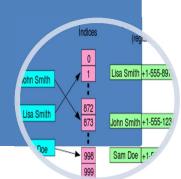
Estructuras JSON

JSON es un formato ligero de intercambio de datos. Leerlo y escribirlo es simple para humanos, mientras que para las máquinas es simple interpretarlo y generararlo. Está basado en un subconjunto del Lenguaje de Programación JavaScript. JSON es un formato de texto que es completamente independiente del lenguaje, pero utiliza convenciones que son conocidas por los programadores de la familia de lenguajes C, incluyendo, C++, C#, Java, JavaScript, Perl, Python,... Para más información <http://www.json.org/jsones.html>.

JSON está formado por dos tipos de estructuras:

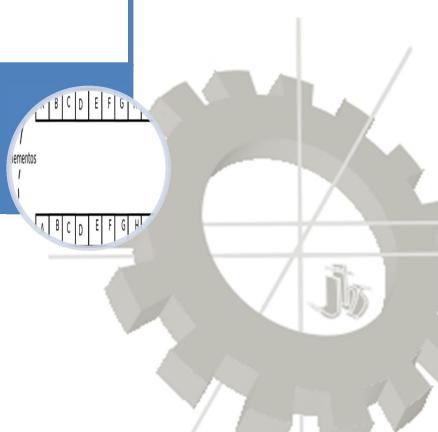
- objeto, registro, estructura, diccionario, tabla hash, lista de claves o array asociativo

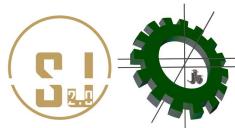
Colección de pares de nombre/valor



- arrays, vectores, listas o secuencias

Lista ordenada de valores





Estas dos estructuras son universales, siendo soportado por todos los lenguajes. En JSON, se presentan de dos formas:

- Como Objeto, conjunto desordenado de pares nombre/valor. Un objeto comienza con llave de apertura { y termina con llave de cierre }. Cada nombre es seguido por dos puntos : y los pares nombre/valor están separados por coma ,

```
{ "persona": {"nombre": "Alicia", "oficio": "Profesora"} }  
{ "zona": {"codzona": 10, "nombre": "Madrid"} }
```

Objeto persona con atributos nombre y profesora y objeto zona con codzona y nombre como atributos

- Un array, es decir, una colección de valores. Un array comienza con corchete de inicio [y termina con corchete de cierre]. Los valores se separan por coma ,

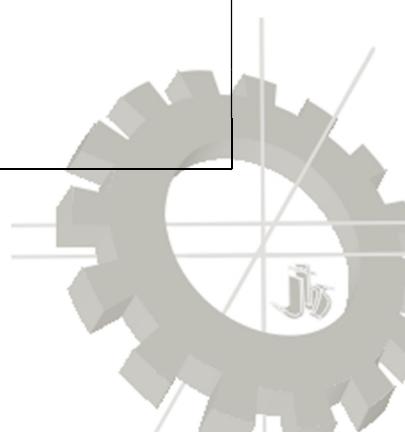
```
{ "persona": [  
    {"nombre": "Alicia", "oficio": "Profesora", "ciudad": "Talavera"},  
    {"nombre": "María Jesús", "oficio": "Profesora"}  
]
```

```
{ "zona": [  
    {"codzona": 10, "nombre": "Madrid"},  
    {"codzona": 20, "nombre": "Málaga", "tasa": 15}  
]
```

Los objetos no tiene por qué tener los mismos pares/valor

- Un valor puede ser una cadena de caracteres con comillas dobles, un número, true o false o null, un objeto o un array. Estas estructuras pueden anidarse. En el ejemplo se muestra un objeto de nombre ventana con distintos tipos de nombre/valor.

```
{ "ventana": {  
    "titulo": "Gestión Artículos",  
    "alto": 300,  
    "ancho": 500,  
    "menú": null,  
    "modal": true,  
    "botones": ["ok", "cancel"]  
}
```





- Una cadena de caracteres es una colección de cero o más caracteres Unicode, encerrados entre comillas dobles, usando barras divisorias invertidas como escape. Un carácter es una cadena de un único elemento.
- Un número es similar a un número C o Java, excepto que no se usan los formatos octales y hexadecimales.

