BASE DE DATOS - Clase 8, 9 y 10

NoSQL

Dada la evolucion de los tamaños y tipos de datos, se necesitaba que diseñar nuevos SGBD

Caracteristicas deseadas:

- Mayor escalabilidad
- mayor performance en las aplicaciones web
- mayor flexibilidad
- mayor capacidad de distribucion

Los SGBD relacionales tienen varias limitaciones (los joins son costosos y las transacciones distribuidas no escalan)

Las BDD NoSQL se clasifican en distintos tipos:

Tipos
clave-valor
orientadas a documentos
wide column
basadas en grafos

Las BDD no relacionales buscan tener mejoras por sobre las relacionales, de ahi surgen estas BDDs Entre esas mejoras esta la FLEXIBILIDAD, ESCALABILIDAD

> Agregado: Conjunto de objetos relacionados entre si que se agrupan en colecciones para ser tratados como una unidad y ser almacenados en un mismo lugar. Ejemplos: posts de Facebook con sus comentarios;;

EN BDD RELACIONALES Y EN BDD ORIENTADAS A GRAFOS NO ESTA EL CONCEPTO DE AGREGADO

BASES DE DATOS DISTRIBUIDAS

Para tener mayor velocidad de procesamiento y la capacidad de almacenar informacion, implementan las funcionalidades de un SGBD de datos distribuidos.

> Un SGBD de datos distribuidos es aquel que corre como sistema distribuido en distintas computadoras (nodos) que se encunetran sobre una red (red local, Internet, Cloud).

Esto aprovecha las ventajas de correr todo el sistema en un entorno de computacion distribuida y con la abstraccion de ser un unico sistema.

Las BDD NoSQL tienen estas funcionalidades entre otras:

Concepto	Explicacion
FRAGMENTACION	Distribuir agregados en un conjunto de nodos o computadoras.
	Se realiza con dos objetivos: mas almacenamiento y mas velocidad de procesamiento a traves de la paralelizacion.
	Se puede fragmentar horizontalmente (los agregados se guardan en distintos nodos) y verticalmente (se guardan un subconjunto de atributos en los nodos y todos comparten la clave)
REPLICACION	(tener un agregado en diferentes nodos por backup o disponibilidad).
	Brinda un mecanismo de backup frente a fallas o perdidas. Permite repartir la carga de procesamiento.
	Garantiza disponibilidad.
	Cuando funcionan solo como mecanismo de backup, se la denominan replicas secundarias. Cuando tambien procesan, son replicas primarias.
	Con las replicas aparece el concepto de CONSISTENCIA, ya que para distintas replicas pueden almacenar distintos valores para un mismo dato

BDD NoSQL KEY VALUE (DynamoDB)

Se componen de agregados del tipo key-value (Ej: {"nombre": "Kiko"})
Las claves son unicas y la unica condicion que tienen es que sean comparables por el =

Las BDD tienen cuatro operaciones elementales: put, delete, update, get

- Ventajas: Simplicidad, ya que no se define un esquema, ni DDL, etc..
  Velocidad ya que prioriza la eficiencia por sobre la integridad
  Escalabilidad dada la capacidad de replicas y repartir consultas entre nodos.

. Vemos el ejemplo de Dynamo, la key-value store de Amazon que utiliza **HASHING CONSISTENTE**
Esta diseñado sobre una arquitectura orientada a servicio: la BDD esta distribuida en un server cluster que posee servidores web, routers de agregacion y nodos de procesamiento (Dynamos instances)

HASHING CONSISTENTE es el mecanismo de lookup. A partir de la clave, dado que es unica, aplica una funcion de hash para saber a que nodo va a ir ese fragmento

Para saber en que nodo esta una determinada clave (LOOKUP) utiliza el metodo HASHING CONSISTENTE
Tiene la ventaja de mover la cantidad de pares que son necesarios unicamente en el caso de que un nodo se caiga o sea borrado.

El **HASHING CONSISTENTE** no solo hashea los datos, sino que tambien hashea los nodos.

Utiliza tambien el modelo de **CONSISTENCIA EVENTUAL**
Los nodos son peers, por lo tanto es totalmente descentralizado. El identificador de cada nodo

generalmente es su IP.

