NoSql

En los últimos años, la cantidad de datos digitales que se genera el mundo se ha multiplicado. Las redes sociales y el cada vez más fácil acceso a Internet del que disponemos las personas hacen que el volumen de tráfico y de datos que se generan sea cada vez mayor.

Con el surgimiento de las bases de datos relacionales las empresas encontraron el aliado perfecto para cubrir sus necesidades de almacenamiento, disponibilidad, copiado de seguridad y gestión de sus datos.
Pero debido a las tendencias actuales de uso de Internet, este tipo de sistemas han comenzado a experimentar dificultades técnicas, en algunos casos bloqueantes, que impiden el buen funcionamiento de los sistemas de algunas de las empresas más importantes de Internet. Este tipo de datos que son masivamente generados reciben un nombre: BigData, y un tipo de tecnología ha surgido para tratar de poner solución a muchos de los problemas de los que adolecen los sistemas de almacenamiento tradicionales cuando intentan manejar este tipo de datos masivos. Esta tecnología se conoce como NoSql.

¿Qué es Big Data?

Diversas personalidades con autoridad en el mundo de Internet han plasmado su opinión respecto a este tema.
Ed Dumbill forma parte del staff de O'Really Media, la conocida marca editorial que publica multitud de libros técnicos al año, y dice esto en el libro Big Data Now:

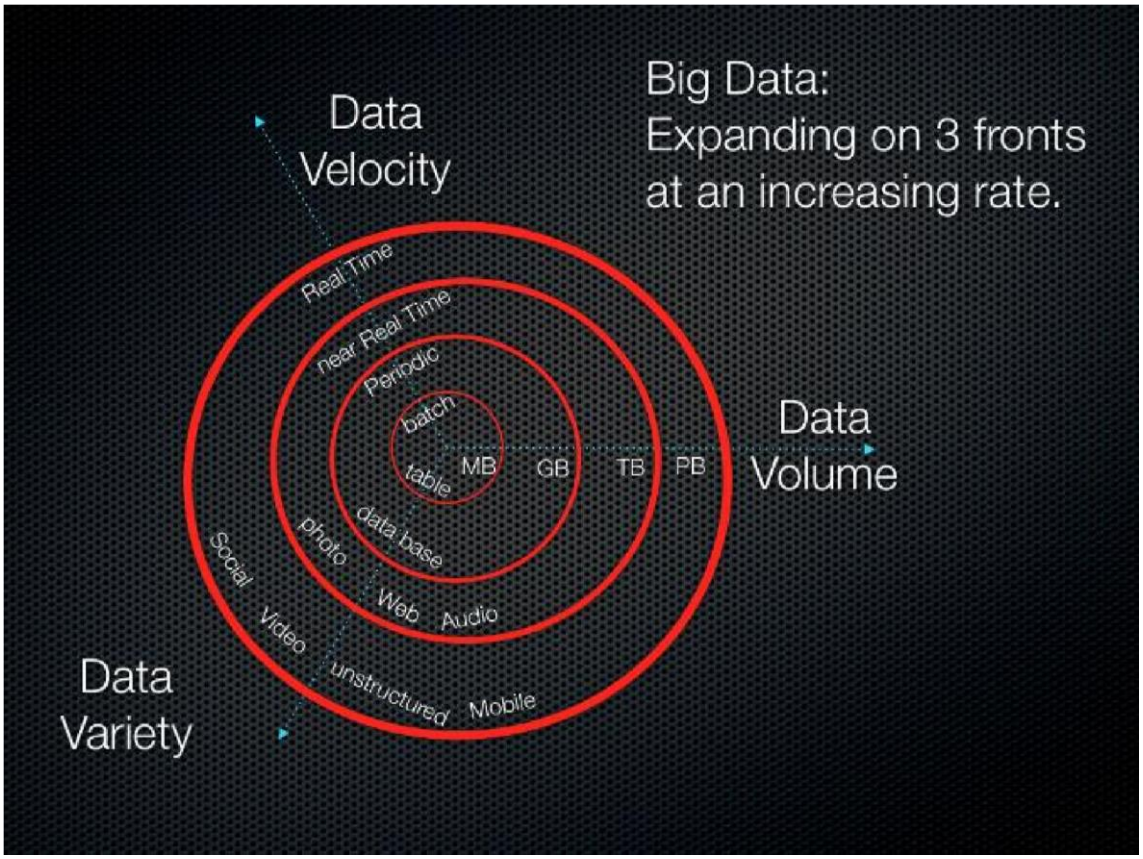
"*Big data is data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the strictures of your database architectures. To gain value from this data, you must choose an alternative way to process it*." (O'Really 2012)

En el blog oficial de Microsoft Enterprise se puede leer:
"*Big data is the term increasingly used to describe the process of applying serious computing power – the latest in machine learning and artificial intelligence – to seriously massive and often highly complex sets of information*" (HowieT 2013)

Adrian Merv, de la revista Teradata expone la siguiente definición: "*Big data exceeds the reach of commonly used hardware environments and software tools to capture, manage and process it within a tolerable elapsed time for its user population*" (Merv 2011)

Atendiendo a estas definiciones, se podría hacer un resumen, a grandes rasgos y proponer nuestra propia definición sobre lo que el Big Data significa:

Big Data es la ingente cantidad de información, en su mayor parte desestructurada, que hoy en día generamos toda la sociedad como consecuencia de nuestra actividad tanto en Internet como fuera de ella.



Conceptos básicos, técnicas y patrones

Antes de pasar a ver más en profundidad los tipos de bases de datos, así como las propias bases de datos en sí, se analizarán los conceptos fundamentales, técnicas y patrones que son comunes a este tipo de bases de datos.

Consistencia

El Teorema CAP

El teorema CAP ha sido ampliamente adoptado por las grandes compañías de internet, al igual que por la comunidad NoSQL.
Las siglas CAP hacen referencia a:

- **Coherencia**. En sistema distribuido, habitualmente se dice que se encuentra en un estado consistente si, después de una operación de escritura, todas las operaciones de lectura posteriores son capaces de ver las actualizaciones desde la parte del sistema desde la que están leyendo.
- **Disponibilidad (Availability).** El sistema está disponible y responde en un tiempo adecuado a todos los clientes. Si el tiempo de respuesta excede un umbral el sistema se lo considera no disponible.
- **Tolerancia a Particiones (Partition Tolerance).** Entendido como la habilidad de un sistema de tener diferentes regiones o divisiones lógicas en la red, y de ser capaz de seguir funcionando aunque una de estas partes quede inaccesible durante un tiempo.

La teoría CAP (también se conoce como teorema de Bewer (Bewer 2012)) expone que es imposible que un sistema distribuido pueda garantizar simultáneamente estas 3 características. Sin embargo, el teorema de CAP también dice que sí puedes garantizar 2 de estas 3 propiedades. Los SGBD NoSQL cumplen dos de estas 3 características, y por tanto, se puede realizar una clasificación de dichos SGBDs en función de esto.

De este modo, atendiendo lo que se acaba de exponer en el punto anterior, podemos deducir que se puede "elegir" dos de estas 3 características para nuestro sistema de datos compartidos. De esta manera se tiene:

- Sistemas coherentes y disponibles, pero con dificultades para funcionar en caso de que haya muchas particiones.
- Sistemas coherentes y tolerantes a particiones, pero con ciertas carencias en temas de disponibilidad.
- Sistemas disponibles y tolerantes a particiones, pero no estrictamente coherente.

El siguiente cuadro ejemplifica lo anteriormente expuesto.

Elección Características Ejemplos
CA. Consistencia y Disponibilidad (se pierde la capacidad Particiones
Confirmaciones dobles Protocolos de validación de caché Relacionales
(Oracle, Mysql, SQL Server),
Neo4J
CP. Consistencia y Particiones (se pierde la capacidad Disponibilidad)
Bloqueos "pesimistas" Ignorar las particiones más pequeñas
MongoDB, HBase, Redis
AP. Disponibilidad y Particiones (se pierde la capacidad Consistencia)
Invalidaciones de caché Resolución de conflictos
DynamoDB, CouchDB, Cassandra

Algunas bases de datos de las que se acaba de exponer son configurables, de forma que podrían pasar de una categoría a otra.
Elegir un tipo de base de datos u otra dependerá enteramente del problema a resolver.
Hay que tener en cuenta las ventajas e inconvenientes de cada tipo de base de datos, de forma que se adapte lo máximo posible, no solo a nuestro modelo de datos, sino también a las características del servicio que se desea ofrecer.

Garantías ACID

El mundo de las bases de datos relacionales está familiarizado con las transacciones ACID.
Las transacciones que se producen en el lenguaje SQL, sea cual sea el sistema gestor de base de datos cumplen siempre las propiedades ACID. Este tipo de transacciones se llaman así porque garantizan la **A**tomicidad, la **C**onsistencia, el aislamiento (**I**solation) y la **D**urabilidad.

- **Atomicity** (Atomicidad). Las transacciones han de ejecutarse por completo o no ejecutarse, pero la transacción no puede quedar a medias.
- **Consistency** (Consistencia o Integridad). Los datos que se guardan tras la transacción han de ser siempre datos válidos.

- **Isolation** (Aislamiento). Las transacciones son independientes y no se afectan entre sí.
- **Durability** (Durabilidad). Una vez finalizada una operación, esta perdurará en el tiempo.

El modelo BASE es un enfoque similar al ACID, aunque perdiendo la consistencia y al aislamiento, a favor de la disponibilidad, la degradación y el rendimiento.

El modelo **BASE** toma su nombre de:
- **Basic Availability**. El sistema funciona incluso cuando alguna parte falla, debido a que el almacenamiento sigue los principios de distribución y replicación.
- **Soft State**. Los nodos no tienen porque ser consistentes entre si todo el tiempo.
- **Eventual Consistency**. La consistencia se produce de forma eventual. Para que un sistema gestor de bases de datos relacional pueda ser considerado tal, debe cumplir el modelo ACID.

MapReduce

MapReduce es un modelo de programación además de una implementación para procesado y tratamiento de grandes cantidades de información. Es una de las 2 piezas más importantes de Hadoop, junto con el *HDFS*, y está basado en un artículo publicado por Google en el año 2004 (Dean and Ghemawat 2004).

El modo de funcionar de *MapReduce*, a grandes rasgos, consiste en tomar una entrada de un tamaño del orden de gigabytes o terabytes de tamaño, dividirla en trozos de un tamaño más pequeño, y realizar una serie de operaciones que operan sobre esos datos de forma paralela en multitud de ordenadores que trabajan en conjunto.
En las siguientes secciones se darán más detalles sobre esto.

Arquitectura

MapReduce tiene dos procesos que se encargan de dividir y procesar los Jobs que un usuario o un agente externo encola al sistema. Estos son:
- El *JobTracker*.
- El *TaskTracker*.

El ***JobTracker*** harías las veces de maestro (similar al *NameNode* del sistema de ficheros) y sería el encargado de recibir los nuevos trabajos (o *Jobs*) y dividirlos en tareas (o *tasks*). También será el encargado de recibir los resultados tanto del *map* como del *reduce* (se verán más adelante) y de presentarle los resultados al agente que pidió la ejecución del *job*. Además, el *JobTracker* realizará tareas de monitorización básicas sobre los *TaskTrackers*, de manera que detectará cuando alguno de ellos queda inaccesible (el nodo muere) para dejar de enviarle tareas, o bien para repartir las tareas que ese nodo tenía asignadas entre el resto de nodos vivos.

Los ***TaskTrackers*** son los agentes encargados de recibir esas tareas en las que se ha subdividido el trabajo y de procesarlas. El *TaskTracker* recibirá una porción de los datos que necesitan ser procesados y el código que tiene que ejecutar.

Una vez que haya finalizado su trabajo, su misión es devolver el resultado al maestro, el *JobTracker*.

Modelo de programación

De forma genérica, *MapReduce* toma una tupla (par clave – valor) por entrada, y genera un conjunto de tuplas como salida. *MapReduce* divide su ejecución en 2 funciones inspiradas en la programación funcional: la función *map* y la función *reduce*.

Map

La fase *map* no es más que un conjunto de valores o datos, a los que debe aplicarse, de forma independiente, una función definida por el usuario. Como resultado de esta operación se obtendrá un conjunto de nuevos pares clave – valor, que serán ordenados convenientemente según su clave antes de presentarse a la salida de la función.
Map(k1,v1) -> list(k2,v2)

Reduce

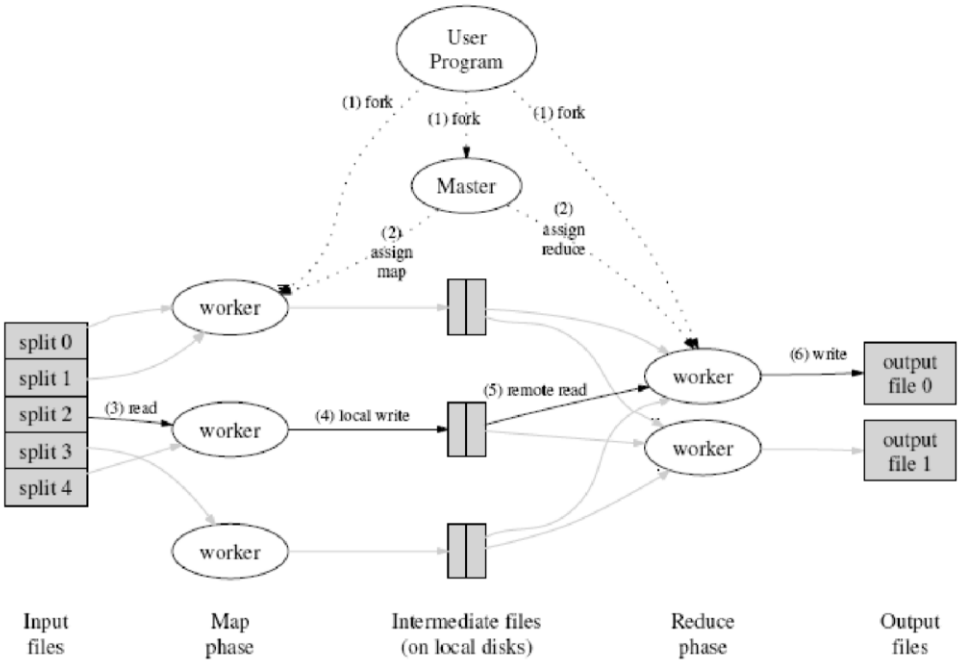
La fase *reduce*, por su parte, recibe un conjunto de tuplas a las que aplica, en conjunto, una función definida por el usuario. Esta función puede ser extremadamente simple (una suma, o una concatenación) hasta tan complicada como se desee. Así, generará a su vez una salida que corresponderá a una lista de nuevos valores para la clave inicialmente dada. Reduce(k2, list (v2)) -> list(v3)

Así, globalmente se puede resumir el proceso *MapReduce* como un conjunto de valores en formato clave valor que son tratados y transformados, dando como resultado del proceso una lista de valores finales.

