# Introducción

Cómo surge BigData y NoSQL

## Bases de datos relacionales

Si bien hubieron avances en la tecnología, las DBR se mantuvieron constantes.

Se enfocan en la ejecución de Transacciones, y proveen:

* Atomicidad
* Consistencia: antes y después de la transacción
* Isolation: dos transacciones se ejecutan independientemente
* Durabilidad: el dato dura constantemente

## Bases de datos OO

Surgen a mediados de los 90 por necesidad de traducir objetos a relaciones. Estas bases de datos no tuvieron mucho éxito ya que no es fácil persistir modelos en OODB.

Ventajas sobre DBR:

Acceso más rápido a datos (no necesita joins)

•Mejor control de concurrencia (Locks por jerarquía de objetos)

•Modelo de datos similar a la representación real.

•Mejor en arquitecturas distribuidas.

•Requieren menos codificación.

Desventajas:

•Modelos de datos más complejos.

•No se lograron desarrollar herramientas y estándares como cuentan los RDBMS.

## Surgimiento BigData

En los últimos 10 años ocurre una revolución

* Mayor cantidades de datos provenientes por ejemplo de apps mobile
* Más datos no estructurados que crecen de forma impredecible

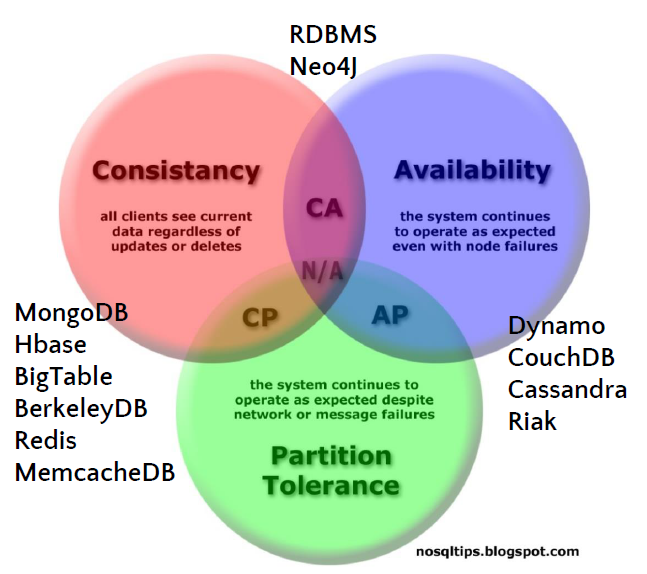
Surge BigData, con datos con:

* Volumen
* Velocidad
* Variedad
* Veracidad dudosa

BigData tiene la necesidad de escalar muy rápidamente, estos datos sería difíciles de almacenar y procesar en una DBR.

