UNIDAD 10

APLICACIONES JAVA CON BD OO y NoSQL

BIBLIOGRAFÍA:

- Manual y tutoriales de DBO4 8.0.
- Manual y documentación de Mongo.

11.1. INTRODUCCIÓN A LAS BDOO.

Muchas aplicaciones Java necesitan tener datos persistentes. En la mayoría de los casos, esto significa usar una base de datos relacional. El API JDBC y los drivers para la mayoría de los SGBD proporcionan una forma estándar de utilizar SQL para ejecutar consultas. Sin embargo, el interface se complica por las "**diferencias conceptuales**" entre el modelo de objetos (dominio de la aplicación) y el modelo relacional (en la base de datos).

El modelo de objetos está basado en principios de ingeniería del software y modela los objetos en el dominio del problema, mientras que el modelo relacional está basado en principios matemáticos y organiza los datos para una almacenamiento y recuperación eficientes.

Ninguno de estos modelos es mejor que el otro, el problema es que son diferentes y no siempre se acoplan de forma confortable en la misma aplicación.

Algunas soluciones a este problema, como Hibernate y Java Data Objects están diseñados para proporcionar al desarrallodor persistencia transparente: la aplicación trata con objetos persistentes utilizano un API orientado a objetos sin necesidad de código SQL embebido en el código Java. La Persistencia Manejada por el Contenedor (CMP) hace un trabajo similar al de los contenedores EJB, pero no existe persistencia general para la plataforma Java. En cualquiera de estas soluciones, los objetos han de ser mapeados a tablas de un SGBDR por el frame subyacente, que genera el SQL necesario para almacenar los objetos. Cuanto más complejo sea el modelo de objetos más díficil será el mapeo. Se necesita crear Descriptores, normalmente ficheros XML, para definir estos mapeos.

La herencia y las relaciones muchos-a-muchos en particular, añaden

complejidad ya que estas relaciones no se pueden representar directamente en el modelo relacional. Los árboles de herencia pueden mapearse a un conjunto de tablas de varias formas, la elección depende de sopesar la eficiencia de almacenamiento y la complejidad de las consultas, ya que se requiere joins de tablas separadas para implementar relaciones muchos-a-muchos.

Almacenar objetos en una BD, que a su vez utiliza su propio modelo de objetos, ofrece otra solución (Modelos objeto-relacionales). Se han desarrollado una gran variedad de Bases de Datos Orientadas a Objetos (OODBMS), pero dichas herramientas pueden ser complejas de configurar y pueden requerir el uso de un lenguaje de definición de objetos. Los objetos se almacenan como objetos, pero no son nativos al lenguaje de la aplicación. Estos productos no han tenido un fuerte impacto en el mercado más allá de sus áreas "nicho" y el esfuerco parece concentrarse principalmente en las APIs orientadas a objetos para bases de datos relacionales así como bases de datos híbridas objeto-relacionales.

Motores de Bases de Datos Embebidos

En algunas aplicaciones no es necesaria la sobreCochega de un potente SGBD empresarial y los requerimientos de almacenacimiento de datos los proporciona mejor un motor de base de datos pequeño y embebible. Por ejemplo SQLite proporciona un motor de base de datos relacional auto-contenido en cada aplicación. Pero aún así la interface con Java es a través del driver JDBC, ya que las soluciones basadas en SQL todavía se ven afectadas por la diferencia de modelo.

En muchos casos la persistencia se puede conseguir más fácilmente utilizando un motor de base de datos de objetos embebido. Nosotros vamos a utilizar dbO4 (aunque hay otros tanto open como comerciales

como objectdb, etc.).

11.1 EL MOTOR db4O

db4o fue creado por Cochel Rosenberg, en un principio comercial, ahora es 'open source' y recientemente se ha liberado bajo la licencia GPL (con ciertas condiciones que puedes ver en su página web). Sus Cocheacterísticas:

- No hay diferencia de modelo: los objetos se almacenan tal y como son en la aplicación orientada a objetos.
- Manejo automático del esquema de base de datos.
- No hay que cambiar las clases para poder almacenarlas.
- Sin complejidades en la unión con el lenguaje Java (o .NET).
- Uniones (JOINS) de datos automatizadas.
- Se instala añadiendo un único fichero de librería de 250Kb (jar para Java o DLL para .NET).
- Cada BD es un único fichero.
- Versionado del esquema automático.
- Consultas-por-ejemplo (Query-By-Example).
- S.O.D.A. (Simple Object Database Access), un API de consultas open source.

La BD db4o es un motor de bases de datos (SGBD) orientado a objetos que consiste en un fichero jar (db4o-8.0-core-java5.jar). Puedes usar una libería cliente/servidor (db4o-8.0-cs-java5.jar) y otros componentes opcionales como soporte de cluster, adapatadores específicos de la plataforma, herramientas estadísticas, etc. en el fichero db4o-8.0-optional-java5.jar

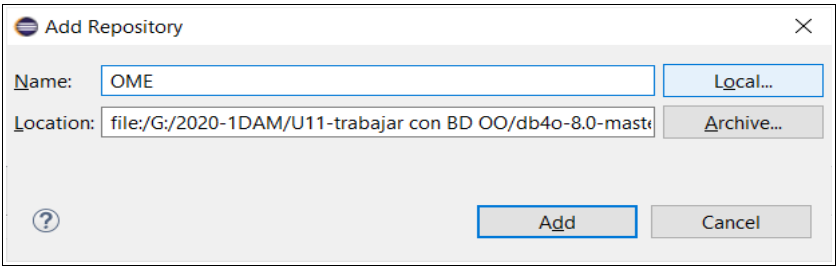
Instalación de db4O

Añade las ficheros **db4o-*.jar** a tu CLASSPATH. Para principiantes es

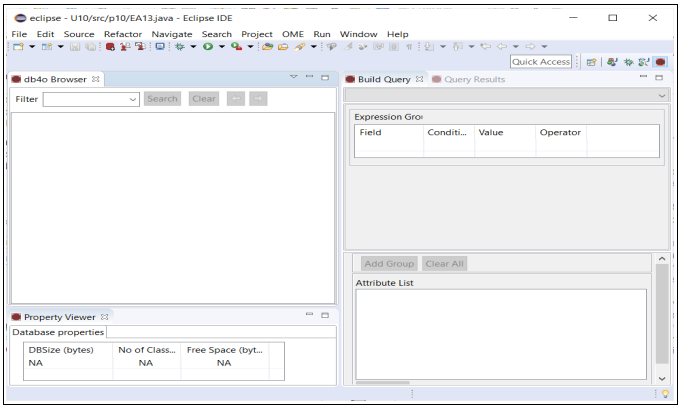
recomendable añadir todo, es decir el jar "**db4o-all.jar**". Si usas un IDE como Eclipse, puedes copiar los ficheros **db4o-*.jar** a la Cochepeta /lib/ de tu proyecto como una librería externa.

Instalación de Object Manager Enterprise (OME)

Object Manager Enterprise (OME) es un visualizador de Objetos de una BD db4o. El fichero zip contiene el plugin de Eclipse. Para instalarlo: unzipea el fichero a una Cochepeta, selecciona 'Help' -> 'Install New Software...' -> 'Add' -> 'Local...' y dale un nombre y selecciona la Cochepeta.



Una vez que tienes OME instalado, accedes seleccionando Window -> Open Perspective -> Other -> "OME". Tiene un aspecto similar a este:



Un vistazo a la API

Los paquetes **com.db4o** y **com.db4o.query** son los que más usaremos.

com.db4o: contiene la mayor parte de la funcionalidad sobre todo los objetos **com.db4o.Db4oEmbedded** y la interface **com.db4o.ObjectContainer**.

La clase **com.db4o.Db4o** tiene métodos estáticos que permiten abrir una base de datos y crear una configuración inicial. Par aplicaciones cliente/servidor necesitas usar **com.db4o.cs.Db4oClientServer** para iniciar un servidor, o connectarte a uno.

La interface más importante es **com.db4o.ObjectContainer**: es tu base de datos -una base de datos en modo aislado o conectado como cliente a un servidor- cada objeto tiene su propia transacción única. Cuando abres un ObjectContainer, estás en una transacción, los cambios que hagas puedes hacerlos permanentes con commit() o deshacerlos con **rollback()**, y comienza automáticamente la siguiente transacción.

Cada **ObjectContainer** mantiene sus propias referencias a los objetos almacenados, de esta manera mantiene la identidad de los objetos y consigue el máximo rendimiento. Cuando cierras el ObjectContainer, todas las referencias a objetos de la RAM se desCochetan.

com.db4o.ext contiene algunos métodos adicionales separados de com.db4o para que las aplicaciones solo importen las extensiones cuando realmente las necesiten. Cada objeto **com.db4o.ObjectContainer** también es un **com.db4o.ext.ExtObjectContainer**. Debes hacer un casting cuando quieras usar unos u otros métodos.

com.db4o.config contiene tipos y clases para configurar db4o.

com.db4o.query contiene la clase **Predicate** para construir consultas nativas. La interface **Query** permite consultar objetos y es preferible que comiences a dominarla antes que la API Soda.

11.2 OPERACIONES BÁSICAS CON DB4O.

Vamos a trabajar con un objeto sencillo (ni herencia, ni composición, ni referencias a otros objetos). Usaremos una clase que representa un Pilotoo de F1.

```
package p11;

public class Pilotoo {
    private String nombre;
    private int puntos;


    public Pilotoo(String nomnbre, int puntos) {
        this.nombre = nombre;
        this.puntos = puntos;
    }


    public int getPuntos() { return puntos; }
    public void sumaPuntos(int p) { this.puntos+= p; }
    public String getNombre() { return nombre; }
    public String toString() { return nombre + "/" + puntos; }
}
```

ABRIR/CREAR Y CERRAR UNA BD STANDALONE (Aislada)

Hay que llamar al método **Db4oEmbedded.openFile()** y aportar una configuración con **Db4oEmbedded.newConfiguration() y la ruta** al fichero de la BD. Devuelve una referencia a una instancia ObjectContainer. Para cerrar la base de datos, se usa el método **close()**.

EJEMPLO 1: Abrir crear una BD y cerrar la conexión.

```
ObjectContainer db = Db4oEmbedded.openFile(
                          Db4oEmbedded.newConfiguration(),
                          "F1.db4" );
try {
     // trabajar con la BD...
} finally {
    db.close();
}
```

Si el fichero indicado existe, se abre. En otro caso, se crea.

ALMACENAR OBJETOS

Se llama al método **store()** pasando el objeto como parámetro.

EJEMPLO 2: Almacenar dos Pilotos.

```
Pilotoo p1 = new Piloto("Michael Schumacher", 100);
db.store(p1);
System.out.println("Almacenado " + p1);
Pilotoo p2 = new Piloto("Rubens Barrichello", 99);
db.store(p2);
System.out.println("Almacenado " + p2);
```

RECUPERAR OBJETOS

db4o tiene varios sistemas de consulta: Query by Example (**QBE**), Native Queries (**NQ**) y el API SODA (**SODA**). Comenzaremos por QBE.

Cuando usamos Query-By-Example, se crea un objeto prototipo que describe lo que quieres recuperar. db4o recuperará todos los objetos de ese tipo que contengan lo mismo (no valores por defecto) que el ejemplo. Los objetos se devuelven como una instancia ObjectSet. Usaremos el método listResult() para mostrar los elementos devueltos.

```
public static void listResult(List<?> r) {
    System.out.println( r.size() );
    for(Object o : r) { System.out.println(o); }
}
```

EJEMPLO 3: Buscar todos los pilotos con QBE.

```
// aportamos un ejemplo vacío -> equivale a todos
// Aunque pasamos el valor 0, no devuelve los pilotos con 0 puntos
// porque 0 es el valor por defecto para los int, long, ...
Piloto proto = new Piloto(null, 0);
ObjectSet r = db.queryByExample(proto);
listResult(r);
```

EJEMPLO 4: Buscar todos los pilotos basándonos en la clase con QBE.

```
ObjectSet r = db.queryByExample(Piloto.class);
listResult(r);
List <Piloto> Pilotos = db.query(Piloto.class);
```

EJEMPLO 5: Buscar un piloto que tenga cierto nombre con QBE.

```
Piloto proto = new Piloto("Michael Schumacher", 0);
ObjectSet r = db.queryByExample(proto);
listResult(r);
```

EJERCICIO 1: Haz un programa que se conecte a F1.db4 y pregunte por una cantidad de puntos y muestre los pilotos que tienen esa puntauación.

ACTUALIZAR OBJETOS

Modificar objetos es una manera de almacenarlos. Puedes recuperarlo previamente de la BD, modificarlo y volver a almacnarlo con store() para actualizarlo (guardar los cambios en la BD). Si antes no lo recuperas de la BD, db4O pensará que no está y lo que hará es una inserción.

EJEMPLO 6: Modificar un piloto.

```
ObjectSet r = db.queryByExample(new Piloto("Michael Schumacher",
0));
Piloto encontrado = (Piloto) r.next();
```

```
encontrado.sumaPuntos(11);
db.store(encontrado);
System.out.println("Se suman 11 puntos a " + encontrado);
r = db.queryByExample(Piloto.class);
listResult(r);
```

BORRAR OBJETOS

Los objetos almacenados se eliminan de la BD con el método **delete()**. Se le pasa el objeto a borrar o un prototipo que se le parezca.

EJEMPLO 6: Buscar un piloto que tenga cierto nombre con QBE.

```
ObjectSet r = db.queryByExample(new Piloto("Michael Schumacher",
0));
Piloto e = (Piloto) r.next();
db.delete(e);
System.out.println("Borrado " + e);
ObjectSet r = db.queryByExample(Piloto.class);
listResult(r);
```

11.3 OPERACIONES MÁS AVANZADAS.

Ahora trabajaremos con objetos más complejos y veremos como se manejan. Añadimos la clase Coche que usa composición, incluyendo un objeto Piloto en una de sus variables de instancia.

```
package p11;

public class Coche {
    private String modelo;
    private Piloto piloto;

    public Coche(String modelo) {
        this.modelo = modelo;
        this.piloto = null;
    }

    public Piloto getPiloto() { return piloto; }
    public void setPiloto(Piloto piloto) { this.Piloto = piloto; }
```

```
        public String getModelo() { return modelo; }
        public String toString() {
            return modelo + "[" + piloto + "]";
        }
    }
```

11.3.1 OBJETOS COMPUESTOS.

ALMACENAR OBJETOS COMPUESTOS

Para almacenar un coche con su piloto, usamos store() sobre el objeto contenedor (Coche) y el objeto Piloto (contenido) se almacena de manera automática.

EJEMPLO 7: Almacenar un coche y su piloto de forma implícita.

```
Piloto proto = new Piloto("Michael Schumacher", 0);
ObjectSet r = db.queryByExample(proto);
listResult(r);
Coche c1 = new Coche("Ferrari");
Piloto p1 = new Piloto("Michael Schumacher", 100);
c1.setPiloto(p1);
db.store(c1);
```

Si nosotros explícitamente almacenamos el piloto antes de almacenar el coche, no hay ninguna diferencia.

EJEMPLO 8: Almacenar un coche y su piloto de forma explícita.

```
Piloto p2 = new Piloto("Rubens Barrichello", 99);
db.store(p2);
Coche c2 = new Coche("BMW");
c2.setPiloto(p2);
db.store(c2);
```

RECUPERAR OBJETOS CON QBE

Para recuperar todos los coches, simplemente aportamos un prototipo

en "blanco".

EJEMPLO 9: Recuperar todos los coches y luego todos los pilotos.

```
Coche proto = new Coche(null);
ObjectSet r = db.queryByExample(proto);
listResult(r);
Piloto protop = new Piloto(null, 0);
r = db.queryByExample(protop);
listResult(r);
```

Ahora vamos a crear un prototipo para indicar todos los coches conducidos por Rubens Barrichello.

EJEMPLO 10: Recuperar todos los coches conducidos por un piloto.

```
Piloto pProto = new Piloto("Rubens Barrichello", 0);
Coche cProto = new Coche(null);
cProto.setPiloto(pProto);
ObjectSet r = db.queryByExample(cProto);
listResult(r);
```

RECUPERAR OBJETOS CON NQ (NATIVE QUERIES)

NQ utiliza un objeto Predicate para indicar la condición que debe cumplir un objeto para ser recuperado. Vamos a recuperar los coches conducidos por un piloto en concreto.