Los espacios en blanco pueden insertarse entre cualquier par de símbolos nombre/valor.

XML Vs. JSON

```
<?xml version="1.0" encoding="UTF-8"?>
<departamentos>
    <TITULO>DATOS DE LA TABLA DEPART</TITULO>
    <DEP_ROW>
        <DEPT_NO>10</DEPT_NO>
        <DNOMBRE>CONTABILIDAD</DNOMBRE>
        <LOC>SEVILLA</LOC>
    </DEP_ROW>
    <DEP_ROW>
        <DEPT_NO>20</DEPT_NO>
        <DNOMBRE>INVESTIGACION</DNOMBRE>
        <LOC>MADRID</LOC>
    </DEP_ROW>
    <DEP_ROW>
        <DEPT_NO>30</DEPT_NO>
        <DNOMBRE>VENTAS</DNOMBRE>
        <LOC>MALAGA</LOC>
    </DEP_ROW>
    <DEP_ROW>
        <DEPT_NO>40</DEPT_NO>
        <DNOMBRE>PRODUCCIÓN</DNOMBRE>
        <LOC>BILBAO</LOC>
    </DEP_ROW>
</departamentos>
```

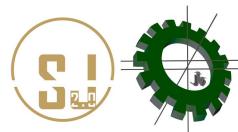
XML

```
{"departamentos": [
    {"TITULO": "DATOS DE LA TABLA DEPART",
     "DEP_ROW": [{"DEPT_NO": 10, "DNOMBRE": "CONTABILIDAD", "LOC": "SEVILLA"}, {"DEPT_NO": 20, "DNOMBRE": "INVESTIGACION", "LOC": "MADRID"}, {"DEPT_NO": 30, "DNOMBRE": "VENTAS", "LOC": "MALAGA"}, {"DEPT_NO": 40, "DNOMBRE": "PRODUCCIÓN", "LOC": "BILBAO"}]}
]}
```

JSON

También podemos representarlo como un array, ya que DEP_ROW se repite:





{"departamentos":

 {"TITULO":"DATOS DE LA TABLA DEPART",

 "DEP_ROW":