A modo de ejemplo, digamos que quiero contar cuantas veces aparece cada palabra en un conjunto de documentos:

- Subo los documentos a un claster (50 servidores)
- Escribo un programa (Map) que encuentra palabras en un documento y genera pares del tipo clave/valor donde la clave es cada palabra que encuentra y el valor es 1.
- Mando el programa a ejecutarse en los 50 servidores.
- Escribo otro programa que consolide los resultados (Reduce), es decir recibe los pares y generas nuevos pares del tipo clave/valor donde la clave es cada palabra y el valor es la suma de veces que aparece.

Un ejemplo equivalente puede ser la frecuencia de acceso a una URL, en este caso el conjunto de datos son los logs de requerimientos de páginas web.


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
split 0
split 1
split 2
split 3
split 4
(3) read
worker
(4) local write
worker
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Bases de Datos NoSQL

En los últimos años, una gran variedad de bases de datos NoSQL han salido a la luz, creadas por compañías principalmente para cubrir sus propias necesidades. Temas como escalabilidad, rendimiento, mantenimiento, etc. que no encontraban en ninguna solución que existía en el mercado.
Debido a la variedad de enfoques que existe entre requisitos y funcionalidades que debe cumplir una base de datos NoSQL, es bastante difícil mantener una visión general de la situación actual de las bases de datos no relacionales. Se puede decir que las bases de datos NoSQL son una categoría independiente dentro del conjunto de bases de datos. Más adelante se expondrán diversos conceptos y características comunes de las bases de datos NoSQL. Ahora se hará una clasificación general de las bases de datos NoSQL más importantes atendiendo a su modelo de datos.

Bases de Datos Clave – Valor

Las bases de datos clave – valor (también llamadas a veces como *tablas hash* o *simplemente tablas*) se estructuran almacenando la información como si de un diccionario se tratara. El diccionario, en este caso, es un tipo de datos que contienen tuplas clave valor. Los clientes añaden y solicitan valores a partir de una clave asociada que conocen de antemano.
Los sistemas modernos de almacenamiento clave – valor se caracterizan por tener una elevada escalabilidad y un rendimiento muy bueno para volúmenes de datos muy grandes. A cambio, su estructura es muy sencilla, y sacrifica ciertas funcionalidades como la consistencia inmediata, la verificación de la integridad de datos o las referencias externas.
En el caso concreto de las referencias externas y la integridad referencial, deberá ser la propia aplicación que hace uso de la base de datos la que se encargue de actualizar los valores correspondientes cada vez que exista un borrado o una modificación sobre el conjunto de datos.
En los sistemas de bases de datos relacionales existen agrupaciones o componentes llamados "*base de datos*" (o su voz inglesa: "*databases")*, y dentro de ellas existen contenedores llamados tablas Por el contrario, en los sistemas clave valor existen los "contenedores" (o su voz inglesa: "*cabinets*"), y dentro de ellos se pueden almacenar tantos pares clave-valor como se desee. Existen distintos tipos de *contenedores*, algunos permiten pares duplicados, otros pares admiten valores nulos, etc.
Muchas de estas bases de datos funcionan almacenando la información en memoria principal, de manera que aumenta aún más la velocidad de respuesta ante nuevas lecturas / escrituras. Otras también realizan un copiado periódico a disco para persistir los datos, y poder recuperarlos más adelante si se establece un punto de restore, o bien la maquina cae y se desea recuperar una imagen concreta de cierto momento.
Mas adelante se analizará en profundidad las bases de datos de tipo clave – valor más utilizadas, como Memcached, Redis o DynamoDB.

Bases de Datos Documentales

Las bases de datos documentales o bases de datos orientadas a documento, son otro tipo de base de datos NoSQL con un grado de complejidad y flexibilidad superior a las bases de datos clave – valor.

En las bases de datos documentales el concepto principal es el de "documento". Un documento es la unidad principal de almacenamiento de este tipo de base de datos, y toda la información que aquí se almacena, se hace en formato de documento.
Las codificaciones más habituales de estos documentos suelen ser XML, YAML o JSON, pero también se pueden almacenar en formato Word o PDF. Habitualmente, los documentos se estructuran dentro de una serie de contenedores llamadas colecciones (o su voz inglesa: "collections") que son proporcionados por el sistema gestor documental.
Los documentos son almacenados dentro de la base de datos con una clave única dentro del almacén, por la cual son también recuperados posteriormente. También existirá un índice principal por esta clave primaria. Se pueden generar índices que afecten a otros campos, pero hay que evaluar bien su construcción, ya que aceleran y mejoran los tiempos de carga por ese campo pero a cambio tienen necesidades de espacio y de mantenimiento a tener en cuenta.

Bases de Datos Orientadas a Grafos

Las bases de datos orientadas a grafos tienen la particularidad de representar la información como si de un grafo se tratara. La información viene representada por los nodos, y las relaciones entre los datos por las aristas. De este modo, se puede emplear la teoría de grafos para recorrer la base de datos y así gestionar y procesar la información
Una base de datos orientada a grafos, de forma generalizada, es cualquier sistema de información donde cada elemento tiene un puntero directo hacia sus elementos adyacentes, es decir, no sería necesario realizar consultas mediante índices.
Como se expone a continuación, existen varios casos en el que el uso de una base de datos orientada a grafos es más eficiente que utilizar cualquier otro tipo de base de datos, tanto relacional como NoSQL.
A diferencia de la mayoría de bases de datos, el **rendimiento** de una base de datos orientada a grafos no se deteriora con el crecimiento de la base de datos, ni con consultas o procesamientos muy intensivos de los datos. Esto es debido a que el rendimiento será siempre proporcional al tamaño y el rendimiento que tenga el recorrer la parte del árbol que esté implicada, y no el tamaño global del grafo.
El modelo de grafos dota al desarrollador de aplicaciones de una elevada **flexibilidad** para manipular el conjunto de datos, ya que le permite conectar los datos de forma que posteriormente sea sencillo realizar consultas o actualizaciones desde una aplicación.
Este tipo de bases de datos nos permiten realizar un mantenimiento progresivo y **ágil** de los sistemas, a medida que la aplicación va creciendo, de forma que se puede adaptar a las nuevas necesidades del negocio.

Bases de Datos Orientadas a Objetos

En las bases de datos orientadas a objetos la información se representa como objetos.
Esta manera de representar la información es análoga a la que hace la programación orientada a objetos, que define y representa la información en un conjunto de datos y de operaciones que se pueden realizar sobre esos datos.

Cuando un sistema aúna una base de datos de este tipo y un lenguaje de programación orientado a objetos, el resultado es un sistema gestor de bases de datos orientado a objetos.
La clave de este tipo de almacenamiento reside en la potencia de los lenguajes de programación orientados a objeto, junto con la capa de persistencia que le proporciona una base de datos que ha sido diseñada específicamente para trabajar con aplicaciones desarrolladas con lenguajes orientados a objetos. Los vendedores de bases de datos relacionales tradicionales se dieron cuenta de esto, y poco a poco fueron implementando capas de integración con los diferentes lenguajes de programación más populares, en lo que se conoce como sistemas de mapeo objeto–relacional (también conocido por sus siglas ORM). Sin embargo, esta traducción de objetos al modelo relacional no siempre se puede llevar a cabo, puesto que siempre se perderá algo de algún lado. O bien el modelo orientado a objetos perderá para adaptarse al 100% al modelo relacional, o bien el modelo relacional asociado al modelado orientado a objetos no cumplirá al 100% el paradigma relacional.
Se puede decir que algunas de las ventajas que tienen las bases de datos orientadas a objetos con respecto al modelo relacional son:

- Deja de emplearse el SQL como lenguaje de base de datos. En su lugar, es el propio código de aplicación la que realiza esta funcionalidad.
- Se elimina la doble representación del modelo de datos: el modelo de los objetos y el relacional. Ahora el diagrama de clases de la aplicación es el que construye el modelo de persistencia de la base de datos.
- Mejora el rendimiento cuando se trata con objetos muy complejos.
- Cualquier cambio en los objetos se aplican directamente en base de datos, no es necesario reconfigurar ni migrar nada.

Las bases de datos orientadas a objetos son, probablemente, las más antiguas de todas las bases de datos NoSQL, ya que el pulso que han mantenido con las bases de datos relacionales data de antes de que Google liberara los artículos de su GFS (Google File System) y Bigtable, la base de datos NoSQL creada por Google, y desde antes de que el movimiento NoSQL empezara a coger fuerza.

Bases de Datos Orientadas a Columnas

Las bases de datos orientadas a columnas son otro caso particular de la enorme familia de las bases de datos NoSQL. En este tipo de almacenes, en contraposición con el modelo relacional, la información se estructura en columnas en lugar de en filas.
Algunas de las bases de datos NoSQL más importantes y con una mayor aceptación pertenecen a este grupo. Bigtable, la solución NoSQL de Google, HBase, la base de datos de Hadoop, Cassandra, impulsada por Facebook y ahora bajo los brazos de Apache Software Foundation, son solo algunos ejemplos de este tipo de bases de datos.
Las comparativas de rendimiento entre los sistemas de gestión por filas (especialmente RDBMS) y los basados en columna, vienen principalmente de la mano de la eficiencia de los accesos que realizan a disco. Los accesos a posiciones consecutivas se producen en una cantidad de tiempo sensiblemente menor que los accesos que se producen a posiciones aleatorias a disco. Tener un sistema con multitud de accesos aleatorios a disco puede lastrar en gran medida el rendimiento del sistema de almacenamiento.
Algunas de las ventajas de utilizar bases de datos orientados a columna son:

- Los sistemas de almacenamiento basados en columnas proporcionan un mejor rendimiento cuando es necesario realizar una transformación de datos que involucra a todas o gran parte de las filas, pero solo a una o una pequeña parte de las columnas. El acceso columnar, en este caso, produce un rendimiento mayor.
- Los sistemas de base de datos basados en columnas son más eficientes cuando hay que sustituir un valor que afecta a una o más columnas en todas las filas de la colección.
- Los sistemas basados en filas son más eficientes en los casos en los que es necesario obtener a la vez varias columnas de una misma fila, siempre que la fila sea lo suficientemente pequeña (esto es, que se pueda obtener con una única búsqueda aleatoria a disco).
- Los sistemas de almacenamiento basados en filas son más eficientes cuando hay que escribir una fila nueva, se proporcionan todos los valores referentes a las columnas, y además la escritura se puede realizar con una única búsqueda en el disco.

El resumen que se puede hacer finalmente es que, las bases de datos basadas en columnas se emplean con mayor rendimiento cuando se utilizan para agilizar la consulta de grandes cantidades de información. Se ven beneficiadas técnicas como la Minería de Datos (en inglés: *Data Mining)*, la inteligencia de negocio (en inglés *Bussiness Intelligence*), informes de marketing (en inglés *Marketing Reports*), ventas, etc.
Por otro lado, los sistemas basados en filas proporcionan habitualmente mejor datos, especialmente en arquitecturas cliente-servidor. El comercio electrónico, la banca o cualquier servicio que requiera la interacción con un usuario o cliente se verá beneficiado de este tipo de base de datos.

MODELO de DATOS

Bases de datos clave-valor

Las bases de datos clave – valor será el primer tipo de bases de datos NoSQL que se abordará en este estudio. Cuando se habla de bases de datos clave – valor se hace referencia a DynamoDB y a Redis.
Este tipo de base de datos posee la estructura de datos más sencilla, lo cual proporcionará ventajas a la hora de particionar y escalar el sistema a lo largo de clústers de decenas e incluso cientos de nodos.

DynamoDB

Introducción

Amazon DynamoDB es un SGBD NoSQL distribuido multi maestro con modelo de datos clave valor. Pero es un SGBD especial, puesto que lo que Amazon ofrece realmente es el servicio de base de datos de manera totalmente gestionado (Amazon 2012)
DynamoDB es un SGBD propietaria desarrollada internamente por Amazon y lanzada allá por 2007, que se incluye dentro de la oferta de servicios que se ofrecen como parte de los AWS (Amazon Web Services).

Características principales

El servicio de Amazon permite únicamente crear nuevas tablas en las que alojar los datos, e interactuar con ellas mediante tus aplicaciones a través de APIs (interfaces de programación que proporciona Amazon para poder añadir, actualizar, borrar y consultar información de la base de datos). Es decir, no se tiene acceso al código fuente, ni al hardware ni a los servicios que hacen funcionar a la base de datos.

Modelo de Datos

El modelo de datos propio de DynamoDB el siguiente:

Tablas

Las tablas son, al igual que en muchos SGBDs, los elementos donde se almacenan las filas o elementos donde se guardan los datos propiamente dichos de la base de datos.
Las tablas en DynamoDB son libres de esquema, esto es, que dos datos de una misma tabla no tienen porque tener los mismos atributos, ni siquiera el mismo número de atributos.

Lo que sí deben tener todos los elementos de una tabla es un atributo obligatorio que haga las funciones de clave primaria, puesto que esta clave identifica unívocamente a cada elemento dentro de los elementos de la tabla. Cada tabla puede contener un número infinito de elementos.

Elementos

Un elemento es información con una estructura que se almacena en la base de datos.
Los elementos son los homónimos de las filas en el modelo relacional. Cada elemento en DynamoDB es el resultado de proporcionar valores a un conjunto de atributos que se definen en el momento de la inserción.
Cada elemento, en conjunto con sus atributos, tiene una restricción de tamaño máximo de 64KB.

Atributos

Los atributos son pares clave – valor o asociaciones de clave – conjunto de valores. Estos atributos no tienen restricción de tamaño máximo.