EJEMPLO 11: Almacenar un coche y su piloto de forma explícita.

```
final String pNombre = "Rubens Barrichello";
List<Coche> r = db.query(
        new Predicate<Coche>() {
            public boolean match(Coche c) {
                return c.getPiloto().getNombre().equals(pNombre);
            }
        });
listResult(r);
```

RECUPERAR OBJETOS CON SODA

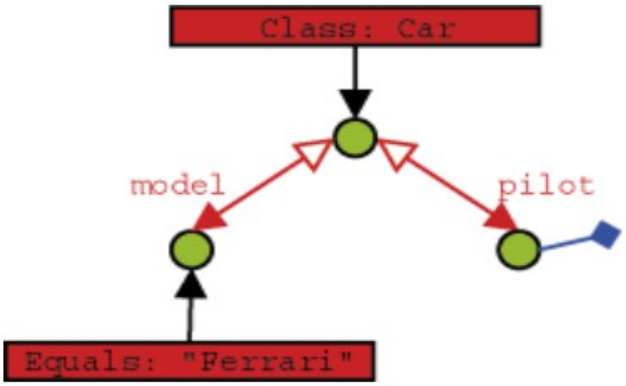
Para recuperar los coches que conduce un piloto, tendremos que descender dos niveles en nuestra consulta:

```
Query query = db.query();
query.constrain(Coche.class);
query.descend("Piloto").descend("nombre")
     .constrain("Rubens Barrichello");
ObjectSet r = query.execute();
listResult(r);
```

También podemos restringir el campo Piloto a través de un prototipo:

```
Query query = db.query();
query.constrain(Coche.class);
Piloto proto = new Piloto("Rubens Barrichello", 0);
query.descend("Piloto").constrain(proto);
ObjectSet r = query.execute();
```

Cuando realizamos un descenso en una consulta, podemos aportar otra Querie como restricción. De forma que podemos comenzar por la clase padre y desdencer a la hija, o por la hija y ascender a la padre.



EJEMPLO 12: Pilotos que conducen un coche del modelo "Ferrari".

```
Query cq = db.query();
cq.constrain(Coche.class);
cq.descend("modelo").constrain("Ferrari");
Query pq = cq.descend("piloto");
ObjectSet r = pq.execute();
listResult(r);
```

MODIFICAR OBJETOS COMPLEJOS

Para modificar objetos compuestos, volvemos a utilizar store(). En este código modificamos el piloto de un coche, y luego le sumamos puntos a su piloto.

```
List<Coche> r = db.query(
                new Predicate<Coche>() {
                     public boolean match(Coche c) {
                         return c.getModelo().equals("Ferrari");
                     }
                });
Coche cf = (Coche) r.get(0);
cf.setPiloto( new Piloto("Fernando Alonso", 0) );
db.store(cf);
r = db.query( new Predicate<Coche>() {
                  public boolean match(Coche c) {
                       return c.getModelo().equals("Ferrari");
                  }
               });
listResult(r);
// Modificar ambos en una única sesión
List<Coche> r = db.query(
                   new Predicate<Coche>() {
                       public boolean match(Coche c) {
                           return c.getModelo().equals("Ferrari");
                       }
                   });
Coche cf = r.get(0);
cf.getPiloto().sumaPuntos(1);
db.store(cf);
r = db.query(new Predicate<Coche>() {
                  public boolean match(Coche c) {
                      return c.getModelo().equals("Ferrari");
             }
```

```
                    });
        listResult(r);
```

Bastante sencillo, ¿No? Pero, qué pasa si dividimos estas dos modificaciones en dos sesiones diferentes, en una modificamos el coche y en otra el piloto. Supngamos que primero modificamos el piloto y luego el coche:

```
        // El piloto del coche se modifica en la sesión 1 y en esta...
        List<Coche> r = db.query(
                            new Predicate<Coche>() {
                                public boolean match(Coche c) {
                                   return c.getModelo().equals("Ferrari");
                                }
                           });
        Coche cf = r.get(0);
        cf.getPiloto().sumaPuntos(1);
        db.store(cf);
        List<Coche> r = db.query(
                             new Predicate<Coche>() {
                                 public boolean match(Coche c) {
                                    return c.getModelo().equals("Ferrari");
                                 }
                             });
        listResult(r);

GENERA LA SALIDA!!
        1
        Ferrari[Fernando Alonso/0]
```

Parece que tenemos un problema, porque los puntos del piloto no cambian ¿Qué ha ocurrido y cómo solucionarlo?

PROFUNDIDAD DE MODIFICACIONES

Imagina que tienes un objeto complejo con muchos miembros que ellos mismos son también complejos. Si cuando actualizas este objeto, también tienes que hacerlo con cada hijo, con cada hijo del hijo, etc. El proceso se vuelve ineficiente y puede que innecesario. Sin embargo,

otras veces habrá que hacerlo. En el caso anterior, modificamos el objeto piloto (contenido en un objeto coche) . Al guardar el cambio, guardamos el objeto Coche y hemos asumido que el piloto modificado también se guardará.

La primera vez, db4o nunca modificó el piloto en la BD, siempre usó el que tenía en memoria, no guardó los cambios en la BD. Si reinicias la aplicación, verás que el objeto no ha cambiado.

Para solucionar este dilema, db4o introduce el concepto de profundidad de modificaciones que indica cuántos niveles de miembros se modifican Por defecto está a 1, lo que significa que solamente las variables miembro de tipo primitivo y String se actualizan, pero no los cambios que ocurran en los objetos miembro del que se modifica.

db4o aporta mucho control de esta profundidad. Usando **cascadeOnUpdate()** hacemos que se actualizan tantos miembros se tengan.

EJEMPLO 13: Pilotos que conducen un coche del modelo "Ferrari".

```
// SESIÓN 1
EmbeddedConfiguration cfg = Db4oEmbedded.newConfiguration();
cfg.common().objectClass(Coche.class).cascadeOnUpdate(true);
ObjectContainer db = Db4oEmbedded.openFile(cfg, "F1.db4");
List<Coche> r = db.query(new Predicate<Coche>() {
                  public boolean match(Coche c) {
                     return c.getModelo().equals("Ferrari");
                  }
                });
if( r.size() > 0) {
   Coche ce = r.get(0);
   ce.getPiloto().sumaPuntos(1);
   db.store(ce);
}
db.close();
// SESIÓN 2
```

```
ObjectContainer db = Db4oEmbedded.openFile(
                        Db4oEmbedded.newConfiguration(), "F1.db4");
List<Coche> r = db.query(new Predicate<Coche>() {
                         public boolean match(Coche c) {
                            return c.getModelo().equals("Ferrari");
                         }
                    });
Coche c = result.get(0);
listResult(r);
db.close();

SALIDA:
1
Ferrari[Fernando Alonso/1]
```

Ahora si que queda almacenada la modificación del piloto.

BORRANDO OBJETOS COMPLEJOS

Como ya vimos, usando el método delete().

```
List<Coche> r = db.query(new Predicate<Coche>() {
                       public boolean match(Coche c) {
                          return c.getModelo().equals("Ferrari");
                       }
                  });
Coche cf = r.get(0);
db.delete(cf);
r = db.queryByExample( new Coche(null) );
listResult(r);
Piloto proto = new Piloto(null, 0);
r = db.queryByExample(proto);
listResult(r);
```

```
SALIDA:
1
BMW[Rubens Barrichello/99]
3
Michael Schumacher/100
Rubens Barrichello/99
Fernando Alonso/1
```

Todo correcto, se borra el coche, pero no el piloto que lo condece. Pero a veces querremos que se borren los objetos que forman (contenidos) en el principal.

BORRANDO RECURSIVAMENTE OBJETOS

Para realizar un borrado recursivo de objetos contenidos en otros que se borran es parecido a las modificaciones, hay que configurar a db4o para que borre también al objeto Piloto de un coche cuando el objeto Coche se borre.

```
EmbeddedConfiguration cfg = Db4oEmbedded.newConfiguration();
cfg.common()
   .objectClass(Coche.class)
   .cascadeOnDelete(true);
ObjectContainer db = Db4oEmbedded.openFile(cfg, "F1.db4");
List<Coche> r = db.query( new Predicate<Coche>() {
                  public boolean match(Coche c) {
                      return c.getModelo().equals("BMW");
                  }
                });
if (r.size() > 0) {
    Coche cf = r.get(0);
    db.delete(cf);
}
r = db.query(new Predicate<Coche>() {
              public boolean match(Coche Coche) { return true; }
           });
listResult(r);
db.close();
```

De nuevo se aplica en la configuración que se pasa al abrir el ObjectContainer.

Nos puede surgir una pregunta, ¿Qué ocurre si los objetos contenidos referencian a un objeto que ha sido borrado? Pues que hay que ser muy cuidadoso porque actualmente db4O no comprueba si un elemento

borrado es referenciado por otros objetos.

```
        EmbeddedConfiguration cfg = Db4oEmbedded.newConfiguration();
        cfg.common()
           .objectClass(Coche.class)
           .cascadeOnDelete(true);
        ObjectContainer db = Db4oEmbedded.openFile(cfg, "F1.db4");
        ObjectSet<Piloto> r = db.query(new Predicate<Piloto>() {
                                    public boolean match(Piloto p) {
                        return p.getNombre().equals("Michael Schumacher");
                                    }
                                });
        if (!result.hasNext()) {
           System.out.println("Piloto no encontrado!");
           db.close();
           return;
        }
        Piloto p = (Piloto) result.next();
        Coche c1 = new Coche("Ferrari");
        Coche c2 = new Coche("BMW");
        c1.setPiloto(p);
        c2.setPiloto(p);
        db.store(c1);
        db.store(c2);
        db.delete(c2);
        List<Coche> co = db.query(new Predicate<Coche>() {
                         public boolean match(Coche c) { return true; }
                          });
        listResult(co);
        db.close();

SALIDA:
        1
        Ferrari[Michael Schumacher/100]

        Piloto proto = new Piloto(null, 0);
        ObjectSet r = db.queryByExample(proto);
        listResult(r);

SALIDA:
        1
        Fernando Alonso/1
```

EJEMPLO 14: Borrar toda la base de datos.

```
ObjectSet r = db.queryByExample( new Object() );
while( r.hasNext() ) {
  db.delete( r.next() );
}
```

11.3.2 ARRAYS Y COLECCIONES.

Ahora usaremos otra clase que recoge las lecturas de sensores que hemos instalado en los coches.

```
package p11;

import java.util.*;

public class LecturasSensor {
    private double[] valores;
    private Date time;
    private Coche coche;
    public LecturasSensor(double[] v, Date t, Coche c) {
        this.valores = v;
        this.time = t;
        this.coche = c;
    }
    public Coche getCoche() { return coche; }
    public Date getTime() { return time; }
    public int getNumValores() { return valores.length; }
    public double[] getValores(){ return valores; }
    public double getValor(int idx) { return valor[idx]; }
    public String toString() {
        StringBuffer str = new StringBuffer();
        str.append(coche.toString())
           .append(" : ")
           .append(time.getTime())
           .append(" : ");
        for(int idx = 0; idx < valores.length; idx++) {
            if( idx > 0 ) {
                str.append(',');
            }
            str.append(valores[idx]);
        }
        return str.toString();
```

```
    }
}
```

Un coche genera las lecturas de su sensor cuando se le pisa la lista de valores durante una carrera.

```
package p11;

import java.util.*;

public class Coche {
    private String modelo;
    private Piloto piloto;
    private List historico;
    public Coche(String modelo) {
        this(modelo,new ArrayList());
    }
    public Coche(String modelo,List historico) {
        this.modelo = modelo;
        this.piloto = null;
        this.historico = historico;
    }
    public Piloto getPiloto() { return piloto; }
    public void setPiloto(Piloto piloto) { this.Piloto = piloto; }
    public String getModelo() { return modelo; }
    public List getHistorico() { return historico; }
    public void instantaneat() {
        historico.add(new LecturasSensor(poll(),new Date(),this));
    }
    protected double[] poll() {
       int factor = historico.size() + 1;
       return new double[] {0.1*factor, 0.2*factor, 0.3*factor };
    }
    public String toString() {
        return modelo + "[" + piloto + "]/" + historico.size();
    }
}
```

ALMACENAR

De la forma ya mencionada.

```
Coche c1 = new Coche("Ferrari");
Piloto p1 = new Piloto("Michael Schumacher", 100);
c1.setPiloto(p1);
db.store(c1);
```

Ahora guardamos un segundo coche al que se le piden dos snapshots.

```
Piloto p2 = new Piloto("Rubens Barrichello", 99);
Coche c2 = new Coche("BMW");
c2.setPiloto(p2);
c2.snapshot();
c2.snapshot();
db.store(c2);
```

CONSULTAS CON QBE

Primero comprobamos que tiene snapshots.

```
LecturasSensor proto = new LecturasSensor(null, null, null);
ObjectSet r = db.queryByExample(proto);
listResult(r);
SALIDA:
2
BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.1,0.2,0.3
BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.2,0.4,0.6
```

Para usar un prototipo para el array, creamos uno solo con los valores que esperamos que tenga:

```
LecturasSensor proto = new LecturasSensor(new double[] {0.3,
0.1 }, null, null);
ObjectSet r = db.queryByExample(proto);
listResult(r);
SALIDA:
1
BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.1,0.2,0.3
```

Observa que la posición de los valores en el array no influye, simplemente se busca aquellos que los contienen.

Para recuperar un cocher por sus lecturas almacenadas, instalamos un historico que contenga los valores.

```
        LecturasSensor protoLec =
                new LecturasSensor(new double[] { 0.6, 0.2 }, null, null);
        List protoHis = new ArrayList();
        protoHis.add(protoLec);
        Coche protoCoche = new Coche(null, protoHis);
        ObjectSet r = db.queryByExample(protoCoche);
        listResult(r);
SALIDA:
        1
        BMW[Rubens Barrichello/99]/2
```

También podemos consultar directamente las colecciones puesto que son objetos.

```
        ObjectSet r = db.queryByExample(new ArrayList());
        listResult(r);
SALIDA:
        2
        []
        [BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.1,0.2,0.3,
        BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.2,0.4,0.6]
```

Pero no funciona con arrays.

```
        ObjectSet r = db.queryByExample(new double[] { 0.6, 0.4 });
        listResult(r);
SALIDA:
        0
```

CONSULTAS NATIVAS

Para encontrar las lecturas que coincidan en ciertos valores simplemente escribimos lo que queremos comprobar en una instancia:

```
        List<LecturasSensor> r = db.query(
            new Predicate<LecturasSensor>() {
                  public boolean match(LecturasSensor ls) {
            return Arrays.binarySearch(ls.getValores(), 0.3) >= 0
```

```
            && Arrays.binarySearch( ls.getValores(), 1.0) >= 0;
            }
           });
        listResult(r);
SALIDA:
        0
```

Ahora coches que tengan ciertas lecturas:

```
        List<Coche> r = db.query(
            new Predicate<Coche>() {
                public boolean match(Coche cc) {
                    List hist = c.getHistorico();
                    for (Object ah : hist) {
                    LecturasSensor ro = (LecturasSensor) ah;
                    if( Arrays.binarySearch(ro.getValores(), 0.6) >= 0
                        || Arrays.binarySearch(ro.getValores(), 0.2) >= 0)
                        return true;
                    }
                    return false;
                }
             });
        listResult(r);
SALIDA:
        1
        BMW[Rubens Barrichello/99]/2
```

CONSULTAS CON SODA

Es similar a los ejemplos anteriores. Primero, recuperamos solo las que tengan valores específicos.

```
        Query q = db.query();
        q.constrain(LecturasSensor.class);
        Query valoresQ = q.descend("valores");
        valoresQ.constrain(0.3);
        valoresQ.constrain(0.1);
        ObjectSet r = q.execute();
        listResult(r);
SALIDA:
        1
        BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.1,0.2,0.3
```

Ahora los coches que coincidan con unas lecturas:

```
        Query q = db.query();
        q.constrain(Coche.class);
        Query historicoQ = q.descend("historico");
        historicoQ.constrain(LecturasSensor.class);
        Query valoresQ = historicoQ.descend("valores");
        valoresQ.constrain(0.3);
        valoresQ.constrain(0.1);
        ObjectSet r = q.execute();
        listResult(r);
SALIDA:
        1
        BMW[Rubens Barrichello/99]/2
```

ACTUALIZAR Y BORRAR

Igual que antes teniendo en cuenta la profundidad.

```
        // Actualizar el coche
        EmbeddedConfiguration cfg = Db4oEmbedded.newConfiguration();
        cfg.common().objectClass(Coche.class).cascadeOnUpdate(true);
        ObjectContainer db = Db4oEmbedded.openFile(cfg, "F1.db4");
        List<Coche> r = db.query(new Predicate<Coche>() {
                          public boolean match(Coche cc) { return true; }
                        });
        if(r.size() > 0) {
            Coche c = results.get(0);
            c.snapshot();
            db.store(c);
            // Mostrar todos los coches...
        }
        db.close();
SALIDA:
        3
        BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.1,0.2,0.3
        BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.2,0.4,0.6
        Ferrari[Michael Schumacher/100]/1 : 1352132370757 : 0.1,0.2,0.3
```