MODELOS DE CONSISTENCIA

Se relacionan con la REPLICACION de las bases de datos distribuidas No se puede usar sincronizacion de reloj en sistemas distribuidos porque siempre presentan un margen de error y las transacciones no los admiten

Para sincronizar se utiliza el algoritmos de ORDENAR EVENTOS

- De estos algoritmos provienen los modelos de CONSISTENCIA SECUENCIAL,
  Una BDD es de este tipo si el resultado de cualquier ejecucion concurrente de los procesos es equivalente al de alguna ejecucion secuencial de las mismas instrucciones.

En otras palabras, si encontramos un orden secuencial logico que represente lo que ve el usuario, decimos que tiene CONSISTENCIA SECUENCIAL

- El modelo de CONSISTENCIA CAUSAL busca capturar eventos que sean causados uno del otro
  Para ver un evento A, antes tienen que ver todos los eventos Bi que hayan influenciado a A

Dos escrituras causadas potencialmente deben ser vista por todos en el mismo orden que se realizaron

- La CONSISTENCIA EVENTUAL se presenta cuando no se escribe en un lapso de tiempo lo suficientemente grande, entonces garantiza que todas las BDD van a leer lo mismo Esto surge de la idea de que las mayorias de las operaciones son solo de lectura, entonces se puede soportar en mayor medida inconsistencias

Dynamo aseguta CONSISTENCIA EVENTUAL donde utiliza un MERKLE TREE que se compone de un hash criptografico donde se evalua una funcion H() que se compone de los Hi() de las ramas del nodo y que es comparada entre los distintos arboles de las distintas BDD. Si los Hi() de los arboles coinciden para el mismo nodo, entonces se asegura que la informacion es la misma.

BASES DE DATOS ORIENTADAS A DOCUMENTOS (MongoDB)

En estas BDD los agregados son los documentos. Los documentos son agregados que siguen una cierta estructura.

Generalmente se manejan con un par clave-valor a traves de un documento (XML, JSON, BSON, etc..)

Es decir, que no tiene la necesidad de definir un esquema rigido para la estructura del documento, sino que utiliza una estructura mas simple key-value.

Luego, se define al documento como un conjunto de pares clave-valor que representa los atributos del documento y sus valores.

Puede haber atributos multivaluados y documentos anidados.

> Para identificar objetos utiliza hashes. Cada documento tiene un hash que lo identifica.

NO utiliza esquemas o es schema-free. No permite juntas (JOIN)

BDD RELACIONAL	MONGODB
ESQUEMA O SCHEMA	BASE DE DATOS
TUPLA	DOCUMENTO
ATRIBUTO	CAMPO
RELACION	COLECCION

Las claves siempre son strings, pero los valores pueden ser distintos tipos de datos.

VER EL TALLER Y LA PRACTICA DE MONGO YA QUE SE VIERON EJEMPLOS

A todos los documentos que se crean, MongoDB le agrega un ID. Si yo no le especifico alguno, lo agrega automaticamente (un hash de 12 bytes). Con al funcion ObjectID(h) convierto una hash en una referencia al documento de dicho hash

> COMENTARIO: Dentro de MongoDB puedo definir variables y luego insertarlas.

EJ:

```
producto1: {
    "codigo" : "EA315",
    "nombre" : "pizza"
}
db.productos.insertOne(producto1)
```

> En este ejemplo, si tuviese otros productos con otro campo como ejemplo *PRECIO*, crearia el atributo en la coleccion y dejaria en NULL el *precio* de aquellos documentos sin precio.

En MONGODB podemos ver a las FK como "documentos referenciados" que son documentos anidados.

Tiene la desventaja de que si queres modificarlo, tenes que ir a todos los documentos donde estan referenciados y cambiar uno por uno

La idea es usarlo para evitar hacer juntas en Mongo. NO HACER JUNTAS CON LOOKUP

Para referenciarlo, utilizo el '.'. EJEMPLO: "cliente.nombre" Si debemos acceder constantemente al documento referenciado, pensar si no conviene que esté embebido.

CONSULTAS BASICAS

Las consultas se hacen con un **find**:

```
db.productos.find({"localidad" : "Moron"})
```

Esto devuelve un cursor que debe ser iterado.