## Surgimiento NoSQL

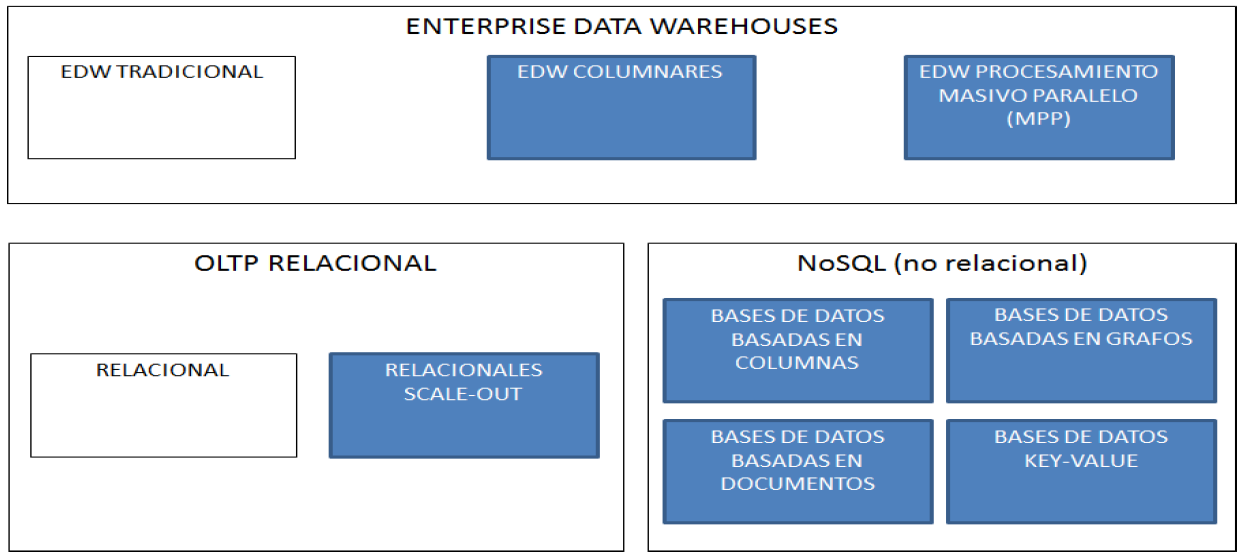
* Generación de Datos más acelerada y en gran tamaño.
* Distribución Horizontal De las BD.
* Tiempos de Respuesta< a 1 segundo.
* Aplicaciones Web/Mobile 7x24.
* Alta Disponibilidad de Servidores en todo momento
* Aplicaciones que generan datos mucho más rápido (Sensores, Logs, …).
* Sacrificar el ACID por otras propiedades Latencia, Particionamiento y Disponibilidad. La consistencia pasa a ser relativa. A veces no es tan importante. Un like más un like menos.
* En 1990, la memoria estaba a u$s45,000 por gigabyte, comparada con los u$s5 por gigabyte en 2012: Ayudó mucho a big data, es más barato almacenar (de muchas maneras diferentes) que procesar (almacenados de una sola forma y procesados para diferentes fines) datos.



\*PartitionTolerance: que sea tolerante a particiones de red. Por ejemplo si un servidor se desconectara de la red, el resto debería ser capaz de seguir funcionando y sincronizarse cuando se reconectaran.

Se fue dando el crecimiento de NoSQL:

* A partir de mediados del 2K
* Registrar actividad de usuarios para análisis de negocio orientado.
* IDC 2012 –Crecimiento del 40-50% de datos digitales no estructurados.
* Los RDBMS crecen a una tasa mucho menor.
* Soluciones escalables y distribuidas horizontalmente.
* Grandes Volúmenes de Datos y Procesamiento en paralelo.
* Incompatibilidad y Alto costo de los RDBMS y los Clusters.



## Conceptos NoSQL

* **Escalamiento horizontal y vertical:** Horizontal es agregar servers, Vertical es comprar equipo más potente, esto tiene un tope. El escalamiento horizontal permite:
  + **Alta disponibilidad y tolerancia a fallos:** cuando un server cae, se resincroniza automáticamente. Además los servers están distribuidos geográficamente, por lo que puedo encontrar los datos en el server más cercano a mi ubicación.
  + **Particionamiento de datos:** Los datos se encuentran distribuidos en muchos servers, ya que probablemente no entran todos en uno. Además pueden estar replicados para dar mayor disponibilidad geográfica.
* **Persistencia políglota:** Diferentes tecnologías de bases de datos para resolver diferentes problemas desde una misma aplicación. La app graba los datos en la DB que mejor se desempeñe.



* **Tipos DBNoSQL:**
  + **Clave-valor.** Buenas para cache
  + **Column-Family:** Buenas para almacenar datos de sensores
  + **Documentales**
  + **Basadas en grafos:** buenas para representar relaciones

# Modelado en NoSQL

El modelado es la forma particular de almacenar datos en algún medio físico. Determina la estructura lógica de una DB.

## Modelado SQL

Creado por los años 70 por Codd. Modela los datos utilizando relaciones en Tablas: conjuntos de relaciones. Los datos se persisten en las columnas

Relaciones

•Encabezado

•Atributos

•Cuerpo

•Tuplas

Claves

•Primarias (relaciones entre diferentes tablas)

•Foráneas

Integridad

•De Entidad

•Referencial

•Semántica

## Modelado NoSQL

Las bases de datos de

* tipo clave-valor (key-value)
* tipo familia de columnas (column-family)
* y las basadas en documentos (document based) comparten características comunes.