El borrado tampoco tiene nada de especial. Borrar es como actualizar, ejemplo:

```
        EmbeddedConfiguration cfg = Db4oEmbedded.newConfiguration();
        cfg.common().objectClass(Coche.class).cascadeOnUpdate(true);
        ObjectContainer db = Db4oEmbedded.openFile(cfg, "F1.db4");
        ObjectSet<Coche> r = db.query(new Predicate<Coche>() {
                            public boolean match(Coche cc) { return true; }
                              });
        if(r.hasNext()) {
            Coche c = (Coche) r.next();
            c.getHistorico().remove(0);
            db.store( c.getHistorico() );
            r = db.query(new Predicate<Coche>() {
                  public boolean match(Coche cc) { return true; }
                 });
            while(r.hasNext()) {
                c = r.next();
                for(int idx = 0; idx < c.getHistorico().size(); idx++) {
                    System.out.println(c.getHistorico().get(idx));
                }
            }
        }
        db.close();
SALIDA:
        BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.1,0.2,0.3
        BMW[Rubens Barrichello/99]/2 : 1352132370523 : 0.2,0.4,0.6
```

Borrar todos los objetos:

```
        EmbeddedConfiguration cfg = Db4oEmbedded.newConfiguration();
        cfg.common().objectClass(Coche.class).cascadeOnDelete(true);
        ObjectContainer db = Db4oEmbedded.openFile(cfg, "F1.db4");
        ObjectSet<Coche> coches = db.query(new Predicate<Coche>() {
                    public boolean match(Coche cc) { return true; }
                    });
        while (coches.hasNext()) { db.delete(coches.next()); }
        ObjectSet<LecturasSensor> ro = db.query(
                            new Predicate<LecturasSensor>() {
             public boolean match(LecturasSensor candidate) {return true;}
                                      });
        while (ro.hasNext()) { db.delete(ro.next()); }
        db.close();
```

11.3.3 TRABAJAR CON OBJETOS QUE USAN HERENCIA.

Definimos y modificamos algunas de las clases.

```
package p11;

import java.util.*;

public class LecturasSensor {
    private Date time;
    private Coche Coche;
    private String descripcion;

    protected LecturasSensor(Date t, Coche c, String desc) {
        this.time = t;
        this.coche = c;
        this.descripcion = desc;
    }
    public Coche getCoche() { return coche; }
    public Date getTime() { return time; }
    public String getDescripcion() { return descripcion; }
    public String toString() {
        return coche + " : " + time + " : " + descripcion;
    }
}
```

```
package p11;

import java.util.*;

public class Temperaturas extends LecturasSensor {
    private double temperatura;
    public Temperaturas(Date t, Coche c, String d, double temp){
        super(t, c, d);
        this.temperatura = temperatura;
    }
    public double getTemperatura() { return temperatura; }
    public String toString() {
        return super.toString() + " temp : " + temperatura;
    }
}
```

```
package p11;
```

```
import java.util.*;

public class Presiones extends LecturasSensor {
    private double presion;
    public Presiones(Date t, Coche c, String d, double p) {
        super(t, c, d);
        this.presion = presion;
    }
    public double getPresion() { return presion; }
    public String toString() {
        return super.toString() + " presion : " + presion;
    }
}
```

Modificamos los coches:

```
package p11;

import java.util.*;

public class Coche {
    private String modelo;
    private Piloto piloto;
    private List historico;
    public Coche(String modelo) {
        this.modelo = modelo;
        this.piloto = null;
        this.historico = new ArrayList();
    }
    public Piloto getPiloto() { return piloto; }
    public void setPiloto(Piloto piloto) { this.piloto = piloto; }
    public String getModel() { return model; }
    public LecturasSensor[] getHistorico() {
        return (LecturasSensor[]) historico.toArray(
                  new LecturasSensor[historico.size()]);
    }
    public void snapshot() {
        historico.add(new Temperaturas(
            new Date(), this,"aceite", aceiteTemp()));
        historico.add(new Temperaturas(
             new Date(), this, "agua", aguaPres()));
        historico.add(new Presion(
             new Date(),this,"aceite", aceitePres()));
```

```
        }
        protected double aceiteTemp() { return 0.1*historico.size(); }
        protected double aguaTemp() { return 0.2*historico.size(); }
        protected double aceitePres() { return 0.3*historico.size(); }
        public String toString() {
            return modelo + "[" + piloto + "]/" + historico.size();
        }
    }
```

ALMACENAR

Externamente la forma de trabajar no cambia, salvo internamente el snapshot.

```
Coche c1 = new Coche("Ferrari");
Piloto p1 = new Piloto("Michael Schumacher",100);
c1.setPiloto(p1);
db.store(c1);
Piloto p2 = new Piloto("Rubens Barrichello",99);
Coche c2 = new Coche("BMW");
c2.setPiloto(p2);
c2.snapshot();
c2.snapshot();
db.store(c2);
```

RECUPERAR

db4o puede trabajar con objetos de todo tipo independientemente de que sean clases derivadas o instancias directas de la clase.

```
// recuperar temperaturas con QBE
LecturasSensor proto = new Temperaturas(null, null, null, 0.0);
ObjectSet r = db.queryByExample(proto);
listResult(r);
```

```
SALIDAS:
4
BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021 : aceite
temp : 0.0
BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021 : water
temp : 0.2
BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021 : oil
temp : 0.30000000000000004
BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021 : water
```

```
        temp : 0.8

        // todas las lecturas de los sensores con QBE
        LecturasSensor proto = new LecturasSensor(null, null, null);
        ObjectSet r = db.queryByExample(proto);
        listResult(r);
SALIDAS:
        6
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2012 : aceite
        temp : 0.0
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2012 : agua
        temp : 0.2
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2012 : aceite
        presion : 0.6
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2012 : aceite
        temp : 0.30000000000000004
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2012 : agua
        temp : 0.8
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2012 : aceite
        presion : 1.5
```

Un caso en el que QBE no podría aplicarse es cuando tenemos una interface o una clase abstracta. En ese caso se puede solucionar con un truco.

```
        ObjectSet r = db.queryByExample(LecturasSensor.class);
        listResult(r);
SALIDA:
        6
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021:aceite
        temp : 0.0
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021:agua
        temp : 0.2
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021:aceite
        presion : 0.6
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021:aceite
        temp : 0.30000000000000004
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021:agua
        temp : 0.8
        BMW[Rubens Barrichello/99]/6 : Mon Nov 05 16:19:30 GMT 2021:aceite
        presion : 1.5
```

Con SODA:

```
Query q = db.query();
q.constrain(LecturasSensor.class);
ObjectSet r = q.execute();
listResult(r);
```

ACTUALIZAR Y BORRAR

Igual que antes, no hay diferencias. Por ejemplo, borrar todo el contenido:

```
ObjectSet r = db.queryByExample(new Object());
while(r.hasNext()) { db.delete(r.next()); }
```

11.3.4 TRANSACCIONES.

Cuando db4o trabaja en modo cliente servidor, permite a varios usuarios acceder al mismo tiempo. En ese apartado comentaremos los conceptos de aislamiento de las transacciones.

COMMIT y ROLLBACK

Como al abrir una base de datos ya se inicia una transacción implícita y se comitea cuando la cierras, ya estás usando transacciones aunque de forma transparente. Las sentencias commit y rollback están disponibles para usarlas de forma explícita.

```
Piloto p = new Piloto("Rubens Barrichello",99);
Coche c = new Coche("BMW");
c.setPiloto(p);
db.store(c);
db.commit();
ObjectSet r = db.queryByExample(Coche.class);
listResult(r);
```

También podemos deshacer los cambios realizados durante la transacción, devolviendo su estado al último commit realizado:

```
Piloto p2 = new Piloto("Michael Schumacher",100);
```

```
        Coche c2 = new Coche("Ferrari");
        c2.setPiloto(p2);
        db.store(c2);
        db.rollback();
        ObjectSet r = db.queryByExample(Coche.class);
        listResult(r);
SALIDA:
        1
        BMW[Rubens Barrichello/99]/0
```

REFRESCAR OBJETOS VIVOS

Hay un problema: si hacemos rollback, la operación afecta a los objetos almacenados en la base de datos, pero no a los objetos que estén existiendo en la Ram de nuestra aplicación.

```
        ObjectSet r = db.queryByExample(new Coche("BMW"));
        Coche c = (Coche)r.next();
        c.snapshot();
        db.store(c);
        db.rollback();
        System.out.println(c);
SALIDA:
        BMW[Rubens Barrichello/99]/3
```

Para solucionarlo hay que actualizar los objetos vivos de la application que hayan participado en un rollback de una transacción para sincronizar el estado de los objetos de la aplicación con el de los objetos almacenados en la BD.

```
        ObjectSet r = db.queryByExample(new Coche("BMW"));
        Coche c = (Coche)r.next();
        c.snapshot();
        db.store(c);
        db.rollback();
        db.ext().refresh(c, Integer.MAX_VALUE);
        System.out.println(c);
SALIDA:
        BMW[Rubens Barrichello/99]/0
```

La operación **ext().refresh()** sincroniza el objeto con su valor en la BD.

11.3.5 ACTIVACIÓN TRANSPARENTE.

Cuando tenemos por ejemplo:

```
Piloto p = new Piloto("Kimi Raikkonen", 110);
Coche c = new Coche("Ferrari");
c.setPiloto(p);
for(int i = 0; i < 5; i++) {
  c.snapshot();
}
db.store(c);
```

Si atravesamos todos los coches y sus lecturas de sensores, veremos algunos problemas:

```
ObjectSet r = db.queryByExample(Coche.class);
Coche c = (Coche) r.next();
LecturasSensor ro = Coche.getHistorico();
while (ro != null) {
    System.out.println(ro);
    ro = ro.getNext();
}
```

```
Ferrari[Kimi Raikkonen/110]/5 : Mon Nov 05 16:19:31 GMT 2021 : oil
temp : 0.0
Ferrari[Kimi Raikkonen/110]/5 : Mon Nov 05 16:19:31 GMT 2021 : water
temp : 0.2
Ferrari[Kimi Raikkonen/110]/5 : Mon Nov 05 16:19:31 GMT 2021 : oil
pressure : 0.6
Ferrari[Kimi Raikkonen/110]/5 : Mon Nov 05 16:19:31 GMT 2021 : oil
temp : 0.30000000000000004
null : null : null temp : 0.0
```

Para evitarlos, hay que configurar el modo activación transparente.

```
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.common().add(new TransparentActivationSupport());
ObjectContainer db = Db4oEmbedded.openFile(config, "F1.db4");
ObjectSet r = db.queryByExample(Coche.class);
if (r.hasNext()) {
    Coche c = (Coche) r.next();
    LecturasSensor ro = c.getHistorico();
    while (ro != null) {
```

```
            System.out.println(ro);
            ro = ro.getNext();
        }
    }
    db.close();

    SALIDA:
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    temp : 0.0
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : water
    temp : 0.2
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    pressure : 0.6
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    temp : 0.30000000000000004
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : water
    temp : 0.8
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    pressure : 1.5
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    temp : 0.6000000000000001
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : water
    temp : 1.4000000000000001
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    pressure : 2.4
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    temp : 0.9
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : water
    temp : 2.0
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    pressure : 3.3
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    temp : 1.2000000000000002
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : water
    temp : 2.6
    Ferrari[Kimi Raikkonen/110]/15 : Mon Nov 05 16:19:31 GMT 2012 : oil
    pressure : 4.2
```

Así no hay sorpresas con miembros null que no hayan sido leídos desde la BD. Si está activada, todos los objetos que no implementen la interface com.db4o.ta.Activatable serán completamente activados ciando se usen. Si cargar todos los objetos enlazados por otros supone un problema de rendimiento debes investigar como implementar la interface Activatable.

11.3.6 PERSISTENCIA TRANSPARENTE.

Controlar la profundidad de las actualizaciones da cierto control al programador para balacear entre modificaciones y eficiencia, pero lo ideal sería que el propio motor decidiese cuando un objeto debe ser almacenado, esta capacidad es la que se conoce como persistencia Transparente. ¿Cómo usarla?

1. Hay que configurar la base de datos. Con TransparentPersistenceSupport.
2. Las clases que la usen deben implementar la interface Activatable que tiene el método bind() para asociar objetos con el contenedor.
3. El objeto es asociado al contenedor cuando es instanciado o almacenado por primera vez en la BD.
4. Cuando un campo del objeto se cambia, se llama al método activate() para ser almacenado en el próximo commit.

Ejemplo: haremos las clases Coche y LecturasSensor con persistencia transparente.

```
package p11;

import java.util.*;
import com.db4o.activation.*;
import com.db4o.ta.*;

public class Coche implements Activatable {
    private String modelo;
    private LecturasSensor historico;
    private transient Activator _activator;
    public Coche(String modelo) {
        this.modelo = modelo;
        this.historico = null;
    }
public String getModelo() {
    activate(ActivationPurpose.READ);
```

```
        return modelo;
    }
    public LecturasSensor getHistorico() {
        activate(ActivationPurpose.READ);
        return historico;
    }
    public void snapshot() {
        activate(ActivationPurpose.WRITE);
        appendToHistory(new Temperaturas(
                            new Date(),this,"aceite", aceiteTemp()));
        appendToHistory(new Temperaturas(
                            new Date(),this,"agua", aguaTemp()));
    }
    protected double aceiteTemp() { return 0.1*contador(); }
    protected double aguaTemp() { return 0.2*contador(); }
    public String toString() {
        activate(ActivationPurpose.READ);
        return modelo + "/" + contador();
    }

    private int contador() {
        activate(ActivationPurpose.READ);
        return (historico==null ? 0 : historico.cuenta());
    }
    private void appendToHistory(LecturasSensor ro) {
        activate(ActivationPurpose.WRITE);
        if(historico==null) { historico = ro; }
        else { historico.append(ro); }
    }
    public void activate(ActivationPurpose purpose) {
        if(_activator != null) {
            _activator.activate(purpose);
        }
    }
    public void bind(Activator activator) {
        if (_activator == activator) {  return; }
        if (activator != null && _activator != null) {
           throw new IllegalStateException();
        }
        _activator = activator;
    }
    }
```

En la clase Sensores habría que hacer modificaciones similares. Y por último, al usar la BD, debemos activar el uso de esta característica:

```
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.common().add(new TransparentPersistenceSupport());
ObjectContainer db = Db4oEmbedded.openFile(config, "F1.db4");
Coche c = new Coche("Ferrari");
for (int i = 0; i < 3; i++) {  c.snapshot(); }
db.store(c);
db.close();
// Todos los objetos guardados
EmbeddedConfiguration config = Db4oEmbedded.newConfiguration();
config.common().add(new TransparentPersistenceSupport());
ObjectContainer db = Db4oEmbedded.openFile(config, "F1.db4");
System.out.println("Leer datos de sensores y modifico descrip:");
ObjectSet result = db.queryByExample(Coche.class);
if (result.hasNext()) {
    Coche c = (Coche) result.next();
    LecturasSensor ro = c.getHistorico();
    while (ro != null) {
        System.out.println(ro);
        ro.setDescripcion("Modificado: " + ro.getDescription());
        ro = ro.getNext();
    }
    db.commit();
}
db.close();
```

Aunque no hemos guardado nada en la segunda modificación, si consultas la BD, todos los cambios están actualizados.