ID (clave primaria) Atributos

```
101 {
Title = "Book 101 Title"
ISBN = "111-1111111111"
Authors = "Author 1"
Price = -2
Dimensions = "8.5 x 11.0 x 0.5"
PageCount = 500
InPublication = 1
ProductCategory = "Book"
}
```

```
201 {
Title = "18-Bicycle 201"
Description = "201 description"
BicycleType = "Road"
Brand = "Brand-Company A"
Price = 100
Gender = "M"
Color = [ "Red", "Black" ]
ProductCategory = "Bike"
}
```

Bases de datos en columna

HBase

Introducción

HBase es un SGBD, la base de datos NoSQL de Hadoop. HBase fue concebida inicialmente a partir de los artículos que liberó Google sobre BigTable allá por 2004. Se podría decir que HBase es la implementación Open Source del proyecto Bigtable de Google.
HBase es un software construido sobre Hadoop, sin el cual no podría funcionar a día de hoy. Hadoop confiere a HBase una característica indispensable en su modelo de datos: la capa de almacenamiento que proporciona disponibilidad y fiabilidad (HDFS), mientras que la otra capacidad de Hadoop, el entorno de computación de alto rendimiento (MapReduce) frecuentemente es utilizado junto con HBase para acceder a la base de datos y tratar datos en ella.

Características principales

HBase es una base de datos **distribuida**, **persistente** y **ordenada en un mapa multidimensional**. Una celda de esta base de datos viene definida por una clave primaria de fila (**rowkey**), por una clave de columna (**columnkey**) y una marca de tiempo (**timestamp)**.
Esto puede resultar ciertamente abstracto, por lo que primero se analizará punto por punto qué significa que HBase tenga estas características.

Distribuido

Como se ha comentado anteriormente, esta característica no es propia de HBase, sino que viene heredada del tipo de sistema de ficheros en el que se apoya. HBase utiliza para almacenar la información el HDFS de Hadoop, aunque también puede apoyarse en otros, como Amazon S3 (el sistema de almacenamiento propio de Amazon). Estos sistemas de ficheros tienen la característica de ser distribuidos y tolerantes a fallos.

Mapa Ordenado

A grandes rasgos, se puede equiparar HBase a un enorme mapa ordenado (*SortedMap*) de cualquier lenguaje de programación moderno que exista hoy en

día. En este tipo de mapas de datos, los pares clave - valor son almacenados en estricto orden alfabético.
Es muy importante escoger una clave primaria relevante, con la que se hará la ordenación del sistema. Por ejemplo, se considerará un ejemplo práctico de una tabla que almacena nombres de dominio. Tal vez tenga sentido almacenar estos DNS's en notación inversa, esto es, en lugar de guardar *hbase.apache.org*, se almacenará *org.apache.hbase*. De esta manera se tendrá acceso en primer lugar al dominio de primer nivel, y se dejarán los subdominios para el final de la clave. Así, las columnas referidas a subdominios permanecerán "cerca" unas de otras, al estar ordenadas primero por el dominio común a todas ellas.

Multidimensional

Ya se ha comentado que HBase es un tipo de base de datos NoSQL orientado a columnas. ¿Qué significa esto?
Para comprender el hecho de que HBase sea una base de datos multidimensional, debe olvidarse el significado clásico de los términos tabla y columna. Estos conceptos han cambiado con respecto a lo que se acostumbra a leer en la literatura tradicional de los sistemas SGBD. Ahora el concepto de tabla se acerca más al que se tiene de un hash /map.
A continuación se expone un ejemplo para ilustrar como almacenaría HBase la información.

```
{
"1" : {
   "A" : "x",
   "B" : "y"
},

"2" : {
   "A" : "y",
   "B" : "z
},
"3" : {
   "A" : "z",
   "B" : "a"
}
}
```

	A	B
1	x	y
2	y	z
3	z	a

En el ejemplo superior se representa un *SortedMap* cuyas claves apuntan a otro *SortedMap* con exactamente 2 claves: A y B. En adelante, se identificará al par clavevalor de nivel superior como **fila**. En este ejemplo, las claves de nivel superior vienen representadas por "1", "2" y "3" respectivamente, y sus correspondientes valores podrían ser valores propiamente dichos, aunque en

este caso son nuevos mapas clave-valor, cuya clave viene identificada por las letras “A” y “B”. Los mapeos que realizan A y B se conocen como Familias de Columnas.
Las familias de columnas de una tabla son especificadas en el momento de la creación de la tabla, y además es muy difícil o directamente imposible modificarlas una vez creadas. Así es preferible pensar bien el diseño de la tabla antes de empezar a poblarla con datos.
La principal característica de las Familias de Columnas es que pueden tener un número arbitrario de columnas. Estas columnas siempre pertenecen a una Familia de Columnas de las que se han preestablecido anteriormente. Estas columnas vienen identificadas por una etiqueta o calificador. Ampliando el ejemplo propuesto anteriormente:

```
{

"1" : {
    "A:foo" : "x"
    "A:bar" : "y"
    "B:" : "a"
},
"2" : {
    "A:foo" : "z",
    "A:bar" : "k",
    "B:" : "b"
},
"3" : {
    "A:bar" : "S",
    "B:" : "c"
}
}
```

	A		B
	Foo	bar	
1	x	y	a
2	z	k	b
3	S		c

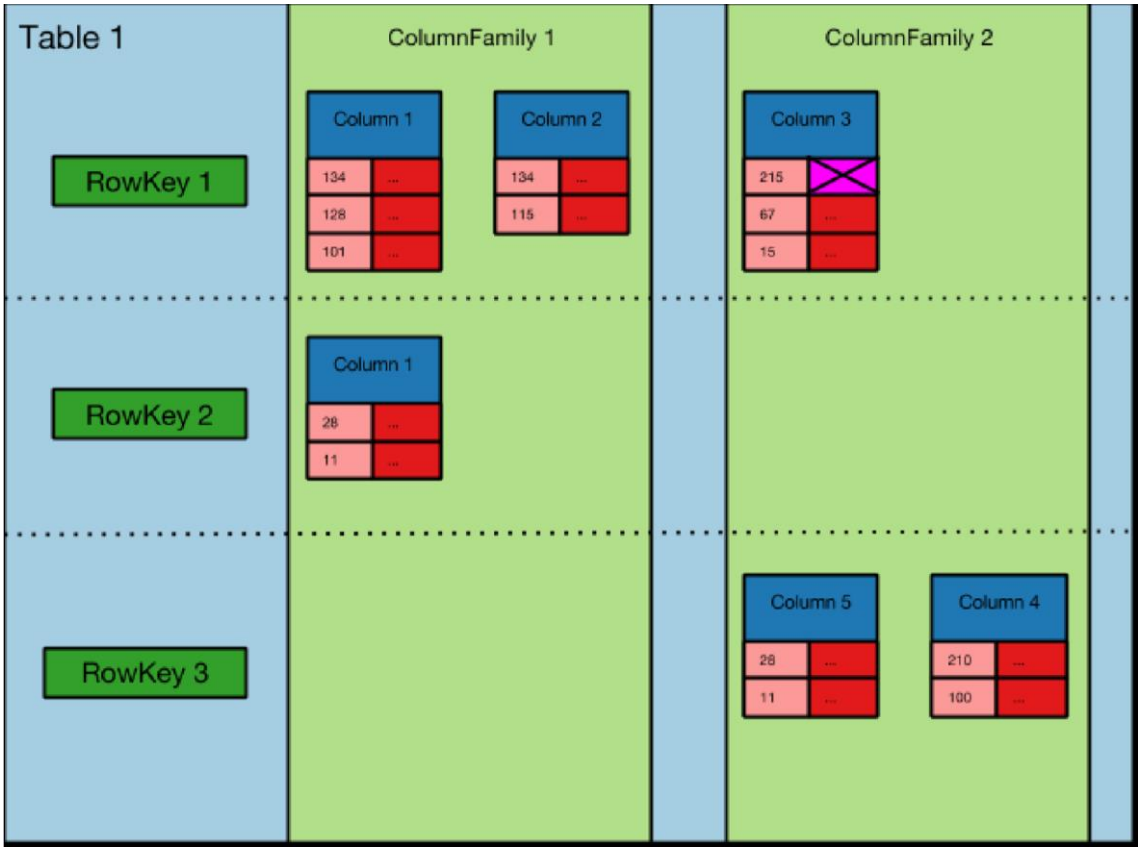
En el ejemplo expuesto, las Column Families A y B se mantienen, pero se han añadido 2 etiquetas para A (foo y bar) y una para B (el conjunto vacío). El conjunto de Column Family y Etiqueta es lo que se denomina Columna. Para referirse a una columna concreta, simplemente se pondrán dos puntos (“:”) entre medias de ambos valores. Esto es, para este ejemplo, se obtendrían las columnas: “A:foo”, “A:bar” y “B:” Aún existe una dimensión más. En HBase, el tiempo marca esta última dimensión. El versionado se consigue incorporando un timestamp de forma implícita con cada fila.
También se puede añadir este campo explícitamente. Esto significa que para una misma celda (combinación de columna – fila) se pueden tener varios valores, valores que serán distintamente accedidos en función del valor del timestamp asociado.

Por último, vale la pena mencionar que todos los miembros de una column families se almacenarán físicamente juntos.

Disperso

En las bases de datos tradicionales, las filas son dispersas pero no las columnas. Esto significa que cada vez que se crea una fila, se reserva siempre espacio para todas y cada una de las columnas, con independencia de si el valor va a ser proporcionado.

La dispersión en esta base de datos principalmente hace referencia al hecho de que, una fila en HBase puede contener cualquier número de columnas de cada column family, desde todas hasta ninguna incluso repetir algunas de ellas. La única condición que debe



Modelo de Datos

En este apartado se dará una visión sobre cómo y de qué manera HBase almacena sus datos. Como se ha comentado anteriormente, HBase utiliza tablas para almacenar la información. Estas tablas están divididas en filas y columnas. Estas columnas son agrupadas a su vez en *column families* previamente definidas (Chang, et al. 2006).

Filas

Las claves principales que se tienen en las filas son String aleatorios, y pueden tener hasta 64 KB de tamaño. Cada lectura / escritura de una fila en una tabla es atómica, independientemente del número de columnas a leer o escribir. Esto fue una decisión de diseño que se tomó pensando en tratar de facilitar, en la medida de lo posible, la predicción del comportamiento del sistema bajo una gran carga de trabajo (multitud de accesos simultáneos sobre los mismos datos). Asimismo,

las claves se mantienen ordenadas lexicográficamente. Los rangos de filas (conocidos como *Regiones*) son particionados dinámicamente. Se puede decir que las regiones son las unidades mínimas de distribución y carga a lo largo del sistema. Cuando una región crece demasiado, se divide, o se puede mover de un servidor a otro. De esta manera, cuando se realizan lecturas sobre un conjunto no consecutivo de filas en la tabla, sólo se requiere comunicación entre un pequeño número de máquinas.
Por ejemplo, si se quisiera almacenar todas las páginas web que son servidas desde un mismo dominio en HBase, la clave principal debería comenzar por el dominio, porque de esa forma se conseguirá que permanezcan "cerca" dentro de la base de datos, y se facilitarán posteriores lecturas y escrituras en el sistema.

Column Families

Las columnas, en HBase, se agrupan en lo que se conoce como *Column Families*. Estas familias de columnas deben ser creadas antes de comenzar a almacenar información dentro de la base de datos. Una vez que la familia se ha definido, las columnas se pueden crear en cualquier momento, incluso al vuelo mientras la base de datos está corriendo sin necesidad de reiniciarla. Pero no se podrán agregar nuevas *ColumnFamilies* una vez se ha empezado a poblar la tabla.
Los nombres de las columnas siguen la siguiente sintaxis:
familia:nombre
El prefijo *familia* debe estar compuesto por caracteres imprimibles, mientras que el *nombre* puede estar compuesto por cualquier combinación aleatoria de caracteres.
Los controles de acceso y el almacenamiento de los datos en disco se realizan todo a nivel de *column family*. Las columnas de las *column families* se almacenan todas consecutivas en memoria y en disco.

Timestamps

Cada celda de HBase puede contener varias versiones de los mismos datos.
En ese caso, la información tiene una dimensión más: el *timestamp*. En HBase, los *timestamps* son números de 64 bits, y pueden ser asignados automáticamente en función del momento en el que se crea el objeto, o bien pueden ser especificados por la aplicación. El almacenamiento se realiza en orden descendente de tal forma que se puedan leer primero las versiones más recientes. A continuación se expondrá un ejemplo de un diseño tanto conceptual como físico de una tabla de HBase. La tabla en cuestión se llama *webtable*, y está basada en el ejemplo descrito en el artículo de Google.

Diseño conceptual

En este ejemplo se mostrará el diseño de una supuesta tabla creada en HBase llamada Webtable. Esta tabla contiene dos *Column Families*: *contents* y *anchor*. A su vez, la CF anchor tendrá dos columnas*: anchor:cssnsi.com* y *anchor:my.look.ca*, mientas la CF *contents* tan solo tendrá la columna *contents:html*.

Así es como quedaría el diseño conceptual de la tabla Webtable.

Row key Timestamp Column Family contents Column Family anchor
"com.cnn.www" t9 anchor:cnnsi.com = "CNN"
"com.cnn.www" t8 anchor:my.look.ca = "CNN.com"

"com.cnn.www" t6 contents:html =
"<html>..."
"com.cnn.www" t5
contents:html =
"<html>..."
"com.cnn.www" t3
contents:html = "<html>..."

Diseño físico

Físicamente, se representarán las columnas en dos diseños independientes puesto que, a pesar de pertenecer a la misma tabla, nada tienen que ver la una con la otra. Las columnas dentro de cada *column family* se almacenan juntas, y así quedaría representado.