 [

 {"DEPT_NO":10,"DNOMBRE":"CONTABILIDAD","LOC":"SEVILLA"},

 {"DEPT_NO":20,"DNOMBRE":"INVESTIGACION","LOC":"MADRID"},

 {"DEPT_NO":30,"DNOMBRE":"VENTAS","LOC":"MALAGA"},

 {"DEPT_NO":40,"DNOMBRE":"PRODUCCION","LOC":"BILBAO"}

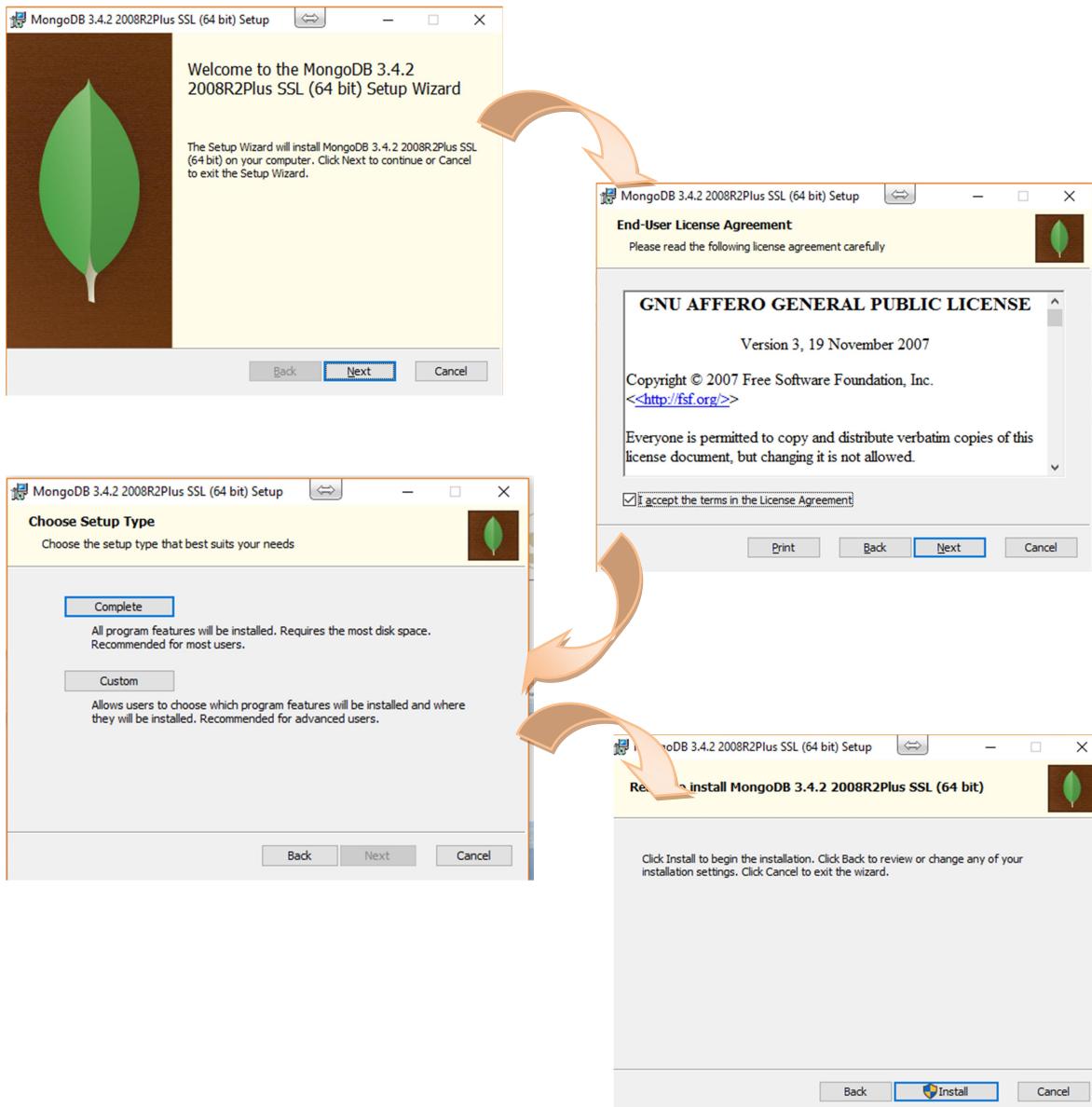
]

}

JSON

Instalación MongoDB

Nos descargamos la versión de MongoDB de <https://www.mongodb.com/download>.





En la carpeta C:\Program Files\MongoDB\Server\3.4\bin se encuentra el fichero mongod.exe que arranca la base de datos. En la unidad C:\ deberemos crear la carpeta C:\data\db que por defecto MongoDB va a utilizar para almacenar la información.

Al arrancar la base de datos, veremos la siguiente pantalla:

Ahora ejecutamos el programa mongo.exe para conectarnos como cliente a la base de datos.