Las bases de datos basadas en grafos (graph based) contienen propiedades distintivas y aplican conceptos diferentes,

En general en las bases de datos NoSQL no existe el concepto de JOIN (cruzar dos tablas entre sí. En NoSQL se hace diferente).

## Principios básicos del modelado NoSQL

* **Denormalización:** 
  + Copiar los mismos datos en múltiples documentos para simplificar u optimizar el procesamiento de las consultas.
  + Permite agrupar los datos que se necesitan para procesar una consulta en un único lugar físico.
  + Es utilizada en las bases de datos de tipo key-value, column-family y document based.
  + En NoSQL, es la forma correcta de armar el modelo.
* **Agregados:** 
  + Un agregado es un registro complejo que permite listas, subregistros y otras estructuras anidadas incluidas.
  + Las bases de datos key-value, column-family y document based usan esta estructura de registro complejo.
  + El término agregado viene del DomainDrivenDesign (DDD), donde se define a un agregado como una colección de objetos relacionados que se desea tratar como una unidad.
  + En las bases de datos NoSQL el agregado es la unidad de manipulación de los datos y de administración de su consistencia. Cuando grabo un dato, grabo un agregado. El agregado se graba por completo o no se graba.
  + Trabajar con agregados hace mucho más fácil que los motores NoSQL operen en clusters, dado que el agregado es la unidad básica de replicación y distribución.
  + Son esquemas débiles de almacenamiento de datos:
    - Minimizan las relaciones uno a muchos a través de entidades anidadas y consecuentemente la reducción de operaciones de join.
    - Enmascaran las diferencias técnicas entre las entidades de negocio y el modelado heterogéneo de las entidades usando una colección de documentos o una tabla. Ej: todos los datos de una factura de un cliente en un mismo documento.
  + Son atómicos.
* **JOINs:**
  + En general las Bases de datos NoSql no soportan los Joins.
  + Los joins se evitan utilizando de-normalización y agregados, esto es, mediante entidades embebidas.
  + En muchos casos los joins son inevitables y deben ser resueltos por la aplicación. Por ejemplo:
    - Las relaciones muchos-a-muchos por lo general son resueltas mediante links y requieren joins.
    - Los agregados son a menudo inaplicables cuando la entidad es frecuentemente modificada internamente.

## Ventajas y desventajas de los distintos modelados NoSQL

**Tipo Key-Value:**

A favor:

* Estructura más genérica.
* El modelo de datos pasa a ser dinámico. Para agregar un nuevo dato, el desarrollador sólo inserta una nueva fila a la estructura.
* Este modelo proporciona una alta **flexibilidad** al permitir agregar atributos en una etapa posterior a la del diseño es algo muy simple. **No tiene que haber NULLs** para datos faltantes.

En contra:

* La **integridad** del tipo de dato **no está asegurada**. (fecha de tipo date)
* La integridad de **otras restriccione**s tampoco son garantizadas por el motor de base de datos. (Ej. Año>1945).
* El valor es generalmente un **dato opaco**.
* **Rigidez de acces**o a los datos.

**Tipo Column-Family:**

* En este tipo de bases de datos la **unidad básica de almacenamiento es la columna**.
* Una columna consiste en un **par nombre-valor** en donde el nombre actúa también como una clave.
* Cada familia de columnas puede compararse a un contenedor de filas en una tabla de un RDBMS, en donde la **clave identifica la fila y la fila consta de múltiples columnas**. La diferencia es que varias **filas no necesitan tener las mismas columnas**.
* Cada registro tiene las columnas que quiere: sirve cuando la columna pasa a ser un dato. O cuando hay mucha variación de las columnas que tienen o no las rows. Como por ejemplo los sensores.

A Favor: Flexibilidad para crear columnas en diferentes filas.

En contra:La i**ntegridad** de tipo de dato y de otro tipo de restricciones **no está asegurada**.