Puedo tener 2 tipos de **referencias** a otros documentos:

- Con el hash (_id) del documento en cuestion (referencia)
- Insertando el documento dentro de otro docuemnto (embebido o anidado)

Ej:

```
db.productos.find({"productos.producto" : id_producto3})
```

En MongoDB se puede sacrificar el concepto de no redundancia y normalizacion. Se hace esto para ganar velocidad de procesamiento. Las funciones de agregacion se dividen en etapas a traves de un pipeline, donde la salida de cada etapa puede ser usada como entrada de la siguiente en el mismo pipeline La busqueda de documentos es muy rapida ya que se basa en hashing

- Agregacion en MongoDB: Es implementada a traves de un pipeline que combina etapas de agrupamiento con selecciones Es operada a traves de un JSON con la secuencia de operaciones Operaciones:
  - match: Filtra los resultados.
  - group: Agrupamiento por uno o mas atributos.
  - sort: Ordenamiento de resultados.
  - limit: Limitado de resultados
  - sample: Seleccion aleatoria de los resultados.
  - unwind: Es como desanidar
    Cada etapa devuelve una coleccion. Por lo tanto, la salida de una etapa es la entrada de la siguiente.

> **SHARDING** es el mecanismo que usa MongoDB para distribuir el procesamiento.
> Consiste en distribuir horizontalmente las colecciones en CHUNKS que se desparraman en cada SHARDS

Un SHARDING CLUSTER de MongoDB esta formado por: . shards(fragmentos): Son los nodos en los que estan los datos realmente, es decir, los chunks. Cada uno de estos corre un proceso llamado mongod . routers: Son los nodos servidores que reciben las consultas y las resuelven. Corre un proceso llamado mongos . servidores de configuracion: Almacenan la configuracion de los shards y los routers

El particionado se realiza a partir de una shard key, que debe ser un atributo o conjunto de atributos INMUTABLE de la coleccion.

Se puede realizar un hibrido y tener colecciones shardeadas y no shardeadas Las no shardeadas se almacenaran en un shard particular (shard primario)

Los SHARDING tienen restricciones: La shard key debe estar definida en todas las colecciones La coleccion debe tener un indice que comience con la shard key. Si no lo tiene, MongoDB se lo creara No se puede cambiar la shard key una vez hecho el sharding Tampoco se puede unshardear una coleccion shardeada

El SHARDING permite disminuir el tiempo de respuesta al distribuir el procesamiento y de realizar consultas sobre un conjunto de datos muy grandes que no cabrian en un solo servidor. Para esto es fundamental el indice de la shard key. La idea es que la BDD sea escalable para Big Data

MongoDB birnda un mecanismo de tolerancia a fallas basado en replicacion de shards Consiste en nodos primarios y secundarios. Los secundarios ademas de ser para backup sirven para realizar consultas pero solo de lectura. Las escrituras (y lecturas tambien) solo se realizan en el primario.

El esquema de replicas de MongoDB es el **master-slave with automated failover**. Consiste en en que cada shard contiene un servidor mongod primario y uno o mas servidores mongod secundarios (mongod es mi servidor que levanta la DB). Las replicas eligen entonces un master a traves de un algoritmo. En caso de que el master falle, los slaves eligen un nuevo master entre si.

BASES DE DATOS WIDE COLUMN (Cassandra)

Agrega todos los datos que pertenecen a una clave primaria en columnas
No es estrictamente orientada a columnas.

Veremos el caso de Cassandra, que es un hibrido de wide-column y clave-valor No es libre de esquemas como MongoDB. Posee una arquitectura share-nothing donde no existe un estado compartido centralizado Esto quiere decir que todos los nodos son iguales y esto hace que sea altamente escalable

RELACIONALES	Cassandra
esquema	Keyspace
Tabla	Column family
Fila	Fila

Una **fila** esta compuesta de una clave compuesta y un conjunto de claves-valor o columnas Es decir, una columna no es mas que un par de clave-valor asociado a una clave. Asi, cada vez que hay una nueva insercion, simplemente se busca la clave y se le agrega una columna con el nuevo dato a esa fila. De aqui que tambien se las considera **wide row**.

Usa el lenguaje CQL que no se parece en potencia a SQL, solo se limita al SELECT FROM WHERE

Las claves primarias estaran divididas en dos partes: la clave de particion y la clave de clustering Cada clave de particion identifica un agregado (un wide row) y cada clave de clustering identifica una fila.
Por ejemplo, en el caso de una libreria con clientes, podriamos diseñar la BDD de tal forma que la clave de particion sea algun tipo de dato que identifica al cliente y luego en cada fila estarian los datos de cada libro que compro, donde el ISBN seria la clave de clustering.