Por defecto, nos conectamos a la base de datos test.

```
C:\Program Files\MongoDB\Server\3.4\bin\mongo.exe - X
MongoDB shell version v3.4.2
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.2
Server has startup warnings:
2017-02-04T23:12:06.202+0100 I CONTROL [initandlisten]
2017-02-04T23:12:06.202+0100 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2017-02-04T23:12:06.203+0100 I CONTROL [initandlisten] ** Read and write access to data and configuration is unrestricted.
2017-02-04T23:12:06.204+0100 I CONTROL [initandlisten]
2017-02-04T23:12:08.174+0100 W FTDC [initandlisten] Failed to initialize Performance Counters for FTDC: WindowsPdhError: PdhExpandCounterPathW failed with 'El objeto especificado no se encontró en el equipo.' for counter '\Memory\Available Bytes'
2017-02-04T23:12:08.175+0100 I FTDC [initandlisten] Initializing full-time diagnostic data capture with directory 'C:/data/db/diagnostic.data'
2017-02-04T23:12:08.486+0100 I INDEX [initandlisten] build index on: admin.system.version properties: { v: 2, key: { version: 1 }, name: "incompatible_with_version_32", ns: "admin.system.version" }
2017-02-04T23:12:08.487+0100 I INDEX [initandlisten] building index using bulk method; build may temporarily use up to 500 megabytes of RAM
2017-02-04T23:12:08.499+0100 I INDEX [initandlisten] build index done. scanned 0 total records. 0 secs
2017-02-04T23:12:08.501+0100 I COMMAND [initandlisten] setting featureCompatibilityVersion to 3.4
2017-02-04T23:12:08.503+0100 I NETWORK [thread1] waiting for connections on port 27017
```

```
C:\Program Files\MongoDB\Server\3.4\bin\mongod.exe - X
statistics.log=(wait=0),
2017-02-04T23:12:06.202+0100 I CONTROL [initandlisten]
2017-02-04T23:12:06.202+0100 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2017-02-04T23:12:06.203+0100 I CONTROL [initandlisten] ** Read and write access to data and configuration is unrestricted.
2017-02-04T23:12:06.204+0100 I CONTROL [initandlisten]
2017-02-04T23:12:08.174+0100 W FTDC [initandlisten] Failed to initialize Performance Counters for FTDC: WindowsPdhError: PdhExpandCounterPathW failed with 'El objeto especificado no se encontró en el equipo.' for counter '\Memory\Available Bytes'
2017-02-04T23:12:08.175+0100 I FTDC [initandlisten] Initializing full-time diagnostic data capture with directory 'C:/data/db/diagnostic.data'
2017-02-04T23:12:08.486+0100 I INDEX [initandlisten] build index on: admin.system.version properties: { v: 2, key: { version: 1 }, name: "incompatible_with_version_32", ns: "admin.system.version" }
2017-02-04T23:12:08.487+0100 I INDEX [initandlisten] building index using bulk method; build may temporarily use up to 500 megabytes of RAM
2017-02-04T23:12:08.499+0100 I INDEX [initandlisten] build index done. scanned 0 total records. 0 secs
2017-02-04T23:12:08.501+0100 I COMMAND [initandlisten] setting featureCompatibilityVersion to 3.4
2017-02-04T23:12:08.503+0100 I NETWORK [thread1] waiting for connections on port 27017
```

Operaciones básicas con MongoDB

Todos los comandos irán en minúsculas, siendo los más comunes:

- **show databases**-Muestra las bases de datos disponibles

- **db**-Muestra la base de datos actual
- **show collections**-Muestra las colecciones de la base de datos actual
- **use nombrebasededatos**-Usa una base de datos. Si no existe la creará en el momento que añadamos un objeto JSON, con las funciones .save o .insert
- **db.nombre-colección.count()**-Indica el número de documentos dentro de las colecciones. También se utilizan .size() y .length().
- **// comentarios al igual que Java**

Crear registros

Para añadir datos a la base de datos utilizaremos los comandos **.save** o **.insert** según el formato:

```
db.nombre_colección.save(dato JSON)
db.nombre_colección.insert(dato JSON)
```

db es la base de datos actual, la que estamos usando (y abierto con use) y nombre de colección es la colección donde se van a añadir los registros, si no existe se crea en ese momento.

Vamos a crear la base de datos *mibasedatos*, y dentro de ella la colección *amigos* con dos amigos:





```
use mibasedatos;

Amigo1={nombre:'Ana',telefono:1111111111,curso:'1DAM', nota:7};

Amigo2={nombre:'Juan',telefono:2222222222,curso:'1DAM', nota:8};

db.amigos.save(Amigo1);

db.amigos.save(Amigo2);

db.amigos.insert({nombre:'Juanito',teléfono:3333333333,curso:'2DAM',nota:6});
```

Identificador De Objetos. El ObjectId(campo _id)

Los identificadores de cada documento (registro) son únicos. Se asignan automáticamente al crear el documento, se generan de forma rápida y ordenada. También se pueden crear de forma manual. El ObjectId o _id, es única en la colección y no se repetirá. Si un documento no tiene _id, MongoDB lo asignará automáticamente. Es lo que ocurre cuando insertamos y no indicamos el identificador.

Consulta de registros

La orden .find() sirve para consultar datos de una colección. Para ellos escribiremos db.nombre_colección.find().

En nuestro ejemplo anterior db.amigos.find() nos mostrará los _id de cada objeto JSON, únicos por colección, con el resto de campos.

Si lo que queremos es mostrar ordenadamente los resultados por nombre db.amigos.find().sort({nombre:1}); o -1 si lo queremos de forma descendente.