11.3.7 CLIENTE / SERVIDOR.

Trabajaremos con estos datos:

```
Piloto p = new Piloto("Rubens Barrichello", 99);
Coche c = new Coche("BMW");
c.setPiloto(p);
db.store(c);
Piloto p2 = new Piloto("Michael Schumacher", 100);
Coche c2 = new Coche("Ferrari");
```

```
c2.setPiloto(p2);
db.store(c2);
```

SERVIDOR EMBEBIDO

Desde el punto de vista de la API no hay diferencias entre una transacción que se ejecute en la misma máquina virtual que aquellas que se ejecuten en máquinas remotas. Para usar transacciones en la misma máquina, abre un servidor de un fichero db4O, ejecutándose en un puerto 0, se indica que no hay red que los separe.

```
// acceder a servidor local
ObjectServer server = Db4oClientServer.openServer(
     Db4oClientServer.newServerConfiguration(), "F1.db4", 0);
try {
    ObjectContainer client = server.openClient();
    // Trabajar con el cliente o abrir más ...
    client.close();
} finally {
    server.close();
}
```

El nivel de aislamiento de una transacción db4o es read committed. Sin embargo, cada cliente tiene su propia caché de objetos que conoce. Para que los cambios comiteados por otros clientes se sincronicen con nuestros propios objetos debemos refrescarlos. Podemos delegar este trabajo a una versión especializada del método listResult() que hicimos.

```
public static void listRefrescaResult(ObjectContainer container,
                                       ObjectSet result,
                                       int depth) {
    System.out.println( result.size() );
    while( result.hasNext() ) {
        Object obj = result.next();
        container.ext().refresh(obj, depth);
        System.out.println(obj);
    }
}
```

```
        // Usar conexiones locales
        ObjectContainer cliente1 = server.openClient();
        ObjectContainer cliente2 = server.openClient();
        Piloto p1 = new Piloto("David Coulthard", 98);
        ObjectSet r = cliente1.queryByExample(new Coche("BMW"));
        Coche c1 = (Coche) r.next();
        c1.setPiloto(p1);
        cliente1.store(c1);
        listResult(cliente1.queryByExample(new Coche(null)));
        listResult(cliente2.queryByExample(new Coche(null)));
        cliente1.commit();
        listResult(cliente1.queryByExample(Coche.class));
        listRefrescaResult(cliente2, cliente2.queryByExample(Coche.class),
        2);
        cliente1.close();
        cliente2.close();
SALIDA:
        2
        BMW[David Coulthard/98]/0
        Ferrari[Michael Schumacher/100]/0
        2
        BMW[Rubens Barrichello/99]/0
        Ferrari[Michael Schumacher/100]/0
        2
        BMW[David Coulthard/98]/0
        Ferrari[Michael Schumacher/100]/0
        2
        BMW[David Coulthard/98]/0
        Ferrari[Michael Schumacher/100]/0
```

La operación rollbacks funciona como es de esperar:

```
        ObjectContainer cliente1 = server.openClient();
        ObjectContainer cliente2 = server.openClient();
        ObjectSet r = cliente1.queryByExample(new Coche("BMW"));
        Coche c1 = (Coche) r.next();
        c1.setPiloto(new Piloto("Fernando Alonso", 0));
        cliente1.store(c1);
        listResult(cliente1.queryByExample(new Coche(null)));
        listResult(cliente2.queryByExample(new Coche(null)));
        cliente1.rollback();
        cliente1.ext().refresh(Coche, 2);
        listResult(cliente1.queryByExample(new Coche(null)));
```

```
        listResult(cliente2.queryByExample(new Coche(null)));
        cliente1.close();
        cliente2.close();
SALIDA:
        2
        BMW[Fernando Alonso/0]/0
        Ferrari[Michael Schumacher/100]/0
        2
        BMW[David Coulthard/98]/0
        Ferrari[Michael Schumacher/100]/0
        2
        BMW[David Coulthard/98]/0
        Ferrari[Michael Schumacher/100]/0
        2
        BMW[David Coulthard/98]/0
        Ferrari[Michael Schumacher/100]/0
```

NETWORKING

Para usar la red, lo único que hace falta es indicar un puerto mayor que cero y configurar varias cuentas para los clientes.

```
        ObjectServer server = Db4oClientServer.openServer(
            Db4oClientServer.newServerConfiguration(), "F1.db4", 16000);
        server.grantAccess(USER, PASSWORD);
        try {
            ObjectContainer client = Db4oClientServer.openClient(
               Db4oClientServer.newClientConfiguration(), "localhost",
               PORT, USER, PASSWORD);
            // Hacer alguna operación ...
            client.close();
        } finally {
            server.close();
        }
```

El cliente se conecta indicando host, puerto, username y password.

```
        // consultar un servidor remoto
        ObjectContainer client =
                     Db4oClientServer.openClient(
                 Db4oClientServer.newClientConfiguration(),
                          "localhost", port, user, password);
```

```
                listResult(client.queryByExample(new Coche(null)));
                client.close();
SALIDA:
        2
        BMW[David Coulthard/98]/0
        Ferrari[Michael Schumacher/100]/0
```

Todas las operaciones se realizan de la misma forma que si estuvieses en modo embebido.

SEÑALIZAR FUERA DE BANDA

A veces un cliente necesita enviar un mensaje al servidor para realizar algo. Es decir, el servidor necesita ser avisado (señalizado) para que realice algo. Esto puede hacerse usando el método **messageRecipient()**, passando un objeto.

```
public void runServer() {
   synchronized(this) {
     ServerConfiguration config =
                  Db4oClientServer.newServerConfiguration();
     config.networking().messageRecipient(this);
     ObjectServer db4oServer = Db4oClientServer.openServer(config,
                                                  FILE, PORT);
     db4oServer.grantAccess(USER, PASS);
     Thread.currentThread().setName(this.getClass().getName());
     Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
     try {
         if (!stop) { this.wait(Long.MAX_VALUE); }
     } catch (Exception e) { e.printStackTrace(); }
     db4oServer.close();
   }
}
```

El mensaje es recibido y procesado por el método **processMessage()**:

```
public void processMessage(MessageContext con, Object message) {
    if (message instanceof StopServer) { close(); }
}
```

Db4o permite a un cliente enviar una señal arbitraria o un mensaje al servidor enviando un objeto.

```
public static void main(String[] args) {
   ObjectContainer oc = null;
   try {
       // conectar con el server
       oc = Db4oClientServer.openClient(
                        Db4oClientServer.newClientConfiguration(),
                        HOST, PORT, USER, PASS);
   } catch (Exception e) {
       e.printStackTrace();
   }
   if ( oc != null) {
       MessageSender emisor = oc.ext()
                                .configure()
                                .clientServer()
                                .getMessageSender();
       emisor.send( new StopServer() );
       oc.close();
   }
}
```

EJEMPLO COMPLETO DEL SERVIDOR

Vamos a poner todo junto, de forma que creamos un sencillo sevidor con un cliente que le puede pedir que se cierre graciosamente. La configuración la comparten a través de la interface ServerInfo.

```
package p11;

public interface ServerInfo {
    public String HOST = "localhost";
    public String FILE = "F1.db4";
    public int PORT = 4488;
    public String USER = "db4o";
    public String PASS = "db4o";
}
```

El server:

```
package p11;

import com.db4o.*;
import com.db4o.cs.*;
import com.db4o.cs.config.*;
import com.db4o.messaging.*;

public class StartServer implements ServerInfo, MessageRecipient {
    private boolean stop = false;

    public static void main(String[] arguments) {
        new StartServer().runServer();
    }

    public void runServer() {
        synchronized(this) {
           ServerConfiguration cfg =
                    Db4oClientServer.newServerConfiguration();
           cfg.networking().messageRecipient(this);
           ObjectServer db4oServer =
                    Db4oClientServer.openServer(cfg, FILE, PORT);
          db4oServer.grantAccess(USER, PASS);
        // Para identificar el thread en un debugger
        Thread.currentThread().setName(this.getClass().getName());
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
        try {
           if( !stop ) {
                this.wait(Long.MAX_VALUE);
           }
        } catch (Exception e) { e.printStackTrace(); }
        db4oServer.close();
    }
}

public void processMessage(MessageContext con, Object message) {
    if(message instanceof StopServer) { close(); }
}

public void close() {
    synchronized(this) { stop = true; this.notify(); }
}
}
```

El cliente que para al servidor:

```
package p11;

import com.db4o.*;
import com.db4o.cs.*;
import com.db4o.messaging.*;

public class StopServer implements ServerInfo {

    public static void main(String[] args) {
        ObjectContainer oc = null;
        try {
            oc = Db4oClientServer.openClient(
                  Db4oClientServer.newClientConfiguration(),
                  HOST, PORT, USER, PASS);
        } catch (Exception e) { e.printStackTrace(); }
        if( oc != null) {
            MessageSender enviador = oc.ext()
                                       .configure()
                                       .clientServer()
                                       .getMessageSender();
             enviador.send( new StopServer() );
             oc.close();
        }
    }
}
```

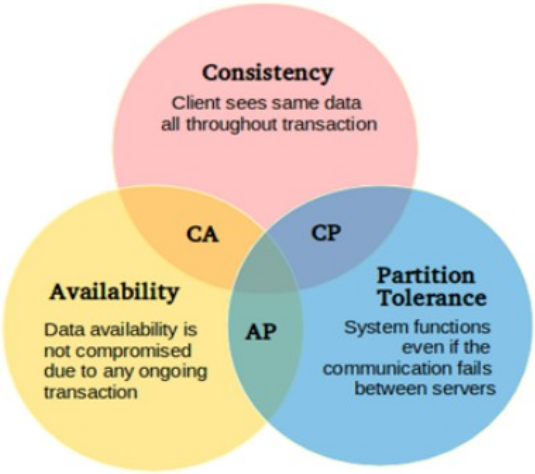
4. INTRODUCCIÓN A LOS SGBD NOSQL

Una BD relacional además de la eficiencia en la manipulación y almacenamiento de los datos, persigue protegerlos garantizando su calidad (consistencia, integridad y coherencia) ante fallos u operaciones. Suelen implementar las características conocidas como ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad).

Para dar soporte a las transacciones, frecuentemente bloquean parte de sus datos para que cada transacción al completarse los deje en un

estado consistente independientemente de que todos los cambios que realiza tengan éxito o falle miserablemente. Cualquier petición de cambio sobre estas partes bloqueadas son denegadas. Esta aproximación funciona bien mientras la cantidad de cambios por segundo que debe procesar el sistema sea asumible y la cantidad de datos manejable.

Sin embargo, este comportamiento (que garantiza la consistencia) compromete su disponibilidad en sistemas que deben recibir un gran flujo de de peticiones como los clusters de empresas como amazon.com. Por este motivo, las BDs se particionan (se usan varios servidores cada uno se ocupa de un trozo de la BD lógica) en varios grids, y el sistema debe funcionar incluso si la comunicación entre los distintos servidores se interrumpe.



El teorema ce CAP (CAP theorem), acuñado por Eric Brewer, aporta un conjunto de requerimientos que debe implementar una aplicación con arquitectura distribuida: Consistencia, Availability (disponibilidad) y

tolerancia al particionamiento (Partition tolerance). Argumenta que en la práctica, es imposible conseguir las 3 cosas en un entorno distribuido. En esta situación, **hay que escoger dos de ellas**, las que más interesen en cada momento.

Las BD no SQL abandonan la estricta implementación de la consistencia que aplican las BD relacionales para favorecer la facilidad de particionamiento o la eficiencia. Se pueden incluir en alguno de estos grupos:

- **Almacena grafos**: la información se almacena en estructuras similares a los grafos. Neo4J, HyperGraphDB, etc.
- **Almacenan parejas clave-valor**.
- **Almacenan información en columnas**: en vez de en filas: Cassandra, Hbase, etc.
- **Basadas en documentos**: crean estructuras de datos complejas llamadas documentos: MongoDB, CouchDB, etc.

Y todas suelen diferenciarse de las relacionales en que tienen:

- **Formato Neutral**: representan información en formatos distintos.
- **Sin Joins**: puede prescindirse de las operaciones join de SQL.
- **Sin esquemas fijos**: hay mucha libertad a la hora de definir la información que guardan.

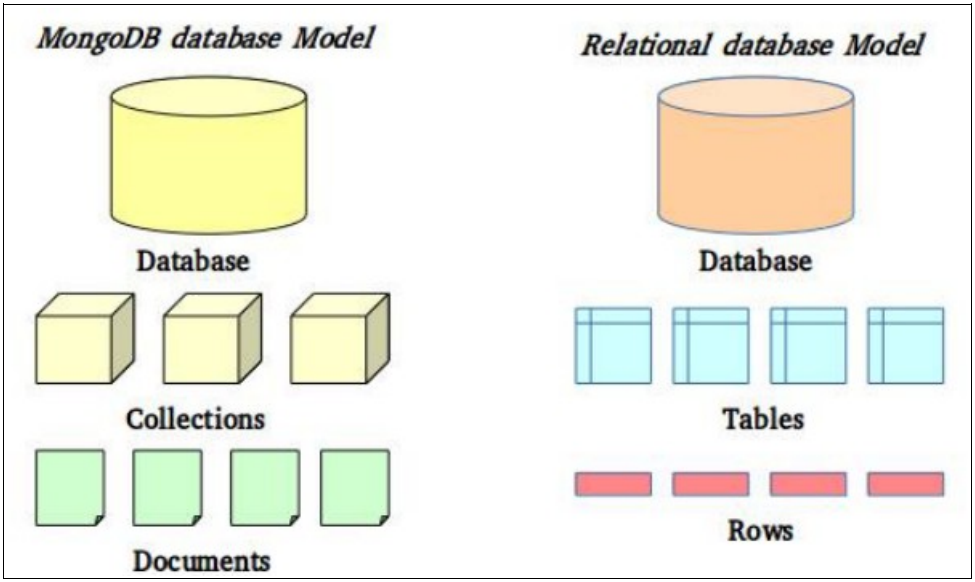
4.1. CARACTERÍSTICAS DE MONGO.

MongoDB es una BD orientada a documentos, multiplataforma, que persigue el alto rendimiento y disponibilidad, fácilmente escalable. Utiliza los siguientes conceptos de almacenamiento:

- **Database**: un contenedor físico de colecciones, un conjunto de ficheros de disco. Cada servidor puede ocuparse de varias

bases de datos.

- **Colección**: un grupo de documentos. Sería el equivalente de una tabla del modelo relacional. Cada colección se almacena en una BD. No fuerzan un esquema concreto.
- **Documento**: un conjunto de parejas clave-valor sin un esquema predefinido. Sería el equivalente a una fila de una tabla relacional.



EJEMPLO : documento usado en MongoDB. Se usa JSON (javascript):

```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by: 'tutorials',
   url: 'http://www.unaurl.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100,
```

```
   comentarios: [
      {
         user:'user1',
         message: 'My first comment',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
      },
      {
         user:'user2',
         message: 'My second comments',
         dateCreated: new Date(2011,1,25,7,45),
         like: 5
      }
   ]
}
```

_id es un número hexadecimal de 12 bytes que garantiza la unicidad de cada documento. Cuando creas uno puedes aportarlo, y si no lo haces se genera de forma automática: 4 bytes un timestamp, 3 bytes un id de la máquina, 2 bytes un id del proceso servidor Mongo y 3 bytes un entero que se va incrementando. Es como la clave primaria de una fila del modelo relacional.

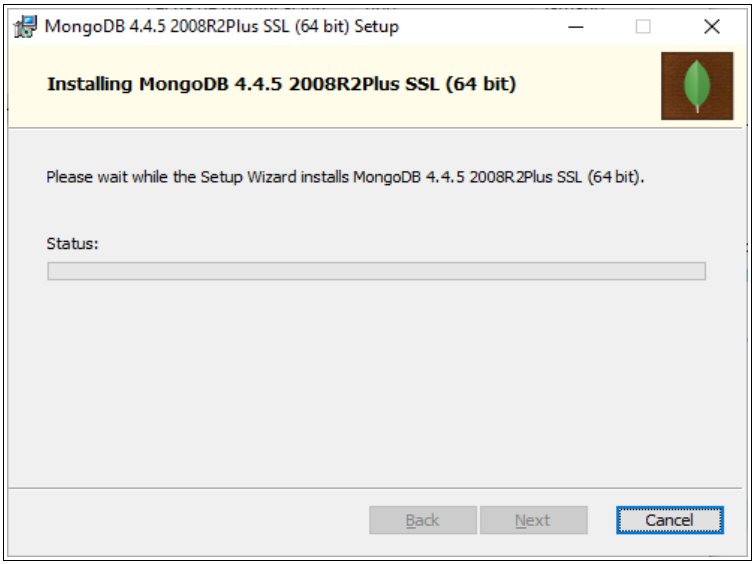
5. USO DEL SGBD NOSQL MONGODB.

5.1 INSTALACIÓN Y PRIMEROS PASOS.

En la página de descargas (http://www.mongodb.org/downloads) tienes los enlaces. En el manual que te puedes descargar del mismo sitio tienes instrucciones detalladas de la instalación en diferentes plataformas. Comentamos la de Windows.

Una vez descargado el archivo .msi adecuado para tu hardware, solo hay que ejecutar el instalador y elegir en Custom para indicarle la carpeta c:\mongo y cuando pida "Choose Setup Type", escoge "Tipical", que es la recomendada para la mayoría de usuarios.