Column Family anchor

Row key Timestamp Column Family anchor
"com.cnn.www" t9 anchor:cnnsi.com = "CNN"
"com.cnn.www" t8 anchor:my.look.ca = "CNN.com"

Column Family contents

Row key Timestamp Column Family contents
"com.cnn.www" t6 contents:html = "<html>..."
"com.cnn.www" t5 contents:html = "<html>..."
"com.cnn.www" t3 contents:html = "<html>..."
Es importante destacar que las celdas vacías que aparecen en la vista conceptual, no aparecen en el modelo físico de datos. Esto significa que físicamente no se almacenará ninguna referencia a estas columnas. Una consulta a la columna *contest:html* con un *timestamp = t8* no devolvería ningún valor, al igual que si se preguntara por *anchor:my.look.ca* con *timestamp = t9*. En caso de no especificar *timestamp*, se devolverán las filas cuyos *timestamps* sean más recientes. Esto es: *contest:html* con *timestamp t6*, *anchor:cnnsi.com* con *timestamp* t9 y *anchor:my.klook.ca* con *timestamp t8*.

Ejemplos reales de uso del SGBD

Facebook

El sitio que más intensamente utiliza HBase en producción actualmente es probablemente, Facebook con el caso particular de su sistema de mensajería. Facebook implementó, allá por el año 2010, su sistema de mensajería instantánea, que se llamaba Messages. Actualmente este producto ha evolucionado y se ha convertido en Facebook Messenger.
Por aquel entonces, Facebook necesitaba una infraestructura que fuera capaz de soportar los 350 millones de usuarios que tenía la plataforma (actualmente cuenta con más de mil millones), que compartían a través de la plataforma más de 15 mil millones de

Bases de datos documentales

A continuación se expondrá en detalle la arquitectura y el modelo de datos de uno de los SGBDs documentales más utilizados: MongoDB.

MongoDB

Introducción

MongoDB (derivado de la palabra inglesa “homongous”) es una base de datos NoSQL Open Source **orientada a documentos**, y ha sido diseñada con la idea de que fuera fácil tanto desarrollar para ella como de ser administrada. La empresa 10gen es la que actualmente se encarga de su mantenimiento y desarrollo. Pero, ¿Qué es una base de datos orientada a documentos? Un documento es una estructura de datos compuesta de pares de campos y valores. Además, estos objetos son muy similares a lo que serían objetos en notación JSON. La principal diferencia con respecto a los almacenes clave – valor es que los valores en MongoDB pueden ser otros documentos, arrays o incluso arrays de documentos.

Pues bien, todos los registros en MongoDB son documentos. Utilizar documentos tiene ciertas ventajas:

* Los documentos son tipos de datos nativos en muchos lenguajes de programación.
* Los documentos embebidos y los arrays minimizan la necesidad de realizar joins muy pesados.
* Tener un esquema dinámico es la base del polimorfismo.

Modelo de datos

MongoDB es un tipo de base de datos de esquema flexible. Esto significa que ha de ser el usuario el que el que determina y declara el esquema de cada una de las tablas justo en el momento de realizar las inserciones. Esta funcionalidad la comparten en cierta forma muchas bases de datos NoSQL.

Esto es bueno y malo a la vez. Por un lado se tiene la flexibilidad de añadir campos cuando son necesarios sin haberlos tenido que haberlos definido de antemano, u obviar algunos en los casos en los que no sean necesarios, pero a la vez se obliga al programador a ser mucho más metódico en su trabajo, puesto que el sistema no avisará de un posible error o despiste en el código. Las bases de datos en MongoDB residen en un host que puede alojar varias bases de datos de forma simultánea almacenadas de forma independiente. Cada una de estas bases de datos puede contener una o más **colecciones**, a partir de las cuales se almacenarán los objetos o documentos.

BSON

BSON es un formato binario que se utiliza para almacenar la información en MongoDB. BSON es la codificación binaria del formato JSON. Esta codificación ha sido elegida porque presenta ciertas ventajas a la hora de almacenar los datos, como la **eficiencia** o la **compresión**.

Básicamente BSON y JSON son los formatos con los que trabaja MongoDB: JSON es el formato con el que se presenta la información a los usuarios y a las aplicaciones y BSON el formato que utiliza MongoDB de forma interna.

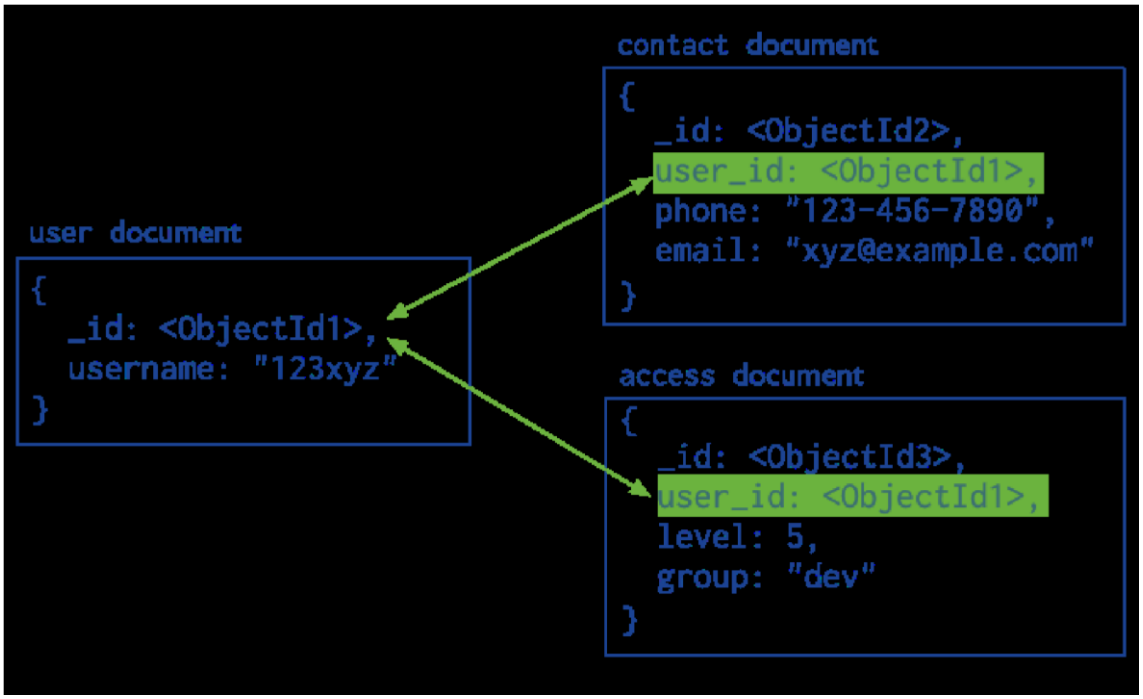
Estructura del documento

El reto principal que tienen los documentos como objetos en MongoDB es el de realizar un buen diseño de los mismos para que sean capaces de representar lo más ampliamente el mundo real. Para ello existen 2 herramientas que permiten a las aplicaciones representar las relaciones entre los datos: las referencias y los datos embebidos.
Una de las decisiones a tomar a la hora de implementar una aplicación que utilice MongoDB como capa de backend de aplicación es la de referenciar o embeber los documentos que se necesite.

Referencias

Las referencias almacenan relaciones entre los datos mediante enlaces entre documentos. Los modelos de datos normalizados utilizan relaciones referenciales entre los documentos. Se usarán modelos de datos normalizados cuando:

- Embeber documentos produzca una duplicación de la información y los costes de mantenimiento sean mayores que las ganancias en lecturas.
- Sea necesario representar un modelo “varios a varios” con cierta complejidad.
- Para modelar conjuntos de datos de mucho tamaño.



Datos embebidos

Los documentos embebidos representan relaciones entre los datos almacenando la información relacionada bajo el mismo documento. Los

documentos permiten embeber estructuras documentales como subdocumentos dentro de un campo o de un array.
En general, se utilizarán modelos embebidos cuando:

- Existan relaciones contenidas entre dos entidades. Estos es, que dos entidades independientes sean habitualmente accedidas a través de sólo una de ellas. Por ejemplo, *Persona* y *Dirección*. Cuando únicamente se permitan búsquedas por *Persona* para obtener su dirección. En ese caso, es preferible tener el objeto *Dirección* embebido dentro del de *Persona*.
- Cuando exista una relación "uno a varios" entre entidades. En este caso, la parte de "varios" suele ir embebida dentro de la de "uno". Los motivos son similares al anterior caso.
- En general, en ambos se trata de minimizar en la medida de lo posible las operaciones de la base de datos. Embebiendo la información se consigue leer o escribir todos los datos simultáneamente y de forma secuencial.

Por otro lado, esta será una peor solución cuando los documentos crecen mucho después de su creación, puesto que se producirá fragmentación y las labores de mantenimiento del documento repercutirán directamente en el rendimiento a la hora de operar con él.

Índices

Las buenas decisiones en lo que a creación de índices se refiere será lo que determine, llegados a un volumen medio-alto de información, que una base de datos en MongoDB tenga unos tiempos de consulta razonables o los tenga muy elevados. Los índices se utilizan para mejorar el rendimiento de las consultas. Es trabajo del administrador de base de datos construir índices en los campos que sean frecuentemente accedidos. Por defecto, MongoDB crea un índice sobre el campo *_id*, el identificador de cada documento. A la hora de crear índices habrá que tener en cuenta una serie de consideraciones (10gen 2014):

- Cada índice requiere de al menos 8KB en disco. Cada índice activo ocupa espacio tanto en disco como en memoria. Hay que tenerlo en cuenta puesto que en algunos casos este espacio puede ser muy elevado.

- Los índices tienen un impacto negativo para las operaciones de escritura. Habrá que evaluar el uso que se le va a dar al sistema, puesto que los sistemas que reciban muchas escrituras y pocas lecturas, requerirán también tareas de actualización para cada índice que exista.
- Los índices no afectan para nada en las operaciones de lecturas sobre campos del documento que no han sido indexados.

Usos reales del SGBD Foursquare

Foursquare es una red social cuya actividad consiste en permitir a sus usuarios realizar *check-in* en lugares concretos (marcar lugares como visitados a medida que el usuario los visita y compartir la localización con amigos y contactos). Cuando Foursquare decidió migrar a MongoDB, su capa de backend consistía en una única base de datos relacional.
Debido al crecimiento exponencial que experimentó desde su creación en 2009 hasta pocos años después, los ingenieros de Foursquare decidieron evaluar, entre otras soluciones, bases de datos NoSQL. Finalmente encontraron en MongoDB la base de datos que les permitía solucionar tanto sus necesidades más inmediatas, como las, previsiblemente, pudieran surgirles más adelante.
Las características que apreciaron de MongoDB fueron:

- El particionado automático, mediante el cual simplemente tienes que añadir nodos al clúster a medida que se van necesitando y el software se encarga de lo demás.
- La indexación geográfica de la que dispone MongoDB, la cual les permitía realizar búsquedas basadas en una ubicación espacial concreta. --- Los ReplicaSet, mediante los cuales se consigue alta disponibilidad y redundancia de nodos en caso de caídas.
- Su modelo de datos dinámico y adaptable a las necesidades en cada punto del

Bases de datos orientadas a grafos

Neo4

Introducción

Neo4j es una base de datos basada en grafos. Neo4j está escrita en Java y ha sido y es desarrollada por la empresa Sueca Neo Technology. Neo4j es probablemente la base de datos basada en grafos más conocida y más utilizada de todas.
Más adelante se abordará el tema con más detenimiento, pero vale la pena mencionar que la principal característica de las bases de datos orientadas a grafos es que están especialmente ideadas para representar relaciones entre datos y datos conectados.
Las bases de datos orientadas a grafo son, probablemente, las que más difieren del resto de bases de datos NoSQL, primero porque abordan su diseño (el diseño de la propia base de datos) de una forma completamente distinta, y segundo porque de entre todas las bases de datos NoSQL son las que, probablemente, más se parezca en su manera forma de realizas las operaciones a las bases de datos relacionales.

Grafos

Antes de continuar, es necesario hacer un breve repaso sobre lo que son los grafos y en lo que consiste la teoría de grafos (Robinson, Webber and Eifrem 2013).
Un **grafo** no es más que un conjunto de **nodos** que mantienen una serie de **relaciones** entre sí. Las relaciones tienen nombre y sentido, y siempre tienen un nodo de inicio y uno de fin. Tanto los nodos como las relaciones poseen **propiedades**. Los **recorridos** son consultas que se realizan sobre una parte o la totalidad del grafo.

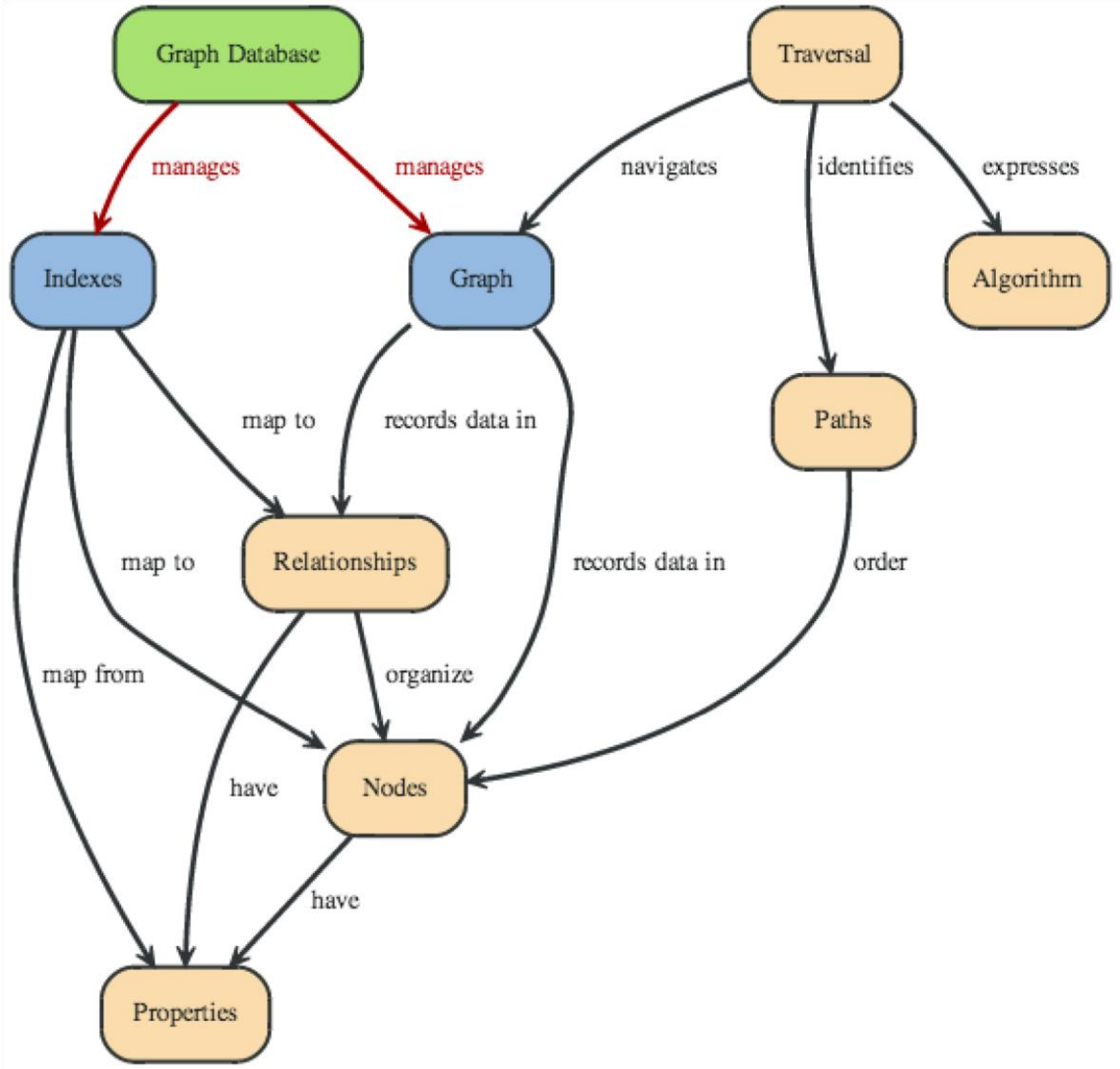
Las siguientes expresiones resumen qué es cada uno de estos elementos y como puede utilizarse conjuntamente con el resto de elementos (Neo Technology 2014):