**Tipo Document-Based:**

* Almacenan documentos en la parte value del almacenamiento key-value, especificado en JSON.
* Puedo **jerarquizar** la información a mi conveniencia.
* Se pueden almacenar documentos enteros en formato de objetos JSON.
* Se pueden anidar tantos pares key-value como se necesite.
* JSON también soporta arreglos y **entiende diferentes tipos de datos, como strings, números, valores booleanos y null**.
* El esquema de los datos puede diferir entre los documentos, pero estos documentos pueden pertenecer a la misma colección (similar a una tabla).
* En los documentos pueden existir los atributos nulos o vacíos. ***Sparse*: no ocupa lugar por un dato que no tiene.**
* Los documentos permiten que **un nuevo atributo sea creado sin la necesidad de definirlo previamente** o cambiar la estructura de los documentos preexistentes.

**Tipo Graph-Based:**

* Los nodos pueden tener varios atributos.
* La información va en las relaciones, no en los atributos.

# Key-Value

* Una clave-valor es una tabla hash simple, principalmente usada cuando **todos los accesos a las bases de datos se hacen por una clave primaria**.
* En el mundo NoSQL son las bases de datos más simples. La complejidad está en cómo modelar las keys.
* La **mayoría**, pero no todas, son **in memory**. Algunas pueden persistir a disco.
* para usar desde una aplicación.
* Las operaciones que se pueden realizar son:
  + **Poner (SET**) un valor asociado a una clave
  + **Obtener (GET**) un valor dada una clave
  + **Borrar (DEL**) una clave
* El **valor es un BLOB (Binary Large Object**).
* La base de datos almacena el BLOB sin saber lo que hay dentro (opaco), y le da esa responsabilidad a la aplicación.
* Dado que estas bases de datos usan acceso por clave primaria, generalmente tienen muy **buenos tiempos de respuesta y escalan fácilmente**.

## En qué caso usarlas

**Almacenar información de sesión**:

* Un buen elemento es el sessionId de la sesión web.
* Las aplicaciones que almacenan sessionId en disco o en un RDBMS pueden beneficiarse de usar una BD clave-valor.
* Todo en una sesión puede almacenarse con un solo PUT o recuperado con un solo GET.
* Toda la info de la sesión se almacena en un solo objeto.

**User profile - preferences**:

* En general cada usuario tiene un único userId, userName, etc.
* Todos los atributos asociados al userId tales como lenguaje, paleta de colores, zona horaria, etc, pueden recuperarse en un solo acceso.
* Todos estos atributos pueden almacenarse en un único objeto.

**Carros de compras**

* Los sitios de e-commerce tienen carros de compra asociados al usuario.
* Toda la información del usuario asociado al carro de compra puede almacenarse como un único objeto.

Son usadas principalmente como DBs en memoria, no tanto para bajarlas a disco. **No manejan información crítica**. No sería tan problemático tener que reiniciar la DB.

## En qué caso NO usarlas

Relaciones entre datos

* Transacciones multi-operación.
* Consultas por los valores dentro de los valores. (a menos que la BD provea comandos específicos para consultar por los values)
* Operaciones que abarcan diferentes conjuntos de datos.
* La **idea es consultar por una key y obtener toda la info que necesito**. Si tengo que hacer **muchas consultas pierde el sentido**.
* Al modificar un alumno para cambiarle un dato, tengo que modificar todo el alumno entero.

## Redis

* Base de datos **key-value avanzada.** “Obtener el email del usuario con mayor puntaje en el partido que comenzó el 23 de junio a las 11pm.”
* Es Open Source [Licencia BSD].
* Escrita en ANSI C.
* Funciona en la mayoría de los sistemas POSIX: Linux, BSD, OS X. No hay soporte oficial para Windows.
* Para lograr su **alta performance, usa datasets in memory**. Por eso no puedo tener toda mi información en esta DB.
* Su value no es tan opaco, puede **almacenar distintas estructuras de datos**.
* La i**nterpretación del dato se hace en la aplicación**.