Por defecto usa la carpeta **c:\data\db** para almacenar datos, pero la tenemos que crear nosotros. Si queremos que utilice otra, debemos indicarlo cuando ejecutamos el servicio, con el parámetro **--dbpath**

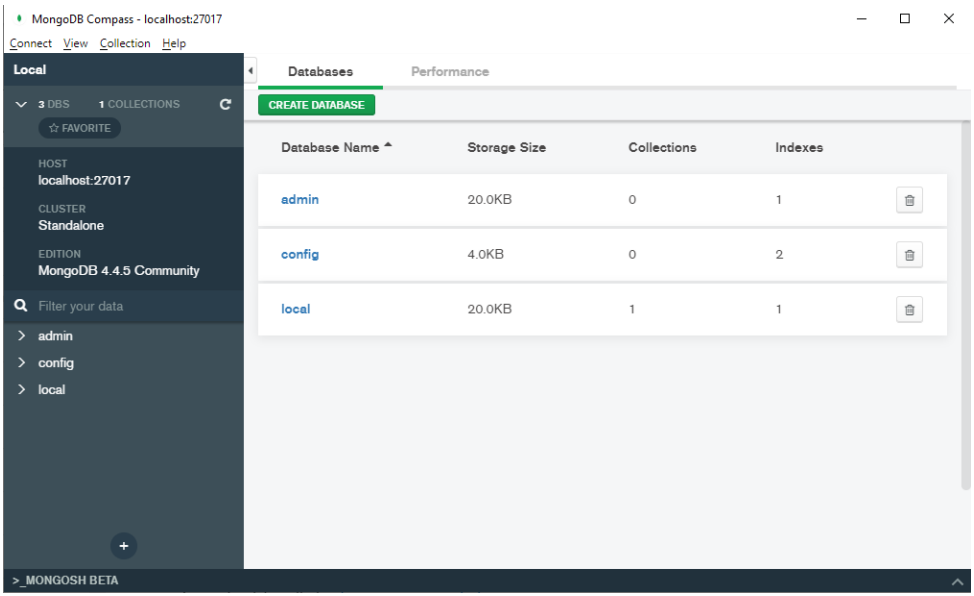


Todos los ejecutables están en la carpeta bin de la carpeta de la instalación. Los programas son:

- **bsondump**, lee los ficheros BSON.
- **Mongo**, es un shell, un cliente en modo texto.
- **Mongod**, el motor de base de datos (el servidor).
- **mongodump** y **mongorestore**, utilidades de backup y restauración.
- **mongoimport** y **mongoexport**, utilidades de importación / exportación (JSON, CSV y TSV).
- **mongofiles** manipula ficheros binarios almacenados como objetos GridFS.

- **mongooplog** rellena entradas en el log para otras instancias de servidor.
- **mongoperf** comprueba el rendimiento de las E/S en disco.
- **mongos** procesa los shard de MongoDB.
- **mongostat** y **mongotop** devuelve e informa de las operaciones de BD.

Esta es una imagen de compass, una herramienta de administración y gestión de MongoDB.



INICIAR EL SERVIDOR

Los GNU/Linux usan scripts en /etc/init.d para controlar servicios. Ubuntu, Fedora, CentOS y RedHat usan start, stop y restart como en el ejemplo:

```
$ sudo service mongodb start
```

```
$ sudo service mongodb stop
$ sudo service mongodb restart
```

Si no tienes los scripts debes iniciarlo de forma manual o con un script:

```
$ mongod
```

Sin indicar opciones, el servidor intenta usar la carpeta /data/db y el puerto 27017. MongoDB tiene 3 formas de configurar el servidor: la línea de comandos, un fichero de configuración y el comando setParameter. Ej:

```
> db.adminCommand( {setParameter:1, logLevel:0} )
```

Muchos de los instaladores usan el segundo método usando el fichero **mongodb.conf**. Los contenidos del fichero pueden parecerse a la figura del recuadro. Colocar --<opción> <valor> en la línea de comandos tiene opciones equivalentes en el fichero. Algunas opciones:

```
# mongodb.conf
storage:
dbPath: /var/lib/mongodb
Journal:
enabled: true
# where to write logging data.
systemLog:
destination: file
logAppend: true
path: /var/log/mongodb/mongod.log
```

```
dbpath: ruta a los datos.
logpath: ruta a los logs.
logappend: a false limpia los log al iniciar.
auth: a true se activa la autenticación.
rest: activa/desactiva otras interfaces con el servidor.
```

Para ejecutar el servidor de manera manual, abre una consola y ejecuta (la ruta a los ejecutables debe estar añadida a la variable de entorno PATH, si no, debes prefijar los ejecutables con su ruta) :

```
mongod --dbpath c:\data\db
```

Debes dejar la consola abierta y cuando quieras parar el servidor,

pulsas control+c. Si el servidor, pese a dar warnings al final escribe este mensaje, es que está esperando a recibir peticiones:

```
[initandlisten] waiting for connections on port 27017
```

Al llamarlo, podemos indicar muchos parámetros, o esos parámetros podemos escribirlos en un fichero de configuración.

En el recuadro tenemos algunas formas de controlar la ejecución del servidor:

```
# En Windows
- Instalar como servicio indicando fic. de config.
mongod.exe --config C:\mongo\mongod.cfg" --install
- Instalar varias instancias dando a cada una nombre único de servicio, con la opción
--serviceName y --serviceDisplayName.
- Iniciar el servicio MongoDB.
net start MongoDB
- Parar el servicio MongoDB.
net stop MongoDB
- Eliminar servicio MongoDB:
mongod.exe --remove
- Servicio que se ejecute autom. al arrancar sistema:
sc.exe create MongoDB binPath= "\"C:\mongo\bin\mongod.exe\"
--service --config=\"C:\mongo\mongod.cfg\"" DisplayName= "MongoDB"
start= "auto"
NOTA: c.exe necesita espacios entre el "=" y los valores de
configuración y escapar de las dobles comillas con \".
- Detener el servicio creado con sc:
net stop MongoDB
- Eliminar servicio creado con sc:
sc.exe delete MongoDB
```

CONECTARNOS CON UN CLIENTE

Si te funciona el servidor, ahora vamos a ejecutar en otra ventana de comandos el cliente. Cuando lo llames, si intentas conectarte a una BD que no existe, él la crea automáticamente. Esto puede ser una ventaja

o un inconveniente según lo hábil que seas con el teclado. En la nueva línea de comandos, ejecuta el cliente: **mongo.exe** y prueba esto:

Almacena un documento en la colección llamada ejemplo.

```
> db.ejemplo.save( { msg:"Hola Mundo"} )
WriteResult({ "nInserted":1 })
```

Ahora lee un documento de la colección recién almacenada y comprueba si tiene _id:

```
> db.ejemplo.findOne()
```

El shell que hemos lanzado se comunica con el servidor en JavaScript, para comprobarlo, vamos a ejecutar algunos comandos sencillos.

```
> var a=200;
> a
200
> var b=a/5;
> b
```

También podemos utilizar librerías JavaScript estándar o definir nuestras propias funciones.

```
> Math.sqrt(b)
6.324555320336759
> new Date("2014/1/10")
ISODate("2014-01-09T23:00:00Z")
> function divBy2(n) { return n/2; }
> divBy2(18)
9
```

Aquí tienes algunos comandos útiles:

```
show dbs muestra los nombres de las BD disponibles en el servidor.
show collections muestra las colecciones de la actual BD.
show users muestra los usuarios de la BD actual.
use <nombre db> establece como BD actual a <nombre db>.
Help muestra una lista de todos los comandos.
```

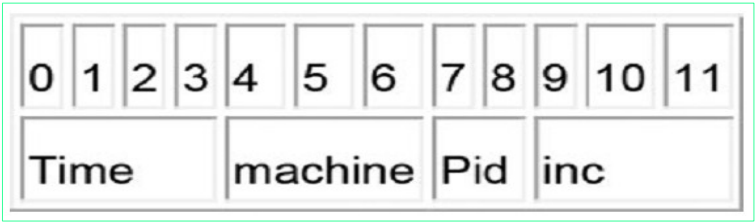
5.2 MODELO DE DATOS.

Cada documento dentro de una BD de MongoDB tiene una clave que es el identificador que diferencia ese documento de cualquier otro, se llama **_id** y se añade automáticamente a cualquier documento que crees en una colección.

Su valor debe ser único en cada colección, inmutable y de cualquier tipo salvo array. Es el primer atributo de cada nuevo documento (elemento obligatorio).

Si no indicas ningún valor de forma manual, se le asigna un valor de tipo **ObjectId** que es un valor binario de 12 bytes, diseñado para que haya una alta probabilidad de que sea único:

- 4 bytes con un timestamp (cuando se crea) con segundos desde el 1 de enero de 1970.
- 3 bytes de un ID de la máquina.
- 2 bytes con el ID del proceso.
- 3 bytes con un contador. Tanto este como el timestamp se almacenan en formato *Big Endian* para asegurarse de que los valores van creciendo.



Nota: *Cada driver que se conecte a MongoDB (PHP, Python, java, etc.) soporta este tipo de dato que puede usar al crear nuevos documentos.*

Nota: *puedes usar la función* ***ObjectId()*** *desde el shell de*

MongoDB para crear un valor o indicar en una cadena hexadecimal el valor: ***ObjectId(string)***

MongoDB usa notación punteada para acceder a cualquier parte de un documento:

```
un_documento.visitas
```

Para acceder a los elementos de un array con la posición del índice comenzando por 0 para el primer elemento, concatenan el nombre del array con el punto y la posición del elemento encerrado entre comillas: **'<array>.<index>'**. Ejemplo:

```
un_documento.obras.0
```

Para acceder a un campo de un subdocumento embebido dentro de otro se usa el nombre del subdocumento en el documento encerrado entre comillas: **'<subdocumento>.<campo>'**. Ejemplo:

```
un_documento.nombre.ape1
```

RELACIONES DE UNOS DOCUMENTOS CON OTROS

El desafío clave a la hora de modelar datos es mantener un equilibrio entre las necesidades de la aplicación, el rendimiento del motor de BD y los patrones de recuperación de datos.

Siempre hay que tener en cuenta como usa la aplicación los datos (consultas y modificaciones) además de la estructura de los datos.

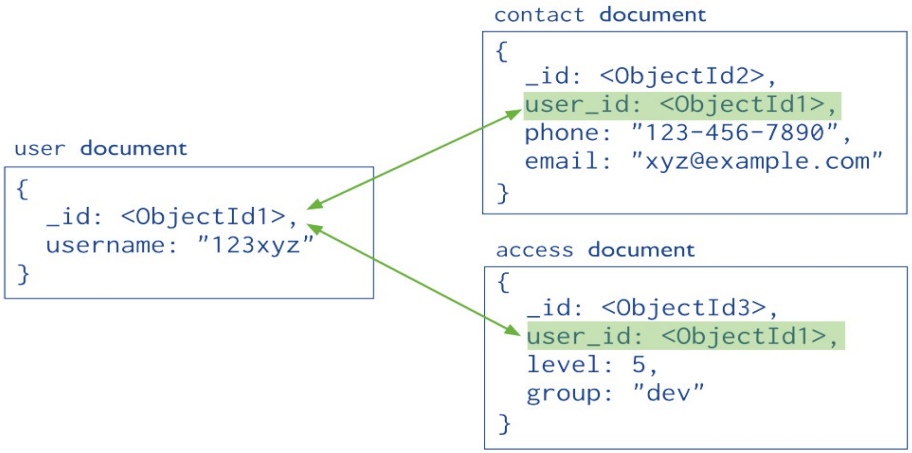
En MongoDB lo fundamental es pensar como utilizar la estructura documento y como van a representar las aplicaciones las relaciones entre los datos. Hay dos mecanismos para hacerlo: **usar referencias** (como SQL) o **embeber documentos** dentro de otros.
Referencias: representa la relación de un dato con otro, almacenando

un enlace o referencia al dato con el que se relacionan. Si se utiliza, se llama **modelo de datos normalizado**.

Cuando conviene usar modelos de datos normalizados:

- Si al embeber puedes duplicar datos, creando redundancias y no tienes suficiente ganancia de rendimiento que justifique los problemas que pueden dar esas redundancias.
- Debes representar relaciones complejas como las de cardinalidad muchos a muchos.
- Debes modelar grandes modelos con relaciones jerárquicas (en árbol).

Aunque las referencias son más flexibles que embeber, obligan a las aplicaciones del lado cliente a resolver las referencias para acceder a los datos → **más operaciones contra el servidor**.



Los modelos de datos embebidos (**modelo de datos desnormalizado**) permiten a las aplicaciones almacenar trozos de información más complejos en la misma unidad de datos. Una consecuencia de ello es que las aplicaciones necesitan **menos operaciones** para realizar el

mismo trabajo que con un modelo normalizado. En general, **se usan modelos embebidos cuando**:

- Tienes relaciones de datos del tipo "contiene".
- Tienes relaciones con cardinalidad uno a muchos. En estas relaciones los documentos hijos (la parte muchos) pueden embeber a su único documento padre.

En general, embeber ofrece mayor rendimiento en las operaciones de lectura y permiten modificar información de entidades y entidades relacionadas en operaciones atómicas.

En MongoDB, l**as operaciones de escritura son atómicas a nivel de documento** y si modificas varios documentos en la misma o diferentes colecciones, esas operaciones no serán atómicas. Por eso en un modelo desnormalizado, los cambios son atómicos pero en un modelo normalizado (usando referencias) no lo son de forma colectiva.

Por contra, **si después de crearlos, los documentos embebidos**

crecen, pueden afectar negativamente al rendimiento de las operaciones de escritura (cambios) y generar fragmentación de datos.

5.3 SENTENCIAS PARA TRABAJAR CON DATOS.

INSERTAR DOCUMENTOS

La función **insertOne()** añade un documento a una colección. La función **insert()** inserta uno o varios documentos y **insertMany()** inserta los documentos que hay en el array que se le pasa. Por ejemplo, si queremos añadir un comentario a la BD de nuestro blog, podemos pasar el objeto o guardarlo antes en una variable. Ojo con las claves que son case-sensitive.

```
> un_post = {"titulo":"Mi post", "contenido":"Hola mundo",
"fecha":new Date() }
```

Y ahora lo insertamos en una colección de nombre blog (si no existe, la crea).

```
> db.blog.insert(un_post)
WriteResult( { "nInserted" : 1 } )
```

Ejemplos:

```
> db.media.insertOne( {tipo:"CD", artista:"Nirvana",
titulo:"Nevermind",
... pistas: [
... {pista:1,
... titulo: "Smells Like Teen Spirit",
... duracion:"5:02"},
... {pista:2,
... titulo: "In Bloom",
... duracion:"4:15"}
... ]
... }
... )
WriteResult( {"nInserted":1} )

> db.media.insertOne( {tipo:"LIBRO", titulo:"AFRICANUS, El hijo
```

```
del consul", Autores:["Santiago Posteguillo"],
isbn:"9788466639323", editorial:"Ediciones B",  paginas:720,
publicado:2006} )
```

CONSULTAR DOCUMENTOS

Las funciones **find()** y **findOne()** se utilizan para consultar una colección. La diferencia es que findOne solamente devuelve un documento y find puede devolver todos. Vamos a recuperar el primer documento de la colección media.

```
> db.media.findOne()
```

Ahora consultamos todos los documentos:

```
> db.media.find()
```

Si no quieres ver todos los documentos, solamente los que cumplan alguna condición, pasas la condición como primer parámetro (documento). Por ejemplo, solo quiero ver los de tipo "CD":

```
> db.media.find( {tipo:"CD"} )
```

Si no quieres ver todos los campos del documento, puedes activar (1) o desactivar (0) los campos que quieres ver. El campo _id siempre se ve a no ser que lo desactives. Ej: solo titulo de CDs:

```
> db.media.find( {tipo:"CD"}, {titulo:1, _id:0} )
```

Si no quieres ver todos los campos pero si todos los documentos, pasa el vacío el criterio:

```
> db.media.find( {}, {titulo:1, _id:0} )
```

Si quieres ver todos los campos menos algunos, puedes desactivar los que no quieres:

```
> db.media.find( {}, {titulo:0, _id:0} )
```

Mejora la visualización de los resultados con **pretty()**.

```
> db.media.find().pretty()
```

Acceder a claves de documentos embebidos (hay que entrecomillar):

```
> db.media.find( {"pistas.titulo":"In Bloom"} ).pretty()
```

La función **sort()** se utiliza para ordenar los resultados. Pasas un documento con las claves que se usan para ordenar con un 1 (ascendente) o -1 (descendente).

```
> db.media.find().sort( {titulo:1} )
> db.media.find().sort( {titulo:-1} )
```

La función **limit()** limita la cantidad de resultados devueltos:

```
> db.media.find({}, {_id:0, titulo:1} ).limit(1)
```

La función **skip()** se utiliza para no mostrar cierto número de documentos. Ejemplo:

```
> db.media.find().skip(1)
```

Estas funciones pueden combinarse entre si para implementar un mecanismo de paginación, por ejemplo haciendo páginas de 10 documentos, si queremos ver la tercera página (documento del 21 al 30:

```
> db.media.find().sort( {titulo:-1} ).skip(20).limit(10)
```

En cuanto al orden hay que definir varios conceptos:

- Orden natural: orden en que se recuperan los documentos según estén almacenados (no es el orden en que se intertan, depende de índices, motor de almacenamiento usado, …)
- Capped collection: colecciones que garantizan que el orden natural es el orden en que se insertan los documentos. Ideales

para tener datos de auditoría. Tienen tamaño fijo, si se sobrepasa se sobreescriben los más antiguos para mantener el orden natural. Hay que crearlas de forma explícita, :

```
> db.createCollection( "login", {capped:true, size:20480} )
> db.login.find().sort( { $natural:-1 } ).limit( 10 )
```

FUNCIONES AGREGADAS

La función **count()** se utiliza para contar documentos del resultado (es como el count(*) de SQL). No le afecta skip() ni limit(), salvo que indiques que le afecte (parámetro a true).

```
> db.media.find().count()
> db.media.find().skip(1).count()
> db.media.find().skip(1).count(true)
```

La función **distinct()** elimina documentos que tengan valores repetidos de una clave:

```
> db.media.find().distinct("titulo")
```

La función **group()** devuelve un array de elementos agrupados. Tiene 3 parámetros obligatorios:

- Key: indica por qué criterios quieres hacer los grupos.
- Initial: es el valor numérico que sirve de base para cada grupo, normalmente cero.
- Reduce: agrupa juntos todos los documentos similares. Acepta dos parámetros:
  - Items: El documento actual.
  - El contador: va contando los objetos del grupo.