- Un **grafo** almacena datos en **nodos**
- Un **grafo** almacena datos en **relaciones**
- Los **nodos** se organizan en **relaciones**
- Los nodos y relaciones tienen propiedades
- Un **recorrido** navega un **grafo**
- Un **recorrido** identifica **rutas**
- Las **rutas** ordenan **nodos**
- Los **índices** son **recorridos** especiales para encontrar **nodos** o **relaciones**

Realmente todos los elementos comentados aquí forman parte de la fisionomía de los grafos excepto los índices. Los índices son propios de la base de datos orientada a grafos. Es la propia base de datos la que los genera y mantiene, y los utiliza para acelerar según qué tipo de consultas.

Los nodos, también conocidos como vértices, representan entidades, habitualmente personas o cosas, mientras que las relaciones, también conocidas como aristas o arcos, son elementos que generan una interconexión entre los nodos.

En teoría de grafos, un grafo se representa como G = (V, E), donde V es el conjunto de vértices y E el conjunto de aristas (edges).



Aplicaciones reales

Muchos estudios sostienen que la representación de la información como grafo es algo inherente a la propia naturaleza de los datos. Representar los problemas del mundo real en forma de grafo es un pequeño paso más allá de lo que hace el resto de bases de datos y, probablemente, el enfoque que muchos problemas necesitan para ser abordados de una manera más sencilla.

Uno de los campos con más potencial en explotar este tipo de bases de datos es el de la **química** y la **biología**. La información sobre los compuestos químicos en su forma más básica (átomos y enlaces) se acopla perfectamente a la estructura y visión que aportan los grafos.

Otro caso particularmente interesante y que cuenta con bastante literatura al respecto es el de las **redes sociales**.

Los grafos aquí no solo tienen porque representar personas y las relaciones que mantienen entre sí, sino también enlaces a sitios web, elementos multimedia o mensajes.

Redes sociales con un volumen muy grande de información, que previsiblemente puedan generar grafos de tamaños enormes, pueden estar interesados en algoritmos de caminos mínimos entre dos puntos, o recorridos maximizando valores o pesos.

La propia **web** como tal es en esencia un grafo de información enlazada entre sí. Los links establecen la forma más básica de unir páginas o comunidades de información entre sí. El motor que Google utiliza para determinar la relevancia de los sitios web tiene su base en recolectar y analizar los enlaces que apuntan desde y hasta cada página web.

Neo4j tiene soporte completo en lo que a transacciones ACID se refiere. Esto convierte a Neo4j en una base de datos transaccional, y por este motivo se acerca mucho más que el resto de bases de datos NoSQL a las bases de datos relacionales.

Arquitectura

Alta disponibilidad

Neo4j ha sido diseñado para correr tanto en modo single-node como en un modo distribuido en el que siempre existe un maestro como nodo principal del clúster, pudiendo existir también cero, uno o más esclavos. La lógica que se sigue en este apartado de Neo4j es muy similar a la que siguen otras bases de datos, como por ejemplo, MongoDB.

La estructura que sigue Neo4j para proporcionar al sistema de alta disponibilidad es similar a la que se suele emplear de maestro-esclavo, con la salvedad de que todos los nodos del sistema pueden recibir operaciones de escritura, y no solo el maestro.

Cada nodo contiene la lógica necesaria para coordinarse y comunicarse con el resto de miembros del clúster. Cuando un nodo arranca, examinará sus ficheros de configuración. Si pertenece ya a un clúster, pasará a formar parte del mismo como esclavo. En caso de que pertenezca a un clúster que no exista, se creará y se colocará como maestro.

Cuando un esclavo se hace cargo de una escritura en el sistema, de forma síncrona deberá escribirla también en el maestro. Recordar que será este último el encargado de replicar la información por el resto de esclavos del sistema. Tras haber recibido la orden de escritura, la operación se realizará síncronamente entre el esclavo y el maestro. Para ello, el esclavo debe estar completamente actualizado con el maestro, es decir, mantener exactamente las mismas bases de datos. Una vez que el maestro haya finalizado su escritura, el esclavo comenzará su transacción.

A la vez que se está ejecutando la operación de escritura anteriormente descrita, el maestro puede implementar una replicación optimista. La replicación optimista es un método por el cual, en el mismo momento en el que el maestro intenta realizar la escritura en su base de datos local, también intenta hacer llegar a los esclavos los nuevos datos. Esta replicación evidentemente puede fallar, pero en ese caso se dará por buena la transacción de escritura y se procederá a realizar una escritura retardada en los esclavos. Cuando un nodo esclavo permanece

indisponible por un periodo determinado de tiempo, el resto de nodos del sistema le marcan como nodo fallido. Cuando el nodo vuelve a desempeñar su funcionamiento normal, deberá actualizarse por completo antes de poder volver a poder formar parte del clúster.
Si es el maestro el nodo que cae, debido a problemas en la red o debido a problemas de hardware, un nuevo maestro es elegido de entre todos los esclavos del sistema, de forma que su rol pasa de esclavo a maestro. Normalmente, entre que el maestro del sistema cae y uno nuevo toma el reemplazo transcurren unos segundos. Durante ese lapso de tiempo, el sistema no acepta escrituras.

Backup

Los backups en Neo4j son copias que se realizan de una base de datos mientras ésta está funcionando, vía red o de forma local.

Existen dos tipos de copias de seguridad, las *full* y las *incremental*.

- Las *full backup* son copias que se realizan de absolutamente toda la base de datos por completo en caliente, es decir, mientras la base de datos continúa funcionando como lo hace normalmente. Además, no solicita ningún bloqueo. Esto significa que, mientras se está realizando la copia de seguridad, otras operaciones y otras transacciones pueden estar teniendo lugar a la vez. Para asegurarse de que la copia de seguridad es completamente consistente y contiene toda la información de la base de datos hasta el momento de la copia, todas las transacciones que den comienzo después de haber iniciado la operación de *full backup* serán después también ejecutadas en la copia.
- Los *incremental backup* no son copias de la base de datos directamente. En su lugar, lo que guardan estas copias de seguridad son los logs de transacciones que han tenido lugar desde el último backup, *full* o *incremental*. De esta manera, el tiempo y el espacio que ocupa este tipo de backups será menor, pero también hay que tener en cuenta de que, en caso de tener que ser restaurado, es necesario partir del *incremental* o el *full backup* del que se ha partido.

Para restaurar un backup, lo único que es necesario es reemplazar la carpeta donde Neo4j aloja su base de datos con el contenido del backup. Hay que tener en cuenta que los backups que implementa Neo4j son copias de la base de datos completamente funcionales.

Modelo de consultas

Neo4j tiene asociado un potente lenguaje que permite realizar querys sobre los datos almacenados en la base de datos. Este lenguaje se denomina Cypher.

Cypher es un lenguaje de consulta declarativo y orientado a grafos, que realizar consultas y actualizaciones sobre la estructura del grafo. Cypher es un lenguaje sencillo pero potente, de forma que el programador puede centrarse en el dominio más que en tratar de acceder por su cuenta mediante cualquier otro tipo de lenguajes a la base de datos.
Como lenguaje declarativo, Cypher se centra en intentar expresar el *qué* se quiere obtener del grafo en lugar del *cómo* obtenerlo. Este enfoque hace que realizar optimizaciones en las consultas de la base de datos sea simplemente un detalle de implementación, y le abstrae de todo el componente físico propio del

sistema gestor de la base de datos (como por ejemplo, nuevo índices). Cypher está inspirado en muchos lenguajes de programación, incluso en lenguajes propios de bases de datos. Sin ir más lejos, Cypher toma prestado de SQL la mayoría de su sintaxis, así como muchas de las cláusulas como *WHERE* u *ORDER BY*. También toma prestados ciertos aspectos relacionados con los patrones y las expresiones regulares de SPARQL, y otras tantas funciones de lenguajes como Python o Haskell.

Ejemplos de uso real del SGBD

eBay

Hace menos de un año eBay compró una empresa llamada Shutl, especializada en realizar envíos. Un servicio especialmente popular que tienes esta pequeña empresa es la modalidad de envío en el mismo día, que te permite enviar una mercancía de un lado a otro de Reino Unido en menos de
24h. El record, que puede visualizarse en su web[56], es de 13’ 57’’.

Mapeo Objeto / Relacional (ORM)

Al desarrollar una aplicación siguiendo el paradigma orientado a objetos, los programadores se enfrentan, entre otros desafíos, al problema de la persistencia de los datos, debido a las diferencias que existen entre el modelo relacional y el modelo orientado a objetos. Este problema la mayoría de los casos es solucionado con la ayuda de ciertas herramientas que se encargan de generar de manera automática el acceso a datos, abstrayendo al programador de este problema.

El Modelo Relacional es un modelo de datos basado en la lógica de predicado y en la teoría de conjuntos para la gestión de una base de datos. Siguiendo este modelo se puede construir una base de datos relacional que no es más que un conjunto de una o más tablas estructuradas en registros (filas) y campos (columnas), que se vinculan entre sí por un campo en común. Sin embargo, en el Modelo Orientado a Objetos en una única entidad denominada objeto, se combinan las estructuras de datos con sus comportamientos. En este modelo se destacan conceptos básicos tales como objetos, clases y herencia.

Entre estos dos modelos existe una brecha denominada desajuste por impedancia dada por las diferencias entre uno y otro. Una de las diferencias se debe a que en los sistemas de bases de datos relacionales, los datos siempre se manejan en forma de tablas, formadas por un conjunto de filas o tuplas; mientras que en los entornos orientados a objetos los datos son manipulados como objetos, formados a su vez por objetos y tipos elementales.

Además en el modelo relacional no se puede modelar la herencia que aparece en el modelo orientado a objetos y existen también desajustes en los tipos de datos, ya que los tipos y denotaciones de tipos asumidos por las consultas y lenguajes de programación difieren. Esto concierne a tipos atómicos como integer, real, boolean, etc. La representación de tipos atómicos en lenguajes de programación y en bases de datos pueden ser significativamente diferentes, incluso si los tipos son denotados por la misma palabra reservada, ej.: integer. Esto ocurre también con tipos complejos como las tablas, un tipo de datos básico en SQL ausente en los lenguajes de programación.

Para atenuar los efectos del desajuste por impedancia entre ambos modelos existen varias técnicas y prácticas como los Objetos de Acceso a Datos (Data Acces Objects o DAOs), marcos de trabajo de persistencia (Persistence Frameworks), mapeadores Objeto/Relacionales (Object/Relational Mappers u ORM), consultas nativas (Native Queries), lenguajes integrados como PL-SQL de Oracle y T-SQL de SQL Server; mediadores, repositorios virtuales y bases de datos orientadas a objetos .

Mapeo Objeto/Relacional

El mapeo objeto-relacional es una técnica de programación para convertir datos del sistema de tipos utilizado en un lenguaje de programación orientado a objetos al utilizado en una base de datos relacional. En la práctica esto crea una base de datos virtual orientada a objetos sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (esencialmente la herencia y el polimorfismo).

Las bases de datos relacionales solo permiten guardar tipos de datos primitivos (enteros, cadenas de texto, etc.) por lo que no se pueden guardar de forma directa los objetos de la aplicación en las tablas, sino que estos se deben de convertir antes en registros, que por lo general afectan a varias tablas. En el momento de volver a recuperar los datos, hay que hacer el proceso contrario, se deben convertir los registros en objetos. Es entonces cuando ORM cobra importancia, ya que se encarga de forma automática de convertir los objetos en registros y viceversa, simulando así tener una base de datos orientada a objetos.

Entre las ventajas que ofrecen los ORM se encuentran: rapidez en el desarrollo, abstracción de la base de datos, reutilización, seguridad, mantenimiento del código, lenguaje propio para realizar las consultas. No obstante los ORM traen consigo algunas desventajas como el tiempo invertido en el aprendizaje. Este tipo de herramientas suelen ser complejas por lo que su correcta utilización requiere un espacio de tiempo a emplear en conocer su funcionamiento adecuado para posteriormente aprovechar todo el partido que se le puede sacar. Otra desventaja es que las aplicaciones suelen ser algo más lentas. Esto es debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

En el mapeo objeto-relacional encontramos el uso de algunos patrones de diseño como el Repository y el Active Record.

Patrón Repository

El patrón Repository utiliza un repositorio para separar la lógica que recupera los datos y los mapea al modelo de entidades, de la lógica del negocio que actúa en el modelo. El repositorio media entre la capa de fuente de datos y la capa de negocios de la aplicación; encuesta a la fuente de datos, mapea los datos obtenidos de la fuente de datos a la entidad de negocio y persisten los cambios de la entidad de negocio a la fuente de datos. En la figura 1 se muestra un esquema patrón Repository.

Los repositorios son puentes entre los datos y las operaciones que se encuentran en distintos dominios. Un repositorio elabora las consultas correctas a la fuente de datos y mapea los resultados a las entidades de negocio expuestas externamente. Los repositorios eliminan las dependencias a tecnologías específicas proveyendo acceso a datos de cualquier tipo [1].

El patrón de diseño Repository puede ayudar a separar las capas de una aplicación web ASP.NET ya que provee una arquitectura de 3 capas separadas, lo que mejora el mantenimiento de la aplicación y ayuda a reducir errores. Además facilita las pruebas unitarias.

Patrón Active Record

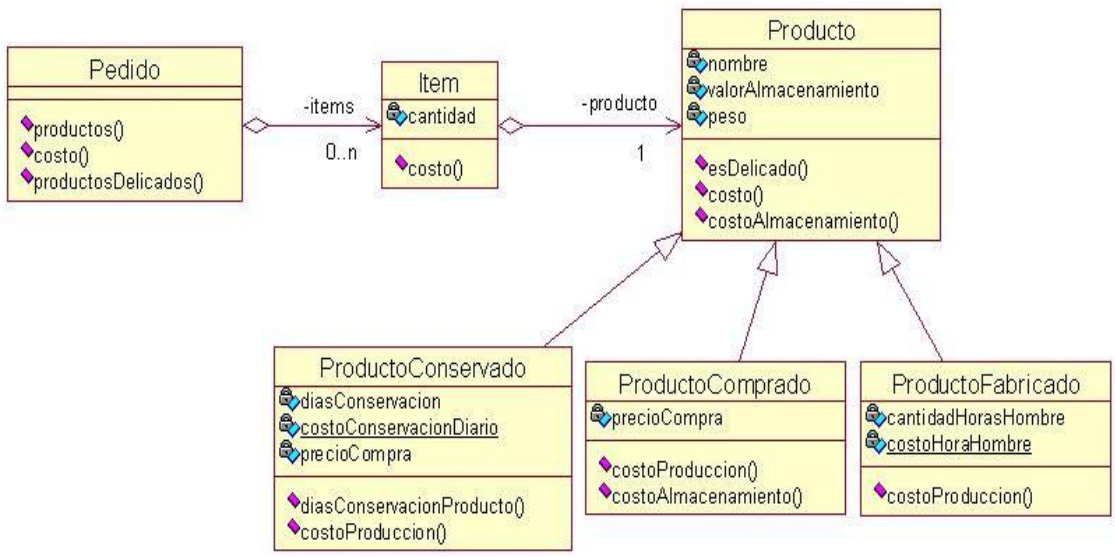
Active Record es un patrón en el cual, el objeto contiene los datos que representan a una fila (o tupla) de nuestra tabla o vista, además de encapsular la lógica necesaria para acceder a la base de datos. De esta forma el acceso a datos se presenta de manera uniforme a través de la aplicación (lógica de negocio + acceso a datos en una misma clase). En la figura 2 se muestra un esquema del patrón Active Record.

Una clase Active Record consiste en el conjunto de propiedades que representa las columnas de la tabla más los típicos métodos de acceso como las operaciones CRUD (Create, Read, Update, Delete), búsqueda (Find), validaciones, y métodos de negocio [5]. Este patrón constituye la aproximación más obvia, poniendo el acceso a la base datos en el propio objeto de negocio. De este modo es evidente como manipular la persistencia a través de él mismo. Gran parte de este patrón viene de un Domain Model y esto significa que las clases están muy cercanas a la representación en la base de datos. Cada Active Record es responsable de sí mismo, tanto en lo relacionado con persistencia como en su lógica de negocio .

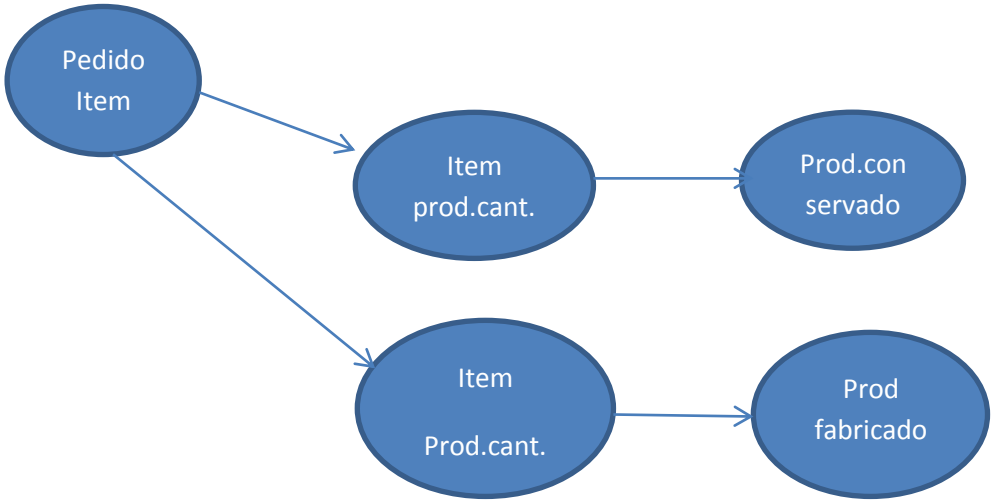
Desajuste por impedancia

Se entiende pro desajuste pro impedancia a las dificultades de traslación de un mundo a otro, es decir de un modelo orientado a objetos a una base de datos relacional.

Ejemplo

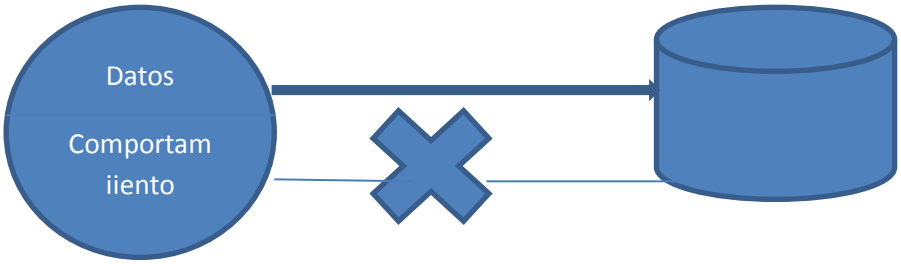


Los productos siguen en memoria una estructura de grafos



Estos objetos viven en el ambiente. El tema es que la memoria es limitada y necesito poder almacenar los objetos en un medio que me permita recuperarlos el día de mañana (concepto que llamaremos **persistencia**).
Entonces tengo distintas opciones:
- Puedo conservar la misma estructura en una base de objetos,
- o puedo utilizar una base de datos relacional como repositorio de información.
La pregunta es: este grafo que acabamos de dibujar ¿encaja justo en una estructura basada en el álgebra relacional?

• Originalmente podemos establecer una equivalencia entre el concepto de Tabla/Clase, y Registro/instancias de esa clase. Pero más adelante empezaremos a tener ciertas dificultades para que este mapping sea tan lineal.



• Los objetos tienen comportamiento, las tablas sólo permiten habilitar ciertos controles de integridad (constraints) o pequeñas validaciones antes o después de una actualización (triggers). Los stored procedures no están asociados a una tabla, lo que genera el mismo problema que en la programación estructurada: toco un campo y necesito al menos recompilar todos los stored procedures asociados. Cada actualización impacta en los datos por un lado y en los programas que actualizan los datos (Find/Replace masivo).

• Los objetos encapsulan información, no para protegerse (siempre puedo hackearlo), sino para favorecer la abstracción del observador. Una tabla no tiene esa habilidad: si tengo una entidad con un atributo ACTIVO que es VARCHAR2(1), tengo que entrar al constraint check o bien tirar un SELECT para saber si el ACTIVO es 1 ó 0, “S” ó “N”, e inferir en el mejor de los casos qué significan los valores de ese campo. Puedo documentarlo en un diccionario de datos, algo que está ajeno a la base que estoy tocando y que necesita una sincronización propia de gente pulcra y obsesiva.

• En objetos puedo generar interfaces, que permiten establecer un contrato entre dos partes, quien publica un determinado servicio y quien usa ese servicio. En el álgebra relacional la interfaz no se convierte en ninguna entidad, con lo cual hay que contar con ciertos trucos para generar objetos que sólo encapsulan comportamiento.

• La herencia es una relación estática que se da entre clases, que favorece agrupar comportamiento y atributos en común. Cuando yo instancio un objeto recibo la definición propia de la clase y de todas las superclases de las cuales heredo. En el modelo lógico de un Diagrama Entidad/Relación yo tengo supertipos y subtipos, pero en la implementación física las tablas no tienen el concepto de herencia. Hay que hacer adaptaciones que se verán mas adelante.

• Al no existir el comportamiento en las tablas y no estar presente el concepto de interfaz _ no hay polimorfismo en el álgebra relacional: yo puedo recibir un objeto sin saber exactamente de qué clase es, y a mí no me interesa averiguarlo, sólo me concentro en lo que le puedo pedir

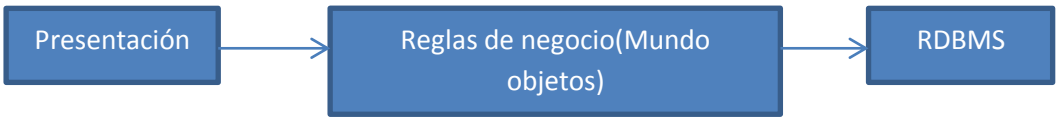
y en su contrato. Al tirar un query sobre una tabla, necesito saber de qué tabla se trata (claro, al menos en los queries tradicionales, que no acceden a los metadatos).

Todo esto es lo que se conoce como “Impedance mismatch” _ (dificultades por impedancia) las dificultades de traslación de los dos mundos.

De todas maneras, estamos usando la base de datos relacional sólo como repositorio para almacenar los datos de los objetos. El comportamiento sigue estando en
cuando se instancian. Por ese motivo tenemos lposibilidad de adaptar el modelo relacional al de objetos…

Algunas otras cosas a tener en cuenta:

• ¿Cómo manipulo los datos en una base relacional? A través de un lenguaje declarativo, el SQL donde digo qué quiero. Y muchas veces nos vamos a encontrar en la disyuntiva de algo muy fácil de hacer con un query/stored procedure vs. tener que armar una “regla de negocio” en objetos que implique muchas más líneas de código.
El tema es que muchas veces no puedo zafar de tener:

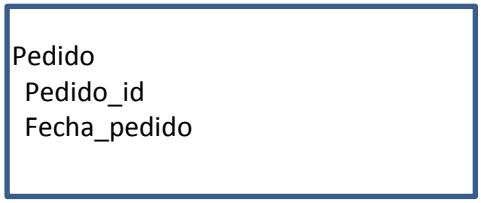


Eso tiene muchas contras:
• estoy mezclando tecnologías
• pierdo control cuando hay un cambio: es difícil así conservar el encapsulamiento.

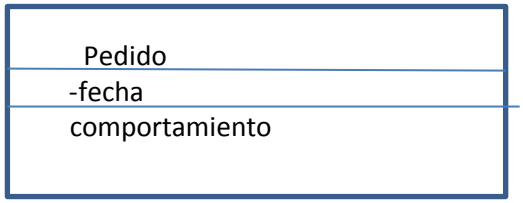
<u>Ejemplos de cómo se resolverían algunos casos</u>

Supongamos que Pedido tiene una fecha de pedido.

Modelo relacional : tabla pedido



Modelo objetos: clase pedido

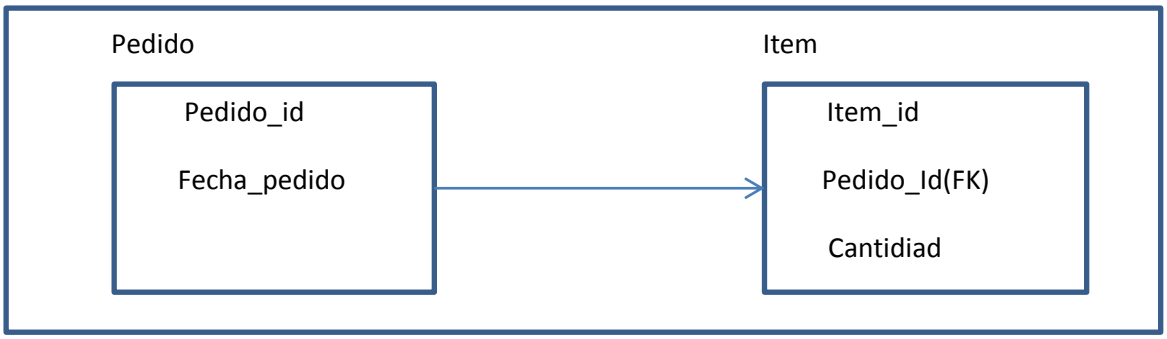


Fíjense qué diferencia tenemos en la tabla vs. la clase Pedido. Además del comportamiento (en la cajita de abajo), necesitamos una clave que identifique unívocamente a cada registro de la tabla Pedido.
Estamos hablando del mismo registro si tienen el mismo PEDIDO_ID.

En objetos se maneja el concepto de identidad: cada objeto sabe que es él y ningún otro objeto, entonces no es necesario trabajar con un identificador. Cuando yo referencio a un objeto, se que le estoy enviando un mensaje a ése objeto, no necesito identificarlo porque alguien me pasó la referencia a él. Si no lo conozco no le puedo pedir cosas.
Hablamos del mismo objeto si objeto1 == objeto2.

Agregamos el vínculo con Item

Modelo relacional



Fíjense que como el Pedido es la tabla madre y los ítems son hijos, la clave principal de ITEM se compone del Identificador del Pedido y el del Item.