La clave de particion es la clave primaria, por lo que debe ser la misma para todas las filas (Padron, DNI, ISBN, etc..) No es obligatorio tener una clave de clustering

Recordar que en el WHERE solo puede haber atributos que pertenezcan a la clave primaria

> Las claves no deben obligatoriamente ser minimales

Al momento de hacer las consultas, se utiliza la clave de particion que es la que se compara por el operador igual (=)

Tambien implementa el metodo de HASHING CONSISTENTE para el almacenamiento de los datos/nodos en cada cluster.

El diseño de Cassandra impone:

- Las columnas de las partition key deben ser comparadas por = contra valores constantes.
- Si una columna de la clustering key es usada en una condicion del predicado, las restantes columnas de la key que la preceden tambien deben ser usadas en algun otro predicado.
-

PUNTOS PARA DISEÑAR UNA BASE DE DATOS

- En Cassandra tampoco existe el concepto de junta, asi que cualquier resultado parecido a este debe ser guardado desde el principio como una tabla mas
- No existe tampoco la integridad referencial. La aplicacion se debe encargar de esto
- Esta presente la desnormalizacion de datos para tener mas performance
- Diseño orientado a consultas, es decir, el modelo estara basado en que tipo de consultas realizaremos

Diagrama Chebotko
Busca que cada consulta se resuelva accediendo a una unica column family, los resultados esten en una unica particion y se respete el lenguaje CQL Ejemplo de esto en el video https://youtu.be/lGbHIbn7giU?t=1490

Metodo de acceso:
Cassandra esta optimizado para altas tasas de escrituras, donde utiliza un arbol llamado LSM-tree. Este tipo de arbol busca ofrecer un acceso secuencial a los datos, dado que el acceso aleatorio es muy costoso. Alguna de sus mejoras son :

- Mantener mas de un nivel en disco
- Mantener varios archivos por cada nivel y mergearlos cuando superan un umbral.
- Utilizar Bloom filters para verificar si una entrada podria estar en uno de los archivos.

NoSQL orientado a grafos (Neo4j)

BASES DE DATOS ORIENTADA A GRAFOS (Neo4j) Aca la estructura es la de un grafo, es decir, que hay nodos y arcos.
Las BDD orientadas a grafos sirven para modelar interrelaciones mas complejas entre las entidades. Aca no nos preocupamos tanto por los datos, sino por las interrelaciones.
No tienen el concepto de agregado a diferencia de los demas NoSQL.

Es muy buena esta tecnologia para casos en donde hay muchas interrelaciones unarias (ej: persona con persona modela HijoDe).
En estas BDD existen una LISTA DE ADYACENCIAS que contiene todos los nodos adyacentes de cada nodo.
Cada nodo tiene su lista con los nodos vecinos.
Los arcos siempres son direccionales en Neo4j, aunque no es necesario indicarle siempre la direccion.

Utiliza un lenguaje declarativo llamado Cypher

La BDD esta formada por nodos, donde cada nodo tiene uno o varios labels o etiquetas (ej: Persona)

```
(tom:Persona {nombre: 'Tomas', color: 'Azul', prof: 'Estudiante'})
```

Dentro de cada label, el nodo tiene propiedades con valores.

No sigue ninguna estructura por lo que a mis nodos pueden tener las propiedades que quieran.

Ejemplo de consulta en Cypher:

```
MATCH (p:Persona {nombre: ‘María’}) RETURN p.nombre, p.prof
```

"p:persona" le indica primero un alias 'p' que no se guarda, sino que es utilizado solo en la consulta Luego le dice que busque todas las cosas que sean "Persona" cuyo "nombre" sea "Maria" y las trabaje como 'p' Es analogo a poner un WHERE p.nombre = Maria

Los arcos o interrelaciones tienen que tener nombre y opcionalmente puede tener un atributo. Tienen que ser direccionales
Ej:

```
CREATE (juan)-[:AMIGO_DE]->(lucas)
```

Tambien es necesario que tenga una flecha para algun lado, es decir, tiene que ser dirigido
Este tipo de notacion marcando el vinculo sirve tambien para otras operaciones

ESTE ES EL ESQUEMA BASICO DE NEO4J

> MATCH
> WHERE
> RETURN

EJ:

```
MATCH (n:Persona)-[:AMIGO_DE]-(m:Persona),
          (m:Persona)-[:AMIGO_DE]-(o:Persona),
          (n:Persona)-[:ENEMIGO_DE]-(o:Persona)
    RETURN n.nombre, o.nombre
```

Con un * puedo indicar la cantidad de saltos entre dos nodos
Ej:

```
 MATCH (a:Actor)-[*4]-(kb:Actor {name:"Kevin Bacon"}) RETURN a.name
```

Esto devuelve los actores que estan a distancia 4 de Kevin Bacon

En definitiva, Neo4j trabaja con patrones en sus declaraciones
Un patron puede especificarse a traves de los nodos y sus propiedades, una interrelacion y sus propiedades, o un camino y sus propiedades
A cada patron se le puede asignar un nombre (el alias)

Las operaciones de agregacion siempre se realizan en el RETURN (count(*), sum(), max())

Neo4j establece que no puede haber un nodo repetido mas de una vez en un patron.