Esta función tiene limitaciones (no funciona en entornos donde se hace sharding, máximo de 20 mil claves por grupo) por eso se prefiere usar la función mapReduce(). Ejemplo:

```
> db.media.group (  { key: {titulo:true},
                      initial: {total: 0},
                      reduce : function (items,prev)
                               { prev.total+= 1 }
                    } )
```

Tiene además 3 parámetros opcionales:

- keyf: para cambiar la clave por la que agrupas si los documentos no tienen las que necesitas.
- cond: indicas condición que debe cumplirse para agrupar un documento (como find()).
- finalize: indicas una función que se ejecuta antes de devolver los resultados.

OPERADORES CONDICIONALES

Tenemos las siguientes operaciones de comparación si no es la igualdad lo que nos interesa:

- clave: {$ne: valor} true si la clave es diferente de valor.
- clave: {$lt: valor} (less than), la condición < (menor estricto).
- clave: {$lte: valor} (less than or equal), la condición <= (menor o igual).
- clave: {$gt: valor} (greather than), condición > (mayor estricto).
- clave: {$gte: valor} (greather than or equal), la condición >= (mayor o igual).

La sintaxis es: **clave: { $operador: valor }**

Añade estos datos a la colección media:

```
{tipo:"DVD", titulo:"Matrix, The", publicado:1999, Casting:
["Keanu Reeves", "Carrie-Anne Moss", "Laurence Fishburne", "Hugo
Weaving", "Gloria Foster", "Joe Pantoliano"] }
```

```
{tipo:"DVD", titulo:"Blade Runner", publicado:1982 }

{tipo:"DVD", titulo:"Toy Story 3", publicado:2010 } )
```

Ejemplos de uso:

```
> db.media.find ( { Released : {$gt : 2000} }, { "Cast" : 0 } )
> db.media.find ( { Released : {$gte : 1999 } }, { "Cast" : 0 } )
> db.media.find ( { Released : {$lt : 1999 } }, { "Cast" : 0 } )
> db.media.find( {Released : {$lte: 1999}}, { "Cast" : 0 })
```

Puedes combinarlos para buscar por rangos (al separar por una coma dentro es como un AND):

```
> db.media.find( {Released: {$gte:1990, $lt: 2010}}, {"Cast": 0})
```

OPERADORES CON ARRAYS

- clave: {$in: […] } true si clave es uno de los elementos del array.
- clave: {$nin: […] } true si clave no es uno de los elementos del array.
- clave: {$all […]} true si clave es igual a todos los elementos del array.
-

OPERADOR $or

Aplicado a la misma clave, es mejor usar $in. Pero si queremos aplicarlo entre diferentes claves:

```
$or: [ {clave1:valor1}, {clave2:valor2}, {claveN:valorN} ]
```

OPERADOR $not

Su sintaxis es sencilla, solo hay que ponerlo "fuera" de aquello que queremos negar.

```
$not: { criterio }
```

Podemos recuperar por ejemplo, documentos que no tengan 480 páginas:

```
> db.media.find( {paginas:{$not:{$eq:480}}},
{titulo:1,paginas:1} ).pretty()
```

OPERADOR $exists

Se utiliza para comprobar si los documentos tienen o no, una clave determinada.

```
clave: { $exists: boolean }
```

Si boolean es true devuelve documentos que tienen informada la clave (aunque sea null). Es útil porque al no haber esquema de datos, cada documento puede tener campos diferentes. No es equivalente a la función de SQL. Ejemplo:

```
> db.media.find( {publicado:{$exists:true}}, {casting:0} )
```

RECUPERAR PARTE DE UN ARRAY $slice

Recuperas solo parte del array. Tiene dos parámetros: el nº de elementos devueltos y el segundo opcional, pero que si se usa se escribe en primer lugar, informa del desplazamiento, desde donde se comienza a recuperar elementos. Si se pasan valores negativos, se entiende que se usa el final del array como origen. Ej:

```
> db.media.find({titulo:"Matrix, The"}, {casting: {$slice: 3} })
> db.media.find({titulo:"Matrix, The"}, {casting: {$slice: -3} })
> db.media.find({titulo:"Matrix, The"}, {casting: {$slice:
[2,3]} })
```

RESTO ($mod).

Calcula el resto de dividir la clave por el primer parámetro, y si coincide con el segundo, es true.

```
> db.media.find( {publicado:{$mod:[2,0]}}, {casting:0} )
```

USAR EL TAMAÑO ($size)

Devuelve true si el nº de elementos de la clave coincide.

```
> db.media.find( {pistas:{$size:2}} )
```

USAR EL TIPO DE DATO ($type)

Según el tipo de dato BSON de una clave.

```
> db.media.find( {pistas:{$type:3}} )
```

La lista de tipos: –1 MinKey, 1 Double, 2 Character string (UTF8), 3 Embedded object, 4 Embedded array, 5 Binary data, 7 Object ID, 8 Boolean type, 9 Date type, 10 Null type, 11 Regular expression, 13 JavaScript code, 14 Symbol, 15 JavaScript code with scope, 16 32-bit integer, 17 Timestamp, 18 64-bit integer, 127 MaxKey, 255 MinKey

COINCIDENCIAS COMPLETAS EN UN ARRAY ($elemMatch).

Recuperas solo documentos donde todas las partes que indicas coincidan. Ej:

```
> nirvana = ( {tipo:"CD", artista:"Nirvana", titulo:"Nirvana",
pistas:[ {pista:"1", titulo:"You Know You're Right",
minutos:"3:38"}, {pista:"5",titulo:"Smells Like Teen Spirit",
minutos:"5:02"} ] } )
> db.media.insertOne(nirvana)
```

Si quieres recuperar CD's que tengan la canción "Smells Like Teen Spirit" en la pista 1 y ejecutas:

```
> db.media.find( {"pistas.titulo":"Smells Like Teen Spirit",
"pistas.pista":"1"} )
```

Devuelve 2 documentos, el motivo es que los dos tienen una pista con el título buscado y los dos tienen una pista 1, pero lo que necesitas es que ambas cosas se den a la vez en el cada docum.:

```
> db.media.find( {pistas:{"$elemMatch":{titulo:"Smells Like Teen
Spirit", pista:"1"} } } )
```

EXPRESIONES REGULARES.

MongoDB las tiene forma nativa. Siguen la sintaxis de Perl. Vemos ejemplos para recuperar libros del autor Posteguillo sin considerar mayúsculas/minúsculas y títulos con la frase 'juego de':

```
> db.libro.find({autor:/posteguillo/i} ,
{titulo:1,autor:1}).pretty()
> db.libro.find({titulo:/juego de/i},{titulo:1,autor:1}).pretty()
```

EXPRESIONES JAVASCRIPT ($where).

Usar expresiones javascript (flexible pero ineficiente):

```
> db.media.find( {tipo:"DVD", $where:"this.publicado < 1995"} )
```

ACTUALIZAR DATOS CON update() Y updateOne()

Tiene 3 parámetros principales: criterio, objNew y options.

- criteria selecciona los documentos a modificar.
- ObjNew: indica la información modificada, o usas un operador.
- options tiene dos valores posibles:
  - upsert indica que si el documento no existe no lo crea y si existe, lo modifica.
  - multi controla si se actualizan todos o solo el primero de los que coincidan.

```
> db.media.updateOne( {titulo:"Matrix, The"},
{tipo:"DVD",titulo:"Matrix, The", publicado: 1999,
genero:"Acción"}, { upsert: true} )
```

```
> db.media.updateMany( titulo:"Matrix, The"}, {$set:{tipo:"DVD",
titulo:"Matrix, The", publicado:1999, genero:accion} }, {upsert:
true} )
```

ACTUALIZAR DATOS CON save()

Debes indicar un _id del documento y entonces entiende que es una inserción, si no lo pasas, entiende que es una actualización.

```
> db.media.save( {titulo:"Matrix, The"}, {tipo:"DVD",
titulo:"Matrix, The", publicado:"1999", genero:"acción"})
```

ACTUALIZAR DATOS. OPERADORES.

INCREMENTAR UN VALOR ($inc).

Incrementa una clave en una cantidad si la clave existe. Si no existe, la crea.

```
> manga = ( {tipo:"Manga", titulo:"One Piece", volumenes:612,
leido:520 } )
> db.media.insertOne(manga)
> db.media.updateOne( {titulo:"One Piece"}, {$inc:{leido:4}} )
```

CAMBIAR UN CAMPO ($set, $set:{campo.<posicion>.campo}).

Se utiliza el operador para cambiar el valor de un campo, o elemento de campo array.

```
> db.media.update( {titulo:"Matrix, The"}, {$set:{genero:"ciencia
ficción"}})
```

BORRAR UN CAMPO ($unset).

Borra el valor del campo y la clave que indiques del documento.

```
> db.media.updateOne ( {titulo:"Matrix, The"}, {$unset:
{genero:1} } )
```

AÑADIR UN VALOR A UN ARRAY ($push).

Si el campo existe se añade el valor, si no existe se crea con el valor, si no es array se genera error.

```
> otro_libro = ( {autores:["Hows, David","Membrey, Peter",
"Plugge, Eelco"], ISBN:"978-1-4842-1183-0", editorial:"Apress",
titulo:"Definitive Guide to MongoDB 3ed, The", tipo:"LIBRO" } )
```

```
> db.media.insertOne( otro_libro )
> db.media.updateOne({ISBN:"978-1-4842-1183-0"},{$push:
{autor:"Griffin, Stewie"} } )
```

AÑADIR VARIOS VALORES A UN ARRAY ($push y $each).
Se usa el modificador $each par añadir varios. Puedes usar $slice para limitar los elementos.

```
> db.media.updateOne({ISBN:"978-1-4842-1183-0"},{$push:{autor:
{$each:["Griffin, Peter", "Griffin, Brian"], $slice: -2 } } } )
AÑADIR DATOS A UN ARRAY SI NO ESTÁN ($addToSet).
Se usa como $push, pero no añade repetidos.
> db.media.updateOne( {ISBN:"978-1-4842-1183-0"}, {$addToSet:
{autor:"Griffin, Brian"} } )
> db.media.updateOne( {ISBN:"978-1-4842-1183-0"}, {$addToSet:
{autor:{$each:["Griffin, Brian","Griffin, Meg"] } } } )
```

ELIMINAR ELEMENTOS DE UN ARRAY ($pop, $pull y $pullAll).
$pop elimina el último elemento (si pasas 1) o el primero (si indicas -1).

```
> db.media.updateOne( {ISBN:"978-1-4842-1183-0"}, {$pop:{autor: 1}
}
> db.media.updateOne( {ISBN:"978-1-4842-1183-0"}, {$pop:{autor:
-1} } )
```

$pull borra todas las ocurrencias de un valor en el array.

```
> db.media.updateOne( {ISBN:"978-1-4842-1183-0"}, {$pull:
{autor:"Griffin, Stewie" } } )
```

$pullAll usa un array con los elementos que quieres eliminar del otro.

```
> db.media.updateOne( {ISBN:"978-1-4842-1183-0"}, {$pullAll:
{autor:["Griffin, Louis","Griffin, Peter","Griffin, Brian"] } } )
```

USAR LA POSICIÓN DE LOS ELEMENTOS (clave1.$.clave2).
El $ antes de una clave de tipo array simboliza la posición del elemento del array. Puedes usarlo por ejemplo para cambiar un valor de clave2 del elemento que cumpla un criterio.

```
> db.media.updateOne({"pistas.titulo":"Been a Son"},{$inc:
{"pistas.$.pista":1 } } )
```

MODIFICAR Y DEVOLVER UN DOCUMENTO

findAndModify(<query>,<sort>,<operations>).

Modifica un documento de forma atómica y lo devuelve. Tiene 3 parámetros:

- <query>: selecciona el documento
- <sort>: cuando varios se seleccionan, los ordena.
- <operations>: indica la modificación.

Ejemplos: eliminar el documento y devolver el anterior (antes del cambio), modificar el título y devolver el anterior, y modificar el título y devolver el nuevo documento (después del cambio).

```
> db.media.findAndModify( {titulo:"One Piece",sort:{titulo:-1},
remove:true} )
> db.media.findAndModify( {query:{ISBN:"978-1-4842-1183-0" },
sort:{titulo:-1}, update: {$set: {titulo:"otro titulo"} } } )
> db.media.findAndModify( {query:{ISBN:"978-1-4842-1183-0"}, sort:
{titulo:-1}, update:{$set: {titulo:"Otro"} }, new:true } )
```

OPERACIONES EN LOTES (BULK)

Primero defines una variable que almacene las sentencias a ejecutar. Luego las aplicas y por último puedes comprobar el resultado. Hay dos formas de definir las operaciones BULK:

- initializeOrderedBulkOp(): sentencias ordenadas. Se realizan en cierto orden, si alguna falla, el resto deja de realizarse.
- initializeUnorderedBulk(): Sentencias desordenadas: se realizan en paralelo, si una falla, no afectan a las demás.

```
> var bulk = db.media.initializeOrderedBulkOp();
> bulk.insertOne({tipo:"PELICULA", titulo:"Deadpool",
```

```
publicado:2016} );
> bulk.insertOne({tipo:"CD",artista:"Iron Maiden", titulo:"Book of
Souls" });
> bulk.execute();
> bulk.getOperations(); // BatchType es 1(insert), 2(update),
3(remove)
```

CAMBIAR EL NOMBRE A UNA COLECCIÓN:

renameCollection("nuevo_nombre")

```
> db.media.renameCollection( "tienda" )
```

BORRAR DOCUMENTOS removeOne() Y removeMany()

Recuerda que no hay claves ajenas, por tanto, las referencias a los documentos que borres quedan almacenadas. Debes eliminarlas de forma manual. Ambas usan un criterio para seleccionar lo que hay que borrar.

```
> db.media.removeOne( {titulo:"Matrix, The"} )
> db.media.remove( {} )
```

BORRAR COLECCIONES drop()

Es más rápido que borrar los documentos con remove(), porque no actualiza índices.

```
> db.media.drop()
```

BORRAR BASES DE DATOS dropDatabase()

```
> db.dropDatabase()
```

BORRAR DOCUMENTOS removeOne() Y removeMany()

Recuerda que no hay claves ajenas, por tanto, las referencias a los documentos que borres quedan almacenadas. Debes eliminarlas de forma manual. Ambas usan un criterio para seleccionar lo que hay que borrar.

```
> db.media.removeOne( {titulo:"Matrix, The"} )
> db.media.remove( {} )
```

REFERENCIAR DOCUMENTOS.

DE FORMA MANUAL

La forma más sencilla es usar el _id de otro documento u otra clave que lo identifique. Referencia implica hacer otra lectura para recuperar la información referenciada y que las operaciones no sean atómicas.

```
> apress = ({_id:"Apress", tipo:"editorial técnica", categorias:
["IT", "Software", "Programación"] } )
> db.editorial.insertOne( apress )
> libro = ( {tipo:"LIBRO", titulo:"Definitive Guide to MongoDB
3ed., The", ISBN:"978-1-4842-1183-0", edotorial:"Apress", autores:
["Hows, David","Plugge, Eelco","Membrey,Peter","Hawkins, Tim"] } )
> db.media.insertOne( libro )
> libro = db.media.findOne( {ISBN:"978-1-4842-1183-0"} )
> db.editorial.findOne( {_id:libro.editorial} )
```

CON DBREF: {$ref : <colección>, $id : <valor>[,$db : <BD>] }

La ventaja de usar DBRef es que los documentos pueden cambiar, si tus documentos referenciados no cambian, referenciar de forma manual es apropiado. Si usas DBRef, la referencia se almacena en la BD. Los drivers se aprovechan de esto para automatizar accesos. Ejemplos:

```
> db.editorial.drop()
> db.media.drop()
> apre= ({_id:"Apress", tipo:"editorial técnica",categorias:
["IT","Soft"]})
> db.editorial.save( apre )
> libro = ({tipo:"LIBRO", titulo:"Guide to MongoDB", ISBN:"978-1-
4842-1183-0", editorial:[new DBRef('editorial',apres._id)],
autores:["Hows, David"]} )
> db.media.save( libro )
```

INDEXAR DATOS.