Esto implica que si tengo el Pedido 1 con 2 ítems y el Pedido 2 con 2 ítems, la tabla Item tendría la siguiente información: (1,1) (1,2) (2,1) (2,2)

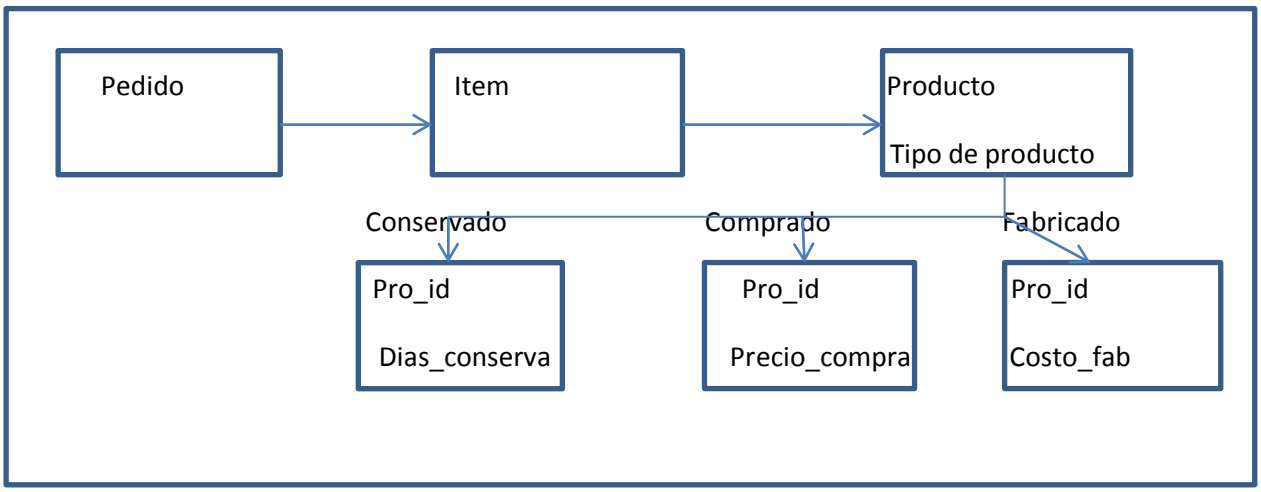
La otra opción es definir al ítem como autoincremental , en este caso los valores de las columnas serían (1,1) (2,1) (3,2) (4,2)

Muchas veces en Objetos se prefiere trabajar con claves autoincrementales, que dependan de una secuencia manejada por la base de datos. ¿Por qué? Porque es molesto definir un campo donde tenga que generar IDs incrementales según el pedido. Es más cómodo que la Base de Datos se encargue de generar ITEM_IDs por tabla, asegurándome por la misma Base que no se repiten (cosa mucho más compleja de asegurar cuando tengo más de una Virtual Machine/Ambiente dando vueltas).

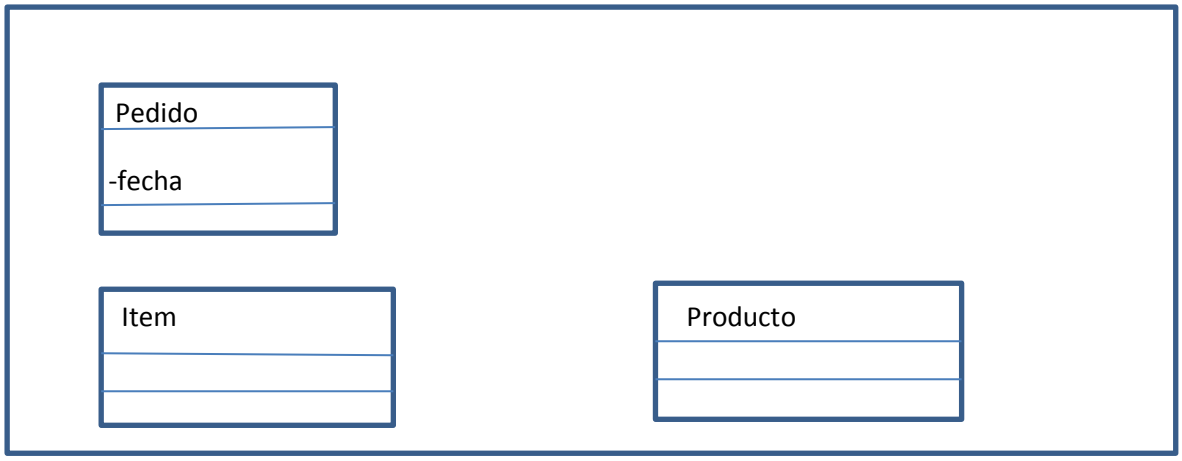
¿Cómo juega el ID dentro del objeto Pedido e Item? Bueno, quizás sí tenga un atributo Id en Pedido e Item pero si yo trabajo en objetos no quiero que el que use a ese objeto esté preguntando por el Id. Depende del contexto, el id puede ser un atributo privado que no tenga getter hacia fuera.

Agregamos el vínculo con Product

Modelo relacional



Modelo objetos



Donde la clase producto tiene subclases.

El modelo relacional de arriba se corresponde a la clase Producto con sus subclases. Entre el modelo lógico relacional y el modelo de objetos parece no haber mucha diferencia. Ahora bien, al implementar físicamente este modelo podemos elegir 3 opciones:
Implementar una tabla por cada clase, implementar una tabla pro subclase o implementar una única tabla.

Cuando trabajamos con objetos polimórficos tiene ventajas y desventajas

Concepto	A favor	En contra
1 tabla	□ Facilidad/> Performance para queries polimórficos □ Evito generar muchas tablas	□ Campos no utilizados
1 tabla por Cada clase (n+1)	□ Es el modelo “ideal” según las reglas de Normalización □ Permite trabajar con queries polimórficos (no necesito saber en qué tabla está la información) □ Permite establecer campos no nulos para cada subclase	□ Es la opc, que mas tablas crea
1 tabla por Subclase	□ Permite establecer campos no nulos y no req. Discriminar	□ Cada subclase repite campos heredados

Una vez adoptada la decisión de trabajar con bases relacionales, la pregunta es qué hacer primero:

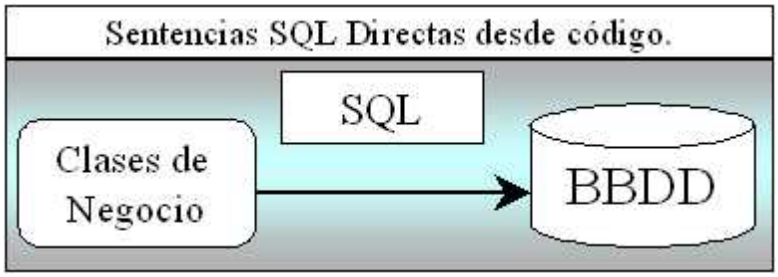
a) Generamos el modelo relacional y luego adaptamos el modelo de objetos en base a las tablas generadas
b) Generamos el modelo de objetos y en base a éste se crean las tablas.

La opción a) supone que es más importante la forma en que guardo los datos que las reglas de negocio que modifican esos datos. Pero a veces no nos queda otra chance, si partimos de un sistema construido o enlatado.
En la opción b) el desarrollo con objetos no se ve ensuciado por restricciones propias de otra tecnología.

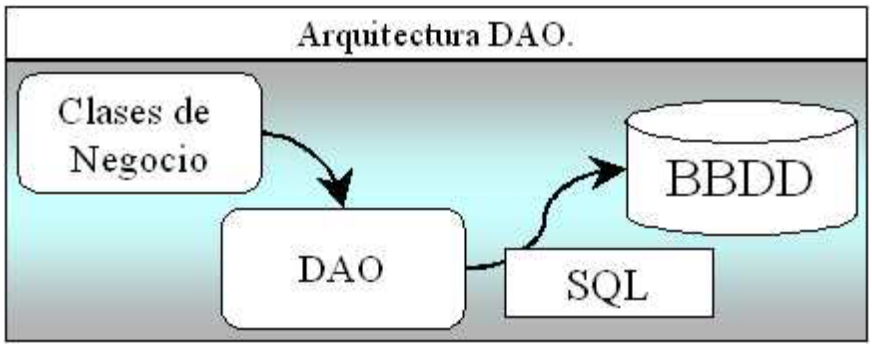
HIBERNATE

Persistencia de los datos y posibilidades que estas ofrecen

En el diseño de una aplicación una parte muy importante es la manera en la cual accedemos a nuestros datos en la base de datos (en adelante BBDD) determinar esta parte se convierte en un punto crítico para el futuro desarrollo.
La manera tradicional de acceder seria a través de JDBC directamente conectado a la BBDD mediante ejecuciones de sentencias SQL:

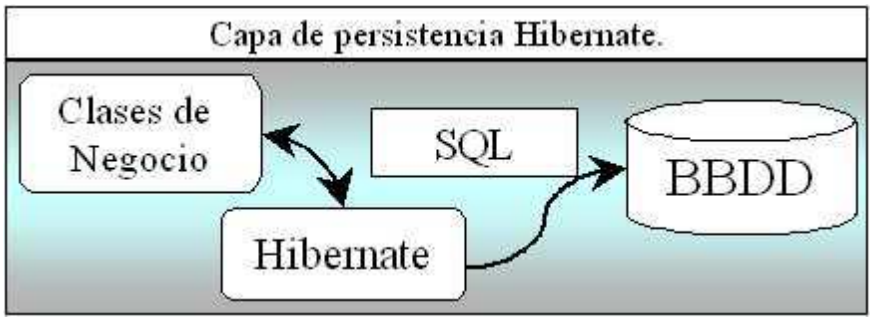


Esta primera aproximación puede ser útil para proyectos o arquitecturas sin casi clases de negocio, ya que el mantenimiento del código está altamente ligado a los cambios en el modelo de datos relacional de la BBDD, un mínimo cambio implica la revisión de casi todo el código así como su compilación y nueva instalación en el cliente.
Aunque no podemos desechar su utilidad. El acceso a través de SQL directas puede ser utilizado de manera puntual para realizar operaciones a través del lenguaje SQL lo cual seria mucho mas efectivo que la carga de gran cantidad de objetos en memoria. Si bien un buen motor de persistencia debería implementar mecanismos para ejecutar estas operaciones masivas sin necesidad de acceder a este nivel.
Una aproximación mas avanzada seria la creación de unas clases de acceso a datos (
DAO Data Acces Object). De esta manera nuestra capa de negocio interactuaría con la capa DAO y esta sería la encargada de realizar las operaciones sobre la BBDD.



Los problemas de esta implementación siguen siendo el mantenimiento de la misma así como su portabilidad. Lo único que podemos decir es que tenemos el código de transacciones encapsulado en las clases DAO. Un ejemplo de esta arquitectura podría ser Microsoft ActiveX Data Object (ADO).

Y como encaja Hibernate en todo esto?. Lo que parece claro es que debemos separar el código de nuestras clases de negocio de la realización de nuestras sentencias SQL contra la BBDD. Por lo tanto Hibernate es el puente entre nuestra aplicación y la BBDD, sus funciones van desde la ejecución de sentencias SQL a través de JDBC hasta la creación, modificación y eliminación de objetos persistentes.



Con la creación de la capa de persistencia se consigue que los desarrolladores no necesiten conocer nada acerca del esquema utilizado en la BBDD. Tan solo conocerán el interface proporcionado por el motor de persistencia. De esta manera conseguimos separar de manera clara y definida, la lógica de negocios de la aplicación con el diseño de la BBDD.
Esta arquitectura conllevará un proceso de desarrollo más costoso pero una vez se encuentre implementada las ventajas que conlleva merecerán la pena. Es en este punto donde entra en juego Hibernate. Como capa de persistencia desarrollada tan solo tenemos que adaptarla a nuestra arquitectura.

Qué es hibernate

Hibernate es una capa de persistencia objeto/relacional y un generador de sentencias sql. Te permite diseñar objetos persistentes que podrán incluir polimorfismo, relaciones, colecciones, y un gran número de tipos de datos. De una manera muy rápida y optimizada podremos generar BBDD en cualquiera de los entornos soportados : Oracle, DB2, MySql, etc.. Y lo más importante de todo, es **open source**, lo que supone, entre otras cosas, que no tenemos que pagar nada por adquirirlo.
Uno de los posibles procesos de desarrollo consiste en, una vez tengamos el diseño de datos realizado, mapear este a ficheros XML siguiendo la DTD de mapeo de Hibernate.
Desde estos podremos generar el código de nuestros objetos persistentes en clases Java y también crear BBDD independientemente del entorno escogido.
Hibernate se integra en cualquier tipo de aplicación justo por encima del contenedor de datos.
Una posible configuración básica de hibernate es la siguiente:

Podemos observar como Hibernate utiliza la BBDD y la configuración de los datos para proporcionar servicios y objetos persistentes a la aplicación que se encuentre justo por arriba de él.

Mapas de objetos relacionales en ficheros XML

Antes de empezar a escribir una línea se debe tener realizado el análisis de la aplicación y la estructura de datos necesaria para su implementación.
El objetivo es conseguir desde el diseño de un objeto relacional un fichero XML bien formado de acuerdo a la especificación Hibernate. El trabajo consistirá en *traducir* las propiedades, relaciones y componentes a el formato XML de Hibernate.
Asumiremos que objeto persistente, registro de tabla en BBDD y objeto relacional son la misma entidad.

Estructura del fichero XML

Esquema general de un fichero XML de mapeo es algo como esto:
<Encabezado XML>
<Declaración de la DTD>
<class - Definición de la clase persistente>
<id - Identificador>
<generator - Clase de apoyo a la generación automática de OID's>
<component - Componentes, son las columnas de la tabla>

A continuación se detallan las características, parámetros y definición de las etiquetas arriba utilizadas así como de algunas otras que nos serán de utilidad a la hora de pasar nuestro esquema relacional a ficheros de mapeo XML.

Declaración de la DTD.
El documento DTD que usaremos en nuestros ficheros XML se encuentra en cada distribución de Hibernate en el propio **.jar** o en el directorio src.

Elemento Raíz **<hibernate-mapping>.** Dentro de él se declaran las clases de nuestros objetos persistentes. Aunque es posible declarar más de un elemento <class> en un mismo fichero XML, no debería hacerse ya que aporta una mayor claridad a nuestra aplicación realizar un documento XML por clase de objeto persistente.

<class> Este es el tag donde declaramos nuestra clase persistente. Una clase persistente equivale a una tabla en la base de datos, y un registro o línea de esta tabla es un objeto persistente de esta clase. Entre sus posibles atributos destacaremos:
1. *name* : Nombre completo de la clase o interface persistente. Deberemos incluir el package dentro del nombre.
2. *table* : Nombre de la tabla en la BBDD referenciada. En esta tabla se realizará las operaciones de transacciones de datos. Se guardarán, modificarán o borrarán registros según la lógica de negocio de la aplicación.
3. *discriminator-value* : Permite diferenciar dos sub-clases. Utilizado para el polimorfismo.
4. *proxy* : Nombre de la clase Proxy cuando esta sea requerida.

<id> Permite definir el identificador del objeto. Se corresponderá con la clave principal de la tabla en la BBDD. Es interesante definir en este momento lo que será para nuestra aplicación un OID(Identificador de Objeto). Tenemos que asignar identificadores únicos a nuestros objetos persistentes, en un primer diseño podríamos estar tentados a asumir un dato con

significado dentro de la capa de negocios del propio objeto fuese el identificador, pero esta no sería una buena elección. Imaginemos una tabla de personas con su clave primaria N.I.F.. Si el tamaño, estructura o composición del campo cambiase deberíamos realizar este cambio en cada una de las tablas relacionadas con la nuestra y eventualmente en todo el código de la aplicación.
Utilizando OID's (Identificadores de Objetos) tanto a nivel de código como en BBDD simplificamos mucho la complejidad de nuestra aplicación y podemos programar partes de la misma como código genérico.
El único problema en la utilización de OID es determinar el nivel al cual los identificadores han de ser únicos. Puede ser a nivel de clase, jerarquía de clases o para todas la aplicación, la elección de uno u otro dependerá del tamaño del esquema relacional.
1. *name* : Nombre lógico del identificador.
2. *column* : Columna de la tabla asociada en la cual almacenaremos su valor.
3. *type* : Tipo de dato.
4. *unsaved-value* ("any|none|null|id_value"): Valor que contendrá el identificador de la clase si el objeto persistente todavía no se ha guardado en la BBDD.
5. *generator* : clase que utilizaremos para generar los oid's. Si requiere de algún parámetro este se informa utilizando el elemento <paramater name="nombre del parámetro">.

Hibernate proporciona clases que generan automaticamente estos OID evitando al programador recurrir a trucos como coger la fecha/hora del sistema. Entre ellas caben destacar :

1.*vm* : Genera identificadores de tipo long, short o int. Que serán únicos dentro de una JVM.
2. *sequence* : Utiliza el generador de secuencias de las bases de datos DB2, PostgreSQL, Oracle, SAP DB, McKoi o un generador en Interbase. El tipo puede ser long, short o int.
3. *hilo* : Utiliza un algoritmo hi/lo para generar identificadores del tipo long, short o int. Estos OID's son únicos para la base de datos en la cual se generan. En realidad solo se trata de un contador en una tabla que se crea en la BBDD. El nombre y la columna de la tabla a utilizar son pasados como parámetros, y lo único que hace es incrementar/decrementar el contador de la columna con cada nueva creación de un nuevo registro. Así, si por ejemplo decimos tener un identificador único por clase de objetos persistentes, deberíamos pasar como parámetro tabla *table_OID* y como columna el nombre de la clase *myclass_OID*.
4. *uuid.string* : Algoritmo UUID para generar códigos ASCII de 16 caracteres.
5. *assigned* : Por si después de todo queremos que los identificadores los gestione la propia aplicación.

<discriminator>Cuando una clase declara un discriminador es necesaria una columna en la tabla que contendrá el valor de la marca del discriminador. Los conjuntos de valores que puede tomar este campo son definidos en la cada una de las clases o sub-clases a través de la propiedad <discriminator-value>. Los tipos permitidos son string, character, integer, byte, short, boolean, yes_no, true_false.
1. *column*: El nombre de la columna del discriminador en la tabla.
2. *type*: El tipo del discriminador.

<property> Declara una propiedad persistente de la clase , que se corresponde con una columna.
1. *name*: Nombre lógico de la propiedad.
2. *column*: Columna en la tabla.

3. *type*: Indica el tipo de los datos almacenados. Mirar **Tipos de datos en Hibernate** para ver todos las posibilidades de tipos existentes en Hibernate.

Tipos de relaciones.(Componentes y Colecciones) En todo diseño relacional los objetos se referencian unos a otros a través de relaciones, las típicas son : uno a uno **1-1**, uno a muchos **1_n**, muchos a muchos **n-m**, muchos a uno **n-1**. De los cuatro tipos, dos de ellas **n-m** y **1-n** son colecciones de objetos las cuales tendrán su propio y extenso apartado, mientras que a las relaciones **1-1** y **n-1** son en realidad componentes de un objeto persistente cuyo tipo es otro objeto persistente.
<many-to-one>La relación **n-1** necesita en la tabla un identificador de referencia, el ejemplo clásico es la relación entre padre - hijos. Un hijo necesita un identificador en su tabla para indicar cual es su padre. Pero en objetos en realidad no es un identificador si no el propio objeto padre, por lo tanto el componente n-1 es en realidad el propio objeto padre y no simplemente su identificador. (Aunque en la tabla se guarde el identificador)
1. *name* : El nombre de la propiedad.
2. *column* : Columna de la tabla donde se guardara el identificador del objeto asociado.
3. *class*: Nombre de la clase asociada. Hay que escribir todo el package.
4. *cascade* ("all|none|save-update|delete"): Especifica que operaciones se realizaran en cascada desde el objeto padre.

<one-to-one>Asociacion entre dos clases persistentes, en la cual no es necesaria otra columna extra. Los OID's de las dos clases serán idénticos.
1. *name* : El nombre de la propiedad.
2. *class* : La clase persistente del objeto asociado
3. *cascade* ("all|none|save-update|delete") : Operaciones en cascada a partir de la asociación.
4. *constrained* ("true"|"false") .

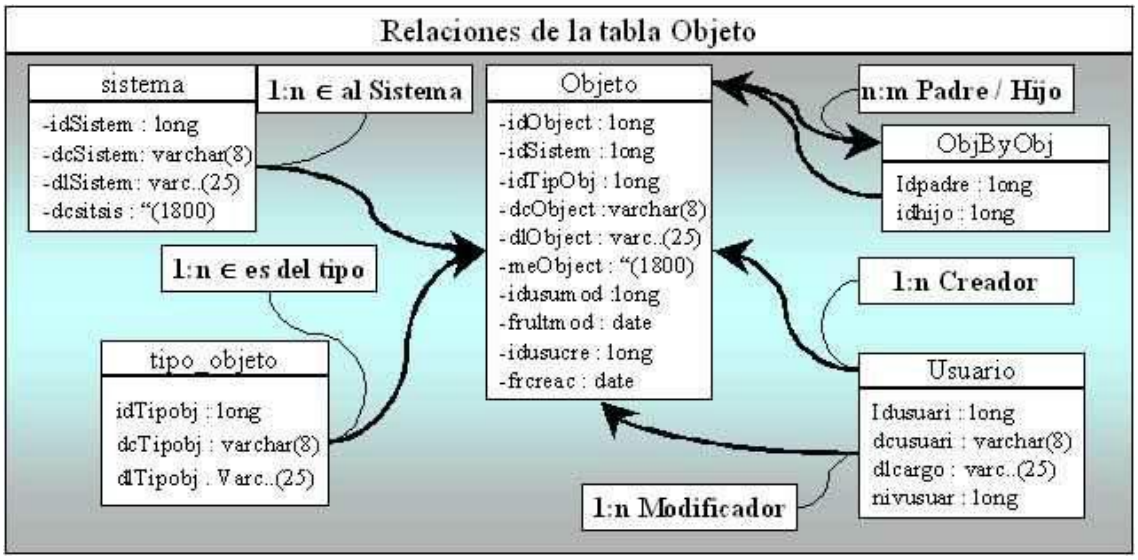
Tipos de datos en Hibernate.
» *Son tipos básicos*: integer, long, short, float, double, character, byte, boolean, yes_no, true_false.
» *string* : Mapea un java.lang.String a un VARCHAR en la base de datos.
» *date, time, timestamp* : Tipos que mapean un java.util.Date y sus subclases a los tipo SQL : DATE, TIME, TIMESTAMP.
» *calendar, calendar_date* : Desde java.util.Calendar mapea los tipos SQL TIMESTAMP y DATE
» *big_decimal* : Tipo para NUMERIC desde java.math.BigDecimal.
» *locale, timezone, currency* : Tipos desde las clases **java.util.Locale**, **java.util.TimeZone** y **java.util.Currency**. Se corresponden con un VARCHAR.
Las instancias de Locale y Currency son mapeadas a sus respectivos códigos ISO y las de TimeZone a su ID.
» *class* : Guarda en un VARCHAR el nombre completo de la clase referenciada.
» *binary*: Mapea un array de bytes al apropiado tipo de SQL.
» *serializable* : Desde una clase que implementa el interface Serializable al tipo binario SQL.
» *clob, blob*: Mapean clases del tipo java.sql.Clob y java.sql.Blob
» *Tipos enumerados persistentes (Persistente Enum Types)*: Un tipo enumerado es una clase de java que contiene la enumeración a utilizar(ej:Meses, Dias de la semana, etc..). Estas clases han de implementar el interface **net.sf.hibernate.PersistentEnum** y definir las operaciones **toInt()** y **fromInt()**. El tipo enumerado persistente es simplemente el nombre de la clase completo. Ejemplo de clase persistente :

```
package com.hecsua.enumerations;
import net.sf.hibernate.PersistentEnum
public class Meses implements PersistentEnum {
private final int code;
```

```
        private Meses(int code) {
                this..code = code;
}
public static final Meses Enero = new Meses(0);
public static final Meses Febrero = new Meses(1);
public static final Meses Marzo = new Meses(2);
...
public int to Int() {return code;}
public static Meses fromInt(int code){
case 0: return Enero;
case 1: return Febrero;
case 2: return Marzo;
...
default: throw new RuntimeException("Mes no valido.");
                }
}
```

Creación de Ficheros XML

El siguiente paso es aplicar todo lo anteriormente explicado en un ejemplo. Partiendo del siguiente diseño, crearemos los ficheros XML de acuerdo a las especificaciones anteriormente explicadas:



Traducir este diseño relacional a ficheros XML siguiendo las especificaciones Hibernate será mucho más sencillo de lo que pueda parecer a primera vista
Para crear el mapeo de **Objeto**[3], comenzaremos con la definición del fichero XML[4], mediante las dos líneas siguientes :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
  DTD//EN"
    "..\..\hibernate\hibernate-mapping-2.0.dtd">
```

En estas dos líneas declaramos el tipo de codificación del fichero XML así como su DTD.

Después comenzamos a definir nuestro objeto persistente, abrimos el tag

<hibernate-mapping>. Una muy buena costumbre es definir un objeto por fichero, ya que hace mucho más legible el código y simplifica su modificación.
El siguiente tag será el **class**:

```
<!- Clase persistente que representa un Objeto dentro del proyecto este puede ser una pantalla, un procedimiento un algoritmo, dependerá del proyecto en cuestión-->
    <class name="com.hecsua.bean.Objeto" table="objeto">
```

Como podéis observar la clase se denominará **Objeto** y estará dentro del package **com.hecsua.bean**. Esta clase hará referencia a la tabla objeto dentro de la BBDD.

El siguiente paso es definir el identificador, el nombre lógico de la propiedad queremos que sea **id**, que haga referencia a la columna **idObject** y es un dato de tipo **long**, el resultado es el siguiente :

```
    <id name="id" column="idobject" type="long">
```

Para obtener los OID's utilizaremos la clase **hilo** sobre la tabla **uid_table** y la columna **next_hi_value** :

```
    <generator class="hilo">
    <param name="table">uid_table</param>
    <param name="column">next_hi_value</param>
    </generator>
```

Las propiedades se definen de manera parecida al identificador, con el nombre lógico de la propiedad (aquí podremos ser tan descriptivos como deseemos), la columna dentro de la tabla a la que hace referencia y el tipo de dato. En el caso de que este necesite parámetros de longitud u otro tipo también se declararán aquí mediante el parámetro adecuado.

```
    <property name="descor" column="dcobject" type="string" length="8" />
    <property name="deslon" column="dlobject" type="string" length="25" />
    <property name="texto" column="meobject" type="string" length="1500"/>
    <property name="frCreacion" column="frcreac" type="date" />
    <property name="frUltimaModificacion" column="frultmod" type="date" />
```

Ahora definimos las relaciones, como se puede ver son todas del tipo muchos a uno, por lo que en realidad son columnas de identificadores ajenos a nuestra tabla. Se definen utilizando la etiqueta many-to-one, el nombre lógico de la propiedad nos ha de recordar el rol de la relación, el parámetro class deberá ser la clase del objeto asociado y la columna que almacenara su identificador.

```
  <!- Relación con la clase persistente Sistema a través de la columna idsistem -->
  <many-to-one name="sistema" class="com.hecsua.bean.Sistema" column="idsistem" />
  <!- Relación con la clase persistente Usuario realizando el rol de Usuario Creador del objeto
      a través de la columna idusucre -->
  <many-to-one name="creador" class="com.hecsua.bean.Usuario" column="idusucre" />
  <!- Relación con la clase persistente Usuario con el rol de Ultimo Usuario que Modifico el
      Objeto a través de la columna idusumod-->
   <many-to-one name="modificador" class="com.hecsua.bean.Usuario"
    column="idusumod" />
```

Por último tenemos que mapear la relación n-m, esta es una relación circular entre registros de la tabla objeto. Para mapear esta relación utilizaremos la siguiente estructura, dejando como se puede observar los identificadores relacionados en la tabla Obj_By_Obj:

```
    <set name="padres" table="obj_by_obj" lazy="true">
    <key column="idobject" />
    <many-to-many column="idobject" class="com.hecsua.bean.Objeto"
     not-null="true" />
```

```
</set>
<set name="hijos" table="obj_by_obj" lazy="true" inverse="true">
<key column="idobject" />
<many-to-many column="idobject" class="com.hecsua.bean.Objeto"
  not-null="true" />
</set>
```

En estas dos relaciones observamos que ambas utilizan la tabla obj_by_obj donde se guardarán las parejas de identificadores relacionados. Una de ellas ha de ser "inverse", con esta etiqueta declaramos cual es el final de la relación circular.

Finalmente cerramos las etiquetas, y acabamos de crear nuestro primer mapeo de un objeto relacional.