Para consultas más complejas como búsquedas en documentos de una o varias condiciones, usamos: db.nombre_colección.find(filtro,campos)

- En filtro indicamos la condición de búsqueda, podemos añadir los pares nombre:valor a buscar.
- En campos se especifican los campos a devolver de los documentos que coinciden con el filtro de consulta. Si queremos devolver todos los campos de los documentos omitimos este parámetro.

db.amigos.find({nombre:"Juanito"})	Buscamos el amigo con nombre Juanito
db.amigos.find({nombre:"Juanito"},{teléfono:1})	Buscamos el teléfono de Juanito
db.amigos.find({curso:"1DAM"},{nombre:1, nota:1})	Buscamos el nombre y la nota de los alumnos de 1º DAM
db.amigos.find({curso:"1DAM"}).count()	Deseamos saber el número de registros que devuelve la consulta.

Selectores de búsquedas de comparación

\$eq, igual a un valor	db.amigos.find({nota:{\$eq:6}})
\$gt, mayor que, \$gte, mayor o igual	db.amigos.find({nota:{\$gt:6}})
\$lt, menor que, \$lte, menor o igual	db.amigos.find({nota:{\$gte:6, \$lte:9}})



\$ne, distinto	db.amigos.find({nota:{\$ne:7}})
\$in, entre una lista de valores y	db.amigos.find({nota:{\$in:[5,7,8]}},{nombre:1})
\$nin, no está en la lista de valores	db.amigos.find({nota:{\$nin:[5,7,8]}})

Selectores de búsquedas lógicos

\$or,	db.amigos.find({\$or:[{nota:{\$gt:7}},{curso:"1DAM"}]})
\$and	Este operador es implícito en las consultas. No hace falta especificarlo
\$not	db.amigos.find({nota:{\$not:{\$ne:7}}})
\$exists, filtra la búsqueda tomando en cuenta la existencia del campo de la expresión.	db.amigos.find({nota:{\$exists:true}})

Actualizar registros

Para actualizar datos utilizaremos el comando .update, siguiendo el formato siguiente:

```
db.nombre_colección.update(  
    filtro_búsqueda,  
    cambios_a_realizar,  
    {  
        upsert: booleano,  
        multi: booleano  
    });
```

Filtro_búsqueda, es la condición para localizar los registros o documentos a modificar.

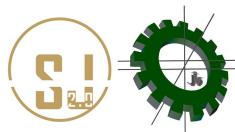
Cambios_a_realizar, son los cambios que se desean hacer. Hay que tener cuidado al utilizar esta orden, pues en cambios_a_realizar se indica cómo quedará el documento que se busca si este existe, es decir: el resultado final del documento es lo que se escriba en cambios_a_realizar.

Si no se escriben todos los campos que tenía el documento, estos no los incluye en la modificación, entonces, los elimina. Por ejemplo:

```
db.amigos.update({nombre:"Ana"},{nombre:"Ana María"});  
db.amigos.update({nombre:"Marleni"},{nombre:"Marleni",teléfono:952666555});
```

Nos encontramos con dos tipos de cambios: cambiar el documento completo por otro que indiquemos, o modificar solo los campos especificados, para ello utilizamos los parámetros upsert y multi, ambos opcionales y su valor por defecto es false:

- upsert – si asignamos true a este parámetro, se indica que si el filtro de búsqueda no encuentra ningún resultado, entonces, el cambio debe ser insertado como un nuevo registro
- multi – en caso de que el filtro de búsqueda devuelva más de un resultado, si especificamos este parámetros a true, el cambio se



realizará a todos los resultados, de lo contrario sólo se cambiará al primero que encuentre, es decir, al que tenga menor identificativo de objeto, `_id`.

```
db.amigos.update({nombre:"Pepita"},  
                  {nombre:"Pepita",teléfono:1457825},{upsert:true});
```

Modifico el teléfono de Pepita y si no existe... la inserta

Update cuenta con operadores que permiten realizar actualizaciones más complejas, que las vemos a continuación:

Operadores de modificación

\$set	Actualiza con nuevas propiedades a un documento (o conjunto) <code>db.amigos.update({nombre:"Pepita"}, {\$set:{edad:24}})</code>
\$unset	Permite eliminar propiedades de un documento <code>db.amigos.update({nombre:"Pepita"}, {\$unset:{edad:24}})</code>
\$inc	Incrementa una cantidad numérica especificada en el valor del campo a incrementar. <code>db.amigos.update({nombre:"Pepita"}, {\$inc:{edad:1}})</code>
\$rename	Renombra campos del documento. <code>db.amigos.update({nombre:'Pepita'}, {\$unset:{edad:'age',nombre:'name'}})</code>

```
db.amigos.update({curso:"1DAM"},{$inc:{nota:1}})  
db.amigos.update({curso:"1DAM"},{$inc:{nota:1}}, {multi:true})
```

En el primer ejemplo incremento nota en uno -sólo el primer registro encontrado-. En el segundo ejemplo incremento la nota a todos los encontrados.

```
db.amigos.update({curso:"1DAM"},{$set:{poblacion:"Talavera"}}, {multi:true})
```

Añado la población a todos los alumnos de 1º DAM

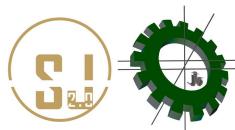
Operaciones con Arrays

Vamos a crear la colección de libros con tres de ellos y un array con temas del libro:

```
db.libros.insert({código:1,nombre:"Acceso a datos",pvp:35,editorial:"Garceta",temas:["Base de datos","Hibernate","Neodatis"]})  
db.libros.insert({código:2,nombre:"Entornos desarrollo",pvp:27,editorial:"RAMA",temas:["UML","Subversion","ERMaster"]})  
db.libros.insert({código:3,nombre:"Programación servicios",pvp:25,editorial:"Garceta",temas:["Socket","Multihilo"]})
```

Para consultar los elementos del array escribimos el array y el elemento a consultar.

```
db.libros.find({temas:"UML"})  
db.libros.find({$or:[{temas:"UML"},{temas:"Neodatis"}]})  
db.libros.find({editorial:"Garceta",pvp:{$gt:25},$or:[{temas:"UML"},{temas:"Neodatis"}]})
```



Operaciones de modificación para arrays

\$push	Añade un elemento al array. <code>db.libros.update({código:1},{\$push:{temas:"MongoDB"})}</code>
\$addToSet	Agrega elementos a un array sólo si estos no existen. <code>db.libros.update({temas:{\$exists:true}},{\$addToSet:{temas:"Base de datos"}},{multi:true})</code>
\$each	Se usa en conjunto con los dos anteriores para indicar que se añaden varios elementos al array. <code>db.libros.update({código:1},{\$push:{temas:{\$each:["JSON","XML"]}}})</code>
\$pop	Elimina el primer o último valor de un array. Con valor -1 borra el primero con otro valor el último. <code>db.libros.update({código:3},{\$pop:{temas:-1}})</code>
\$pull	Elimina valores de un array que cumplan con el filtro indicado <code>db.libros.update({},{\$pull:{temas:{\$in:["Base de datos","JSON"]}}},{multi:true})</code>

Borrar registros

Para borrar datos JSON podemos utilizar las órdenes **.remove** y **.drop**. Se puede eliminar los documentos que cumplan una condición, o todos los documentos de la colección o la colección completa.

```
db.amigos.remove({nombre:"Marleni"});  
db.amigos.remove({nombre:"Ana",teléfono:1111111111});  
db.amigos.remove({}); //borra todos los elementos de la colección.  
db.amigos.drop(); //borra la colección
```

Funciones de agregado

Como otras bases de datos, disponemos de funciones matemáticas, de fecha y de cadenas para utilizarlas en las consultas.

Se insta al alumno a informarse e investigar sobre ellas.

Relaciones entre documentos en MongoDB

MongoDB utiliza 2 métodos o patrones que nos van a permitir establecer la estructura de los documentos y sus relaciones. Vamos a ver cómo son las relaciones entre documentos comparándolas con el modelo relacional. Los métodos son utilizar referencias:

- Referencias Manuales: Guardamos el campo `_id` de un documento como referencia en otro documento. Similar al concepto de clave ajena del modelo relacional. En este método la aplicación debe ejecutar una segunda consulta para devolver los datos relacionados. Este es el método más utilizado.

```
db.emple.insert({_id:'emp1',nombre:"Juan",salario:1000,fechaalta"10/10/1999"})  
db.emple.insert({_id:'emp2',nombre:"Alicia",salario:1400,fechaalta"07/08/2000",oficio:"Profesora"})
```



```
db.emple.insert({_id:'emp3',nombre:"María Jesús",salario:1500,fechaalta:"05/01/2005",oficio:"Analista",comisión:100})  
db.emple.insert({_id:'emp4',nombre:"Alberto",salario:1100,fechaalta:"15/11/2001"})
```

```
db.depart.insert({_id:'dep1',nombre:"Informática",loc:'Madrid',emple:[emp1,emp2]})  
db.depart.insert({_id:'dep2',nombre:"Gestión",loc:'Talavera',emple:[emp3,emp4]})  
db.depart.find()
```

Para visualizar los datos de la combinación de las colecciones necesitaremos hacer dos consultas, una para obtener el departamento a consultar, y la otra para obtener los empleados de ese departamento, que están dentro del array del departamento. Por ejemplo, se desea visualizar los empleados del departamento con identificativo _id igual a dep1.

```
Departrabajo = db.depart.findOne({_id:'dep1'})
```

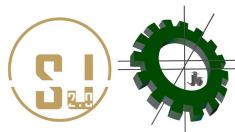
Con el método anterior, cargamos el departamento con _id:dep1 en una variable.

```
emplesdep=db.emple.find({_id:{$in:departrabajo.emple}})
```

Con el método anterior recuperamos los empleados cuyo _id se encuentre enlazado a este departamento.

```
emplesdep=db.emple.find({_id:{$in:departrabajo.emple}}).toArray()
```

- DBRefs son referencias de un documento a otro utilizando el valor del campo _id del primer documento, el nombre de la colección y opcionalmente el nombre de base de datos. Esta opción no es muy usada.



Ejercicios:

1. Crea un nuevo proyecto Java, añadiendo el JAR para trabajar con **Neodatis**. Dentro del proyecto crea un paquete de nombre **clases**. Crea la clase **Paises** con dos atributos y sus getter y setters. Los atributos son **private int id; private String nombrepais;**

Añade el método **toString()** para que devuelva el nombre del país: **public String toString() {return nombrepais;}**

Crea la clase **Jugadores** (en el paquete **clases**, como en el ejemplo del tema) y añade el siguiente atributo con sus getter y setter: **private Paises país;**

Crea una clase Java (con el método **main()**) que cree una base de datos de nombre **EQUIPOS.DB** e inserte países y los jugadores de esos países. Añade otra clase Java para visualizar los países y los jugadores que hay en la BD.

2. Añade al proyecto anterior otra clase java para realizar una consulta en la que para cada jugador de un deporte, visualice también su país.

3. Añade al proyecto anterior clases para eliminar y dar de alta objetos, así como modificar atributos. Comprueba en el explorador de neodatis los cambios realizados.

4. Crea la colección empleados de la base de datos mibasedatos y añade los siguientes registros utilizando **MongoDB**

Emp_no:1,nombre:"Juan",dep:10,salario:1000,fechaalta:"10/10/1999"

Emp_no:2,nombre:"Alicia",dep:10,salario:1400,fechaalta:"07/08/2000",oficio:"Profesora"

Emp_no:3,nombre:"María

Jesús",dep:20,salario:1500,fechaalta:"05/01/2005",oficio:"Analista",comisión:100

Emp_no:4,nombre:"Alberto",dep:20,salario:1100,fechaalta:"15/11/2001"

Emp_no:5,nombre:"Fernando",dep:30,salario:1400,fechaalta:"20/11/1999",comisión:200,oficio:"Analista

4.1. Visualiza los empleados del departamento 10

4.2. Visualiza los empleados del departamento 10 y 20

4.3. Obtén los empleados con salario >1300 y oficio Profesora

4.4. Sube el salario a los analistas en 100€, a todos los analistas

4.5. Decrementa la comisión en 20€ (escribir -20) sólo a los que tengan comisión.

5. Utilizando la colección libros (vista en la página 25) realiza las siguientes consultas:

5.1. Visualiza los libros de la editorial Garceta, con PVP entre 20 y 25 incluidos y que tengan el tema **SOCKET**

5.2. Agrega el tema **socket** a los libros que no lo tengan.

5.3. Baja a 5 el precio de los libros de la editorial Garceta.