CREACIÓN Y USO

Podemos crear índices con **createIndex()** y **ensureIndex()** es un sinónimo en desuso. Hay que pasar de parámetro una clave y en qué orden: (1) ascendente y (-1) descendente.

```
> db.media.createIndex( {titulo:1} )
```

También se pueden crear índices de arrays.

```
> db.media.createIndex( {"pistas.titulo":1} )
```

También puedes usar todo el subdocumento.

```
> db.media.createIndex( {"pistas":1} )
```

También puedes crear índices con varias claves.

```
> db.media.createIndex( {"pistas.titulo": 1, "pistas.minutos":-
1} )
```

Puedes forzar el uso de un índice en una consulta con hint() y puedes indicar que se cree de fondo para que no bloquee los cambios en los documentos de la colección la primera vez que se crea. Puedes comprobar el rendimiento de usarlo con explain().

```
> db.media.createIndex( {ISBN:1}, {background:true} )
> db.media.find( {ISBN:"978-1-4842-1183-0"} ).hint( {ISBN:1} )
> db.media.find( {ISBN:"978-1-4842-1183-
0"} ).hint( {ISBN:1} ).explain()
```

ADMINISTRACIÓN

Ver los que hay:

```
> db.media.getIndexes()
```

Borrar un índice o todos:

```
> db.media.dropIndex( {ISBN:1} )
```

DATOS GEOGRÁFICOS.

Cuando un documento necesita información geoespacial debe tenerla en un objeto embebido o en un array cuyo primer elemento indica el tipo de objeto seguido de elementos longitud y latitud que lo describen. Por ejemplo:

```
> db.restaurantes.insert( {nombre:"Kimono", loc:{type:"Point",
Coordinates:[52.370451, 5.217497]} } )
```

El elemento type indica el objeto GeoJSON que puede ser un Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon o GeometryCollection.

- Point: señala un punto geográfico localizado por longitud y latitud. Como el ejemplo anterior.
- LineString: el elemento se extiende a lo largo de una línea (una calle por ejemplo) y necesita el punto inicial y el final. Ej:

```
 > db.calles.insert( {nombre: "Westblaak", loc:
{type:"LineString", coordinates: [ [52.36881,4.890286],
[52.368762,4.890021] ] } } )
```

- Polygon: se usa para indicar una figura geométrica. El primer punto debe coincidir con el último para cerrar el área. Ej:

```
> db.parcela.insert( {nombre:"SuperMall", loc: {type:"Polygon",
coordinates: [ [ [52.146917,5.374337], [52.146966,5.375471],
[52.146722,5.375085], [52.146744,5.37437],
[52.146917,5.374337] ] ] } } )
```

- Multi-versiones (MultiPoint, MultiLineString, etc.) es un array de los tipos anteriores. Ej:

```
> db.restaurantes.insert({nombre: "Shabu Shabu", loc: {type:
```

```
"MultiPoint", coordinates: [52.1487441, 5.3873406],
[52.3569665,4.890517] }})
```

INDEXARLOS.

Comenzamos por usar una base de datos que vaya a tener colecciones con documentos que incluyen datos geoespaciales y definimos algunos documentos.

```
> use restaurantes
> db.restaurantes.insert( {nombre:"Kimono", loc:{type:"Point",
coordinates: [ 52.370451, 5.217497] } } )
> db.restaurantes.insert( {nombre:"Shabu Shabu", loc:{type:
"Point", coordinates: [51.915288,4.472786] } } )
> db.restaurantes.insert( {nombre: "Tokyo Cafe", loc:
{type:"Point",
coordinates: [52.368736, 4.890530] } } )
```

Puedes crear índices de los datos geográficos pasando a la sentencia de creación del índice el parámetro 2dsphere.

```
> db.restaurantes.ensureIndex( { loc: "2dsphere" } )
```

El parámetro 2dsphere indica a la función ensureIndex() que indexe una coordenada que describe información en 2 dimensiones de la esfera terrestre. Por defecto, ensureIndex() asume que se dan claves latitud/longitud y que se usan los intervalos -180 a 180. Sin embargo puedes sobreescribir estos valores usando las claves min y max. Ejemplo:

```
> db.restaurantes.ensureIndex( { loc: "2dsphere" }, { min : -500 ,
max : 500 } )
```

Puedes expandir los índices geoespaciales (hacer índices compuestos) usando valores de una segunda clave. Por ejemplo:

```
> db.restaurantes.ensureIndex( { loc: "2dsphere", categoria: 1 } )
```

CONSULTARLOS.

Comenzamos buscando un lugar exacto (una consulta normal) pero que no devuelve nada:

```
> db.restaurantes.find( { loc:[52,5] } )
```

Una mejor aproximación es usar el operador $near: que devuelve los 100 primeros cercanos.

```
> db.restaurantes.find( {loc: {$near: {$geometry:{type:"Point",
coordinates: [52.338433,5.513629] } } } } )
Otra posibilidad es indicar los operadores $maxDistance o
$minDistance. Ej:

> db.retaurantes.find( {loc: { $near:{ $geometry: { type:"Point",
coordinates: [52.338433,5.513629] }, $maxDistance : 40000} } } )
```

También tiene el operador **$geoWithin** para saber indicar que quieres los documentos que están dentro de una figura de tipo **$box** (cuadrado), **$polygon** (polígono), **$center** (círculo) y **$centerSphere** (círculo sobre una superficie esférica como la terrestre). Ej:

```
> db.restaurantes.find( { loc:{$geoWithin: {$box:
[ [52.368549,4.890238],
[52.368849,4.89094] ] } } } )
> db.restaurantes.find( { loc: { $geoWithin : { $center :
[ [52.370524, 5.217682], 10] } } } )
> db.restaurantes.find( { loc: { $geoWithin : { $centerSphere :
[ [52.370524, 5.217682], 10]} } } )
```

Si también quieres saber la distancia de cada documento puedes usar la función geoNear() en vez de find(). Ej:

```
> db.runCommand( {geoNear:"restaurantes", near:{type:"Point",
coordinates:[52.338433,5.513629] }, spherical:true})
```

FICHEROS BINARIOS (gridFS).

MongoDB limita el peso de un documento a 16M, para ficheros grandes ha especificado un sistema de almacenamiento a parte, denominado

GridFS que tiene 2 colecciones. Una contiene el nombre del fichero y la información sobre él (metadatos) y la otra contiene los datos del fichero en trozos (chunks) de 255K. Estas colecciones se crean en el espacio de nombres fs.

COMANDO mongofiles.

Hay utilidades de la línea comandos que podemos usar para jugar un poco con ficheros. Por defecto trabaja en la BD test, pero podemos cambiarla:

- Listar los ficheros que tiene: c:\mongofiles /db:libreria list
- Añadir un fichero: mongofiles put "c:\ficheros\ikercasillas.jpg"
- Buscar un fichero: mongofiles search iker
- Borrar ficheros (borra todos los que coincidan): mongofiles delete nombre
- Pedir ayuda: mongofiles --help
- Puedes ver los chunks del fichero:

```
> db.fs.chunks.find( {}, {size:0} )
```

Un programa en Python de ejemplo:

```
>>> from pymongo import MongoClient
>>> import grid fs
>>> db = MongoClient().test
>>> fs = gridfs.GridFS(db)
>>> with open("c:\fichero.txt") as texto:
>>>   uid = fs.put(texto)
>>> recuperar = fs.get(uid)
>>> for palabra in recuperar:
>>>    print(palabra)
>>> fs.delete(uid)
```

CONSULTAS AVANZADAS (Pipelines).

Solo nos queda ejecutar el comando agregate pasando cada uno de estos pasos como parámetro:

```
> db.libros.aggregate( {"$project":{"autor":1}}, {"$group":
```

```
{"_id":"$autor", "count":{"$sum":1}}}, {"$sort":{"count":-1}},
{"$limit":3} )
```

Para depurar los diferentes pasos del agregate, puedes ir probando poco a poco, es decir, empezar por ejemplo solamente con el $project e ir añadiendo los demás en vista de los resultados.

```
> db.libros.aggregate( {"$project":{"autor":1}} )
```

Ahora, añadimos al $project el $group.

```
> db.libros.aggregate({$project:{"autor":1}}, {$group:
{_id:"$autor", count:{"$sum": 1}}} )
```

Cada operador de los que vamos a ver recibe un conjunto de documentos, les aplica alguna transformación y devuelve resultados a otro operador, formando una cadena de transformaciones (pipeline). Los operadores pueden combinarse en cualquier orden y tantas veces como se quiera.

OPERADOR DE PIPELINE $match

Filtra docs. dentro de un pipeline, de forma que podamos ejecutar una agregación sobre un subconjunto de docs. Por ejemplo, si quisiéramos calcular estadísticas (agregación) para libros solo de una editorial, deberíamos añadir una expresión como:

```
{ $match: {editorial:"Planeta"} }
```

$match puede utilizar todos los operadores para consultas ($gt, $lt, $in, etc). Una buena práctica es poner las expresiones $match tan pronto como se pueda. Los beneficios son dos: primero filtramos docs que no necesitamos pronto, evitando procesamiento innecesario y segundo que $match utiliza índices, si existen, cosa que no hará en medio del pipeline ya que los docs. están en memoria y por tanto sin índices. Ejemplos:

```
> db.libros.aggregate( {$match:{precio:{$gt:20}} } )
> db.libros.aggregate( {$match:{enstock:true} } )
> db.libros.aggregate( {$match:{titulo:/(juego|ladron)/i} } )
```

OPERADOR DE PIPELINE $project

La "proyección" es mucho más potente en el pipeline que en el lenguaje "normal" de la query. Nos permite extraer campos de subdocumentos, renombrarlos y realizar operaciones sobre ellos.

La operación más sencilla de $project realiza una selección simple de unos docs. de entrada, es decir, de los docs. de entrada con campos a, b y c, decimos que nos quedamos únicamente con a y c. Para incluir o excluir un campo, se utiliza la misma sintaxis que para el segundo argumento de find(), es decir "campo": 1 para incluir o "campo":0 para excluir. Por defecto, el campo "_id" siempre se devuelve si existe en los docs. de entrada. Se elimina al excluirlo de forma explícita:

```
{"$project": {"_id":0} }
```

Con $project también podemos renombrar el campo proyectado. Por ejemplo, para transformar el campo "_id" de los libros como "isbn" haríamos esto:

```
> db.libros.aggregate( {$project: {isbn:"$_id"}} )
```

La clave está en $_id, cuando ponemos el símbolo del dólar ($) más el nombre de un campo, nos estamos refiriendo al valor que tomará dicho campo para cada doc. en el entorno de agregación. Por tanto con $project: {isbn:"$_id"} decimos que proyectamos un nuevo campo isbn como el valor que tendrá el campo _id en cada documento de entrada. Igualmente podríamos haber renombrado cualquier otro campo, por ejemplo el campo "enstock" como "disponible". Ejemplos:

```
> db.libros.aggregate( {$project:{isbn:"$_id",
disponible:"$enstock"}} )
> db.libros.aggregate( {$project:{isbn:"$_id",
```

```
disponible":"$enstock", _id:0} } )
> db.libros.aggregate( {$project:{isbn:"$_id",
disponible:"$enstock", _id:0, autor:1} } )
```

La operaciones más simples en $project son inclusión, exclusión y renombrar("$nombre"). Hay otras mucho más potentes. Podemos usar expresiones, que combinan literales y variables.

EXPRESIONES MATEMÁTICAS CON $project

Permiten manipular valores numéricos, que normalmente se especifican en un array. Por ejemplo, supongamos que tenemos una colección de empleados con dos campos: salario y bonus. Si queremos calcular la suma de ambos en una variable total_a_pagar tenemos que utilizar la operación de suma $add, que recibe dos parámetros con los campos o constantes a sumar.

```
> db.empleados.insert({name:"Pedro", salario:24000, bonus:2500})
> db.empleados.insert({name:"Laura", salario:27000, bonus:300})
> db.empleados.aggregate({"$project":{total_a_pagar:{"$add":
["$salario","$bonus"]}}} )
```

Las operaciones se pueden anidar como queramos y conseguir operaciones tan complejas como necesitemos. Supongamos que ahora queremos calcular el salario mensual, para lo queremos dividir la suma anterior por doce.

```
> db.empleados.aggregate( {"$project": {"total_a_pagar_mensual":
{"$divide":[{"$add":["$salario","$bonus"]},12]}}} )
```

Esta es la sintaxis para cada operador matemático.

- "$add": [expr1, expr2, … , exprN]. Toma una o más expresiones y las suma.
- "$substract": [expr1, expr2]. Resta la expr2 a la expr1.
- "$multiply": [expr1, expr2, …, exprn]. Toma una o más expresiones y las multiplica.

- "$divide": [expr1, expr2]. Divide la expr1 entre la expr2.
- "$mod": [expr1, expr2]. Divide la expr1 entre la expr2 y devuelve el resto de la división entera.

EXPRESIONES DE FECHAS CON $project

Muchas agregaciones se basan en el tiempo. ¿Qué ocurrió la semana pasada? ¿Y el mes pasado? Por eso, el entorno de agregación tiene un conjunto de expresiones que pueden utilizarse para extraer información sobre fechas de forma sencilla y son: "$year", "$month", "$week", "$dayOfMonth", "$dayOfWeek", "$dayOfYear", "$hour", "$minute" y "$second". Como es obvio, solo se pueden utilizar este tipo de expresiones sobre campos con el tipo Date. Cada una de estas expresiones son básicamente lo mismo, toman la fecha, la expresión y devuelven un número.

```
> db.libros.aggregate({"$project": { "fecha":1, "year":
{"$year":"$fecha" }}})
> db.libros.aggregate({"$project": { "fecha":1, "mes":
{"$month":"$fecha" } }})
```

EXPRESIONES DE STRING CON $project

Hay una pocas operaciones sobre cadenas. Son:

- "$substr": [expr, start, length]. Devuelve substring de expr, empezando en la posición start con un total de length caracteres. Ejemplo: proyectar los primeros 15 caracteres del campo título.

```
> db.libro.aggregate( {"$project": {inicio:{"$substr":
["$titulo",0,15] }}} )
```

- "$concat": [expr1, expr2, …, exprN]. Devuelve un nuevo string resultado de la concatenación de expr1, expr2 y así hasta exprN. Ejemplo: proyectar un campo descripción en el que concatenamos el titulo, un guión y el autor.

```
> db.libro.aggregate( {"$project":{descripcion:{"$concat":
["$titulo", "-", "$autor"]}, "_id":0}})
```

- "$toLower": expr. Devuelve un string en minúsculas.
- "$toUpper": expr. Devuelve un string en mayúsculas.

EXPRESIONES LÓGICAS CON $project

Existen algunas operaciones lógicas que también podemos utilizar. "$cmp": [expr1, expr2]. Compara las dos expresiones y devuelve 0 si son iguales, un número negativo si expr1 es menor que expr2 y un número positivo si expr1 es mayor que expr2. En el siguiente ejemplo proyectamos un campo barato, que tendrá valor -1 si el precio es mayor que 15 y valor 1 si es menor que 15.

```
> db.libros.aggregate({"$project":{"barato":{"$cmp":
[15,"$precio"]},"precio":1} })
```

- "$strcasecmp": [string1, string2]. Compara dos strings de forma insensible a mayúsculas y minúsculas. El valor devuelto funciona como en el caso de "$cmp".
- "$eq"/"$ne"/"$gt"/"$gte"/"$lt"/"$lte" : [expr1, expr2]. Realiza la comparación entre expr1 y expr2, devolviendo true/false dependiendo de si se cumple o no. Replanteamos el ejemplo anterior de la proyección del campo "barato" utilizando uno de estos comparadores.

```
db.libro.aggregate( {"$project":{"barato":{"$gt":
[15,"$precio"]},"precio":1} } )
```

OPERADOR $group

Permite agrupar docs. basándonos en ciertos campos y combinar sus valores. Es equivalente al GROUP BY de SQL, genera nuevos documentos. Algunos ejemplos: Si tenemos medidas de temperatura

por minuto, y queremos calcular la media por día, deberíamos agrupar las medidas por el valor de su "dia". Si tenemos una colección de estudiantes y queremos organizarlos en grupos basándonos en su nota, debemos agrupar por su campo "nota". Si tenemos una colección de usuarios y queremos saber cuantos usuarios tenemos por ciudad, podemos agrupar por los campos "provincia" y "ciudad", creando un grupo por dicho par.

Cuando elegimos uno o varios campos por los que agrupar, los pasamos a "$group" como el campo "_id" del grupo (nuevos docs.). Para los ejemplos anteriores tendríamos:

```
{"$group" : {"_id": "$dia"} }
{"$group" : {"_id": "$nota"} }
{"$group" : {"_id": {"provincia":"$provincia",
"ciudad":"$ciudad" }} }
```

El resultado de "$group", será un doc. por cada grupo. Pero claro, solamente con los grupos no podemos hacer mucho. Lo que en realidad queremos es poder calcular valores a nivel de grupo. Aquí es donde entran en juego los operadores de agrupación.