### Keys en Redis

* Las claves son BINARY SAFE: **se puede usar cualquier secuencia binaria como key**.
* **Case-sensitive**
* Tamaño máximo de clave: 512 MB
* Recomendaciones:
  + No usar keys demasiado grandes. Recordar que es una BD in memory y que una key mayor implica búsquedas más complejas.
  + No usar keys demasiado chicas, porque pueden resultar muy poco legible.
  + Buscar balance! Tratar de implementar un esquema. Por ej, usar “:” o “-” para separar multi claves (key → “user:3100:purchases”)
* Ser consistentes
* Si una key contiene:
  + Una lista de elementos → Key en plural
  + Datos de un único elemento → Key en singular

### Redis

* Redis es “in-memory data-structure store”.
* Usado como base de datos (clave-valor), caché, message broker.
* Se la suele llamar “**Data structure server” porque en su value puede almacenar distintas estructuras de datos**.
* Values almacenan comportamiento además de datos.

### Tipos de datos

Comentarios sobre las estructuras compuestas:

● Al **agregar un elemento a una key inexistente**, **primero se crea un dataset vacio y luego se agrega el elemento**.

● De igual forma, al **eliminar el último elemento, se elimina la key.**

● Las operaciones sobre keys inexistentes serán manejadas de igual forma a si los datasets

estuvieran vacíos.

### Value como String

Es el tipo de datos por defecto y más simple para un value.

* Para agregar un nuevo par key-value, basta con ejecutar el comando SET key value
* Para obtenerlo, se puede utilizar el comando GET key
* Para eliminar un par key-value, usaremos DEL key
* Para concatenar un valor utilizaremos APPEND key value
* Para utilizar el value como un array de caracteres podemos utilizar SETRANGE key ini fin value y **GETRANGE key ini fin**. Se usa principalmente para cache. Por ejemplo

GETRANGE key 0 3 me devuelve los 4 primeros caracteres

GETRANGE key 5 -1 me devuelve los caracteres del quinto al ultimo

GETRANGE key 5 100 me devuelve los caracteres del quinto al ultimo

Opciones **SET**

**EX seconds** / PX miliseconds -- al valor asignado se le asocia un tiempo de expiración en segundos/milisegundos.

NX / XX -- La operación es exitosa únicamente si la clave no existe/existe previamente.

**SET key value EX seconds**

### Value como Counter

Existe la posibilidad de utilizar el string como un contador:

* Internamente, Redis almacena este valor **como Integer.**
* Los comandos I**NCR key / INCRBY key cantidad** permitirán incrementar por 1 o en una cantidad determinada al value asociado a una key en forma ATÓMICA.
  + **Si dos usuarios simultáneamente modifican su valor, el mismo será consistente.**
* También existe la opción de decrementar un contador con los comandos DECR key / DECRBY key cantidad.
* Si se intenta realizar alguna operación de contadores sobre un string

127.0.0.1:6379> incr likes

(integer) 1

127.0.0.1:6379> incrby likes 21

(integer) 22

127.0.0.1:30000> type likes

string

127.0.0.1:6379> set title hola

OK

127.0.0.1:6379> incr saludo

(error) ERR value is not an integer or out of range

### Value como Hash

* Los hashes permiten dentro de un mismo value, mapear una nueva key con un nuevo valor.
* Son **útiles para representar “OBJETOS”**.
* No hay un límite para la cantidad de elementos que se pueden guardar en un hash (salvo la memoria física!).
* Dentro de las operaciones que se pueden realizar se encuentran:
  + Agregar una nueva entrada para la key: **HSET key subkey value. HMSET key subkey1 value1 subkey2 value2**
  + Obtener todos los pares key-value guardados para una determinada key: **HGETALL key**
  + Obtener el value de una subkey determinada: **HGET key subkey**
  + Obtener el listado de subkeys para una key determinada: **HKEYS key**
  + El value del par subkey-value puede ser tratado como un Counter, y por ende, aplicar las operaciones correspondientes.

> hmset post:1000 title “Great Scott!!” likes 22

shares 1 public 0

OK

> hget post:1000 title

"Great Scott"

> hgetall post:1000

1) "title"

2) "Great Scott"

3) "likes"

4) "22"

5) "shares"

6) "1"

7) "public"