> Ej: Buscar los ancestros de Victor

```
MATCH (victor:Persona {nombre:‘Victor ’}),
(p:Persona)<–[:HIJO_DE *]–(victor:Persona)
RETURN DISTINCT p.nombre
```

((((((((((((PRACTICA))))))))))))))))))))))

call db.schema() me da una vision general del grafo. Me da los nodos y aristas

El PATRON DE NODOS es la semantica para indicar los nodos. La unica condicion obligatoria es que este entre parentesis En una REGEX, el '.' indica 'cualquier caracter' y el '*' indica que puede aparecer N veces

Ejercicios de la diapositiva ((((((((((TALLER)))))))))))))))) 1- MATCH (m {genre:"Science Fiction"}) RETURN m.title ORDER BY m.title LIMIT 10

2- MATCH (m:Movie) WHERE m.genre="Drama" AND m.studio =~".*Pictures*." RETURN m.title, m.studio

3- MATCH(a:Actor) WHERE a:Director AND a.birthday IS NOT NULL RETURN date(dateTime({epochmillis:toInt(a.birthday)})) LIMIT 20

Otra opcion de la 3 MATCH (a:Actor:Director) WHERE a.birthday IS NOT NULL RETURN ....

4- MATCH (d:Director)-[dir:DIRECTED]->(m:Movie) WHERE d.name = "Oliver Stone" return d,m

5- MATCH (a:Actor)-[ac:ACTS_IN]->(m:Movie)<-[ac2:ACTS_IN]-(th:Actor {name:"Tom Hanks"}) RETURN DISTINCT a.name ORDER BY a.name

6- MATCH (a:Actor)-[*4]-(kb:Actor {name:"Kevin Bacon"}) RETURN a.name LIMIT 10

7- MATCH (wa:Director {name:"Woody Allen"})-[d:DIRECTED]->(m:Movie) WHERE NOT (wa)-[:ACTS_IN]->(m) RETURN wa,m

8- MATCH sp=shortestPath((kb: Actor {name:"Kevin Bacon"})-[*]-(mr: Actor {name:"Meg Ryan"})) RETURN sp

9- MATCH (a:Actor)-[:ACTS_IN]-(m:Movie{title:"The Matrix"}), (a)-[:ACTS_IN]-(m2:Movie{title:"The Matrix Reloaded"}), (a)-[:ACTS_IN]-(m3:Movie{title:"The Matrix Revolutions"}) RETURN a.name

10- MATCH (d:Director)-[:DIRECTED]-(m)-[:ACTS_IN]-(a:Actor) RETURN d.name, COUNT(DISTINCT a.name) ORDER BY COUNT(DISTINCT a.name) DESC LIMIT 10

11- WHERE a.birthday IS NOT NULL WITH MAX(date(dateTime({epochmillis:toInt(a.birthday)}))) as max MATCH (a2:Actor) WHERE a2.birthday IS NOT NULL AND date(dateTime({epochmillis:toInt(a2.birthday)})) = max RETURN a2.name, date(dateTime({epochmillis:toInt(a2.birthday)}))

12- MATCH (a:Actor)-[ac:ACTS_IN]->(:Movie), (a)-[d:DIRECTED]->(:Movie) WITH a.name AS nombre, COUNT(DISTINCT ac) AS cant_peliculas,COUNT(DISTINCT d) AS dirigidas WHERE cant_peliculas>10 AND dirigidas>5 RETURN nombre

13- MATCH (a:Actor) WHERE a.birthday IS NOT NULL WITH MIN(date(dateTime({epochmillis:toInt(a.birthday)}))) as min MATCH (a2:Actor) WHERE a2.birthday IS NOT NULL AND date(dateTime({epochmillis:toInt(a2.birthday)})) = min WITH a2 as actor_viejo MATCH s=shortestPath((kb:Actor{name:"Kevin Bacon"})-[*]-(actor_viejo)) RETURN length(s)