- "$sum": value. Suma value para cada documento. Ej: total de precio de los libros de cada autor.

  ```
  > db.libros.aggregate({"$group":{"_id":"$autor","total_precios":
  {"$sum":"$precio"} }})
  ```

- "$avg": value. la media aritmética. Ej: media de precios por autor.

  ```
  > db.libro.aggregate({"$group":{"_id":"$autor","media":
  {"$avg":"$precio"} }})
  ```

- "$max": value y "$min": value. el valor más alto y más bajo del grupo.

- "$first": expr y "$last": expr. primer/último valor en el grupo
  Sólo es útil después de hacer sort.

OPERADOR DE PIPELINE $sort

Podemos ordenar por cualquier campo o conjunto de campos utilizando la misma sintaxis que para las queries "normales". Si tenemos que ordenar un número de documentos muy elevado, es recomendable desde el punto de vista del rendimiento, que hagamos la ordenación al principio del pipeline y además tengamos un índice por el conjunto de campos por los que ordenar. De otra forma $sort puede ser lento y consumir mucha memoria.

Se pueden utilizar tanto campos existentes como campos proyectados. De esta forma es posible por ejemplo completar la agregración en la que calculábamos la media aritmética del precio de los libros por autor, y ordenar los resultados por dicha media calculada ascendente.

```
> db.libros.aggregate( {"$group":{"_id":"$autor", media:
{"$avg":"$precio"}}},  {"$sort":media:1} } )
> db.libros.aggregate({"$project":{precio:1,paginas:1}},{"$sort":
{precio:-1,paginas:1}})
```

OPERADOR DE PIPELINE $limit Y $skip

$limit toma un número N y devuelve los primeros N documentos resultantes.

```
> db.libros.aggregate({"$project":{precio:1,paginas:1}},{"$sort":
{precio:-1,paginas:1}}, {"$limit":3} )
```

$skip toma un número N y desecha l os primeros N documentos del conjunto de resultados.

```
> db.libros.aggregate({"$project":{precio:1,paginas:1}}, {"$sort":
{precio: -1,paginas:1}}, {"$skip":3} )
```

OPERADOR DE PIPELINE $unwind

Coge un array de los documentos, y genera un nuevo documento para cada elemento.

```
> db.libros.insert({titulo:"libro1", genero:["accion",
"terror"]} )
> db.libros.aggregate( {$unwind:"$genero"} )
{titulo:"libro1", genero:"accion"}
{titulo:"libro1", genero:"terror"}
```

OPERADOR DE PIPELINE $out

Permite que los resultados se almacenen en otra colección en vez de directamente en los resultados. Por ejemplo, queremos escribir la colección libros:

```
> db.libros.aggregate( {$unwind : "$genero" }, {$project: { _id:0,
titulo:1}},
{$out:"otros" } )
> db.otros.find( null, {_id:0} )
{titulo:"libro1", genero:"accion"}
{titulo:"libro1", genero:"terror"}
```

OPERADOR DE PIPELINE $lookup

Permite realizar una operación similar a un JOIN de SQL, concretamente un left outer join. Es decir, lee los datos de una colección y los mezcla con datos de otra colección:

```
db.primera.insert( {num:1, english:"one"} )
db.primera.insert( {num:2, english:"two"})
db.segunda.insert( {num:1, ascii:49})
db.segunda.insert( {num:2, ascii:50})
```

Vamos a reunir los documentos que coincidan en el campo num:

```
> db.primera.aggregate([{$lookup:{from: "segunda",
localField:"num", foreignField:"num",
as: "segundaDoc"} }, ] )
```

Obtienes todos los documentos de segunda almacenados en un array.

Para ver los resultados como en un JOIN hay que usar también los operadores $unwind y $project.

```
> db.primera.aggregate( [
{$lookup:{from:"segunda", localField:"num", foreignField:"num",
as:"segundaDoc"}},
{$unwind:"$secundaDoc"},
{$project: {_id :"$num", english:1, ascii:"$segundaDoc.ascii" }}
])
```

Que devuelve:

```
{"_id": 1, "english": "one", "ascii": 49}
{"_id": 2, "english": "two", "ascii": 50}
```

11.6 USAR MONGO DESDE JAVA.

Antes de conectar tu programa de Java con un servidor MongoDB, lógicamente debes tener instalado y configurados el servidor Mongo y el driver MongoDB CLIENT.

- Necesitas descargar el jar **mongodb-driver-*.jar** y **dependency mongodb-driver-core-*.jar**.
- Necesitas incluir estos jar en tu proyecto.

CONECTAR A UNA BASE DE DATOS

Necesitas indicar el nombre de la base de datos y si no existe, se crea una autmáticamente.

EJEMPLO: conexión a una base de datos desde Java.

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class ConectarBD {

   public static void main( String args[] ) {
```

```
        // Crear un Mongo client
        MongoClient mc = new MongoClient( "localhost" , 27017 );
        // Crear Credenciales si está protegida
        MongoCredential creden;
        creden = MongoCredential.createCredential("usuario", "BD",
           "password".toCharArray() );
        System.out.println( "Conectado a la BD " + BD + "..." );
        // Acceder a la BD
        MongoDatabase db = mc.getDatabase("prueba");
        System.out.println("Credenciales ::"+ creden );
    }
}
```

CREAR UNA COLECCIÓN

Para crear una colección usas el método **createCollection()** de la clase **com.mongodb.client.MongoDatabase**.

EJEMPLO: Crear una colección.

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class CreaCollection {

   public static void main( String args[] ) {
      // Crear el Mongo client
      MongoClient mc = new MongoClient( "localhost" , 27017 );
      // Crear credenciales
      MongoCredential creden;
      creden = MongoCredential.createCredential("usuario",
                        "prueba", "password".toCharArray());
      System.out.println("Connectado...");
      //Acceder a la BD prueba
      MongoDatabase db = mc.getDatabase("prueba");
      //Crear una coleccion
      db.createCollection("cantantes");
      System.out.println("Coleccion creada...");
   }
}
```

SELECCIONAR UNA COLECCIÓN

Usa el método **getCollection()** de **MongoDatabase**.

EJEMPLO: selecciona una colección con la que trabajar.

```
            // Añade al código del ejemplo anterior...
            MongoCollection<Document> colec =
                      db.getCollection("cantantes");
            System.out.println("Seleccionada colección cantantes...");
        }
    }
```

INSERTAR UN DOCUMENTO

Usa el método **insert()** de **MongoCollection**.

EJEMPLO: inserta un documento en la colección cantantes.

```
            // Añade al código del ejemplo anterior....
            Document doc = new Document("nombre", "Rafael")
                           .append("genero", "pop")
                           .append("likes", 100)
                           .append("url", "https://www.rafael.com/")
                           .append("discos", "4");
            // Insertar el documento
            colec.insertOne(doc);
            System.out.println("Documento insertado...");
    }
```

RECUPERAR TODOS LOS DOCUMENTOS

Usa el método **find()** de **MongoCollection** que devuelve un cursor sobre el que necesitas iterar.

EJEMPLO: mostrar todos los documentos de una colección

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import java.util.ArrayList;
import java.util.Iterator;
```

```
import java.util.List;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;

public class RecuperaTodos {

    public static void main( String args[] ) {
      // Crear el Mongo client
      MongoClient mc = new MongoClient( "localhost" , 27017 );
      // Crear credenciales
      MongoCredential creden;
      creden = MongoCredential.createCredential("usuario",
                      "prueba", "password".toCharArray());
      System.out.println("Connectado...");
      //Acceder a la BD prueba
      MongoDatabase db = mc.getDatabase("prueba");
      //Crear una coleccion
      db.createCollection("cantantes");
      System.out.println("Coleccion creada...");
      Document doc1 = new Document("nombre", "Rafael")
                      .append("genero", "pop")
                      .append("likes", 100)
                      .append("url", "https://www.rafael.com/")
                      .append("discosOro", "4");
      System.out.println("Documento insertado...");
      Document doc2 = new Document("nombre", "AC/DC")
                .append("genero", "heavy")
                .append("likes", 200)
                .append("tipo", "grupo")
                .append("pais", "Australia");
      List<Document> lista = new ArrayList<Document>();
      lista.add(doc1);
      lista.add(doc2);
      colec.insertMany(lista);
      // Obtener un cursor y recorrerlo
      FindIterable<Document> iterDoc = colec.find();
      int i = 1;
      Iterator it = iterDoc.iterator();
      while( it.hasNext() ) {
           System.out.println(it.next());
           i++;
      }
```

```
    }
}
```

ACTUALIZAR UN DOCUMENTO

Usando **updateOne()** de **MongoCollection**.

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import com.mongodb.client.model.Updates;
import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class UpdatingDocuments {

   public static void main( String args[] ) {
      MongoClient mongo = new MongoClient( "localhost" , 27017 );
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser",
"myDb",
         "password".toCharArray());
      System.out.println("Connected to the database
successfully");
      MongoDatabase database = mongo.getDatabase("myDb");
      MongoCollection<Document> collection =
database.getCollection("sampleCollection");
      System.out.println("Collection myCollection selected
successfully");
      collection.updateOne(Filters.eq("title", 1),
Updates.set("likes", 150));
      System.out.println("Document update successfully...");
      FindIterable<Document> iterDoc = collection.find();
      int i = 1;
      Iterator it = iterDoc.iterator();
      while (it.hasNext()) {
         System.out.println(it.next());
         i++;
      }
   }
}
```

BORRAR UN DOCUMENTO

Usas **deleteOne()** de **MongoCollection**.

```
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import java.util.Iterator;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class DeletingDocuments {

   public static void main( String args[] ) {
      MongoClient mongo = new MongoClient( "localhost" , 27017 );
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser",
"myDb",
         "password".toCharArray());
      System.out.println("Connected to the database
successfully");
      MongoDatabase database = mongo.getDatabase("myDb");
      MongoCollection<Document> collection =
database.getCollection("sampleCollection");
      System.out.println("Collection sampleCollection selected
successfully");
      collection.deleteOne(Filters.eq("title", "MongoDB"));
      System.out.println("Document deleted successfully...");
      FindIterable<Document> iterDoc = collection.find();
      int i = 1;
      // Getting the iterator
      Iterator it = iterDoc.iterator();
      while (it.hasNext()) {
         System.out.println(it.next());
         i++;
      }
   }
}
```

BORRAR UNA COLECCIÓN

Usas **drop()** de **MongoCollection**.

```
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import org.bson.Document;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class DropingCollection {

   public static void main( String args[] ) {
      // Creating a Mongo client
      MongoClient mongo = new MongoClient( "localhost" , 27017 );
      // Creating Credentials
      MongoCredential credential;
      credential = MongoCredential.createCredential("sampleUser",
"myDb",
         "password".toCharArray());
      System.out.println("Connected to the database
successfully");

      // Accessing the database
      MongoDatabase database = mongo.getDatabase("myDb");

      // Creating a collection
      System.out.println("Collections created successfully");
      // Retrieving a collection
      MongoCollection<Document> collection =
database.getCollection("sampleCollection");
      // Dropping a Collection
      collection.drop();
      System.out.println("Collection dropped successfully");
   }
}
```

LISTAR TODAS LAS COLECCIONES

Usas el método **listCollectionNames()** de **MongoDatabase**.

```
import com.mongodb.client.MongoDatabase;
import com.mongodb.MongoClient;
import com.mongodb.MongoCredential;
public class ListOfCollection {
```

```
      public static void main( String args[] ) {
         MongoClient mongo = new MongoClient( "localhost" , 27017 );
         // Creating Credentials
         MongoCredential credential;
         credential = MongoCredential.createCredential("sampleUser",
"myDb",
            "password".toCharArray());
         System.out.println("Connected to the database
successfully");
         MongoDatabase database = mongo.getDatabase("myDb");
         System.out.println("Collection created successfully");
         for (String name : database.listCollectionNames()) {
            System.out.println(name);
         }
      }
}
```

11.6.1 OBJETOS JAVA <-> DOCS

Java trabaja con objetos y MongoDB almacena documentos, que es una forma de representar objetos. Si se implementa la conversión automática objeto <-> documento se simplifica la solución de problemas del mundo real. Mapear documentos de Mongo a Plain Old Java Objects (POJOs) usando el driver de MongoDB.

Imagina que MongoDB tiene una colección llamada notas que almacena documentos como por ejemplo:

```
{ "_id": { "$oid": "56d5f7eb604eb380b0d8d8ce" },
 "alumnot_id": { "$numberDouble": "0" },
 "notas": [
   {"tipo": "examen", "nota": { "$numberDouble": "78.40446309504266" } },
   {"tipo":"cuestionario","nota":{"$numberDouble":"73.36224783231339"} },
   {"tipo":"practica", "nota": {"$numberDouble":"46.980982486720535" } }
         ],
   "clase_id": {"$numberDouble":"339"}
}
```

POJOs

Lo primero que necesitamos es representar este documento en un objeto de Java. Para cada documento o subdocumento necesitaremos una clase POJO asociada.
En el ejemplo, tenemos que el documento principal tiene un array de subdocumentos Notas, por tanto necesitamos dos POJOS en Java:

- Uno para el alumno.
- Otro para sus notas.

```
package p11;

// imports...

public class Alumno {
    private ObjectId id;
    @BsonProperty(value = "alumno_id")
    private Double alumnoId;
    @BsonProperty(value = "class_id")
    private Double clase;
    private List<Nota> notas;
    // getters y setters
    // toString()
    // equals()
    // hashCode()
}
```

Indica que el campo alumnoId
Se mapea a alumno_id

```
package p11;

import java.util.Objects;

public class Nota {
    private String tipo;
    private Double nota;
    // getters y setters
    // toString()
    // equals()
    // hashCode()
}
```

Comentamos la clase MapeaPOJO:

Necesito configurar CodecRegistry para incluir un codec que se encargue de traducir entre BSON y los POJOs.

```
CodecRegistry pojoCodec =
   fromProviders( PojoCodecProvider
                  .builder()
                  .automatic(true)
                  .build()
                );
```

El listado:

```
package p11;

import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.FindOneAndReplaceOptions;
import com.mongodb.client.model.ReturnDocument;
import com.mongodb.quickstart.models.Grade;
import com.mongodb.quickstart.models.Score;
import org.bson.Document;
import org.bson.codecs.configuration.CodecRegistry;
import org.bson.codecs.pojo.PojoCodecProvider;
import java.util.ArrayList;
import java.util.List;
import static com.mongodb.client.model.Filters.eq;
import static java.util.Collections.singletonList;
import static
     org.bson.codecs.configuration.CodecRegistries.fromProviders;
import static
 org.bson.codecs.configuration.CodecRegistries.fromRegistries;

public class MapearPOJO {
    public static void main(String[] args) {
        ConnectionString cs =
          new ConnectionString(System.getProperty("mongodb.uri"));
        CodecRegistry pojoCR =
                fromProviders(PojoCodecProvider.builder()
                                               .automatic(true)
```

```
                                                .build());
        CodecRegistry cR = fromRegistries(
                    MongoClientSettings.getDefaultCodecRegistry(),
                    pojoCR);
        MongoClientSettings clientS =
          MongoClientSettings.builder()
                         .applyConnectionString(connectionString)
                         .codecRegistry(cR)
                         .build();
        try (MongoClient mc = MongoClients.create(clientS)) {
            MongoDatabase db = mc.getDatabase("ejemplo");
            MongoCollection<Alumno> alum =
                       db.getCollection("alumnos", Alumno.class);
            // crea nuevo alumno
            Alumno a1 = new Alumno()
                                .setStudent_id(10003d)
                                .setClase_id(10d)
                                .setNotas(
                        singletonList( new Nota()
                                        .setTipo("practica")
                                        .setNota(50d)));
            alum.insertOne(a1);
            // Encontrar...
           Alumno a = alum.find(eq("alumno_id", 10003d)).first();
            System.out.println("Alumno encontrado:\t" + a);
            // Modificarlo añadiendo examen
            List<Notas> ln = new ArrayList<>( a.getNotas() );
            ln.add(new Nota().setTipo("examen").setNota(42d));
            a.setNotas(ln);
            Document filtro = new Document("_id", a.getId());
            FindOneAndReplaceOptions rdar =
                new FindOneAndReplaceOptions()
                              .returnDocument(ReturnDocument.AFTER);
            Alumno aModif =
                 alum.findOneAndReplace(filtro, a, rdaf);
            System.out.println("Alumno:\t" + aModif);
            // borrar el alumno
            System.out.println(alums.deleteOne(filtro)); } } }
```

11.7. EJERCICIOS.

EJERCICIO 1: Haz un programa que se conecte a F1.db4 y pregunte

por una cantidad de puntos y muestre los pilotos que tienen esa puntuación.

```
Int puntosBuscados = sc.nextInt();
Piloto proto = new Piloto(null, puntosBuscados);
ObjectSet r = db.queryByExample(proto);
listResult(r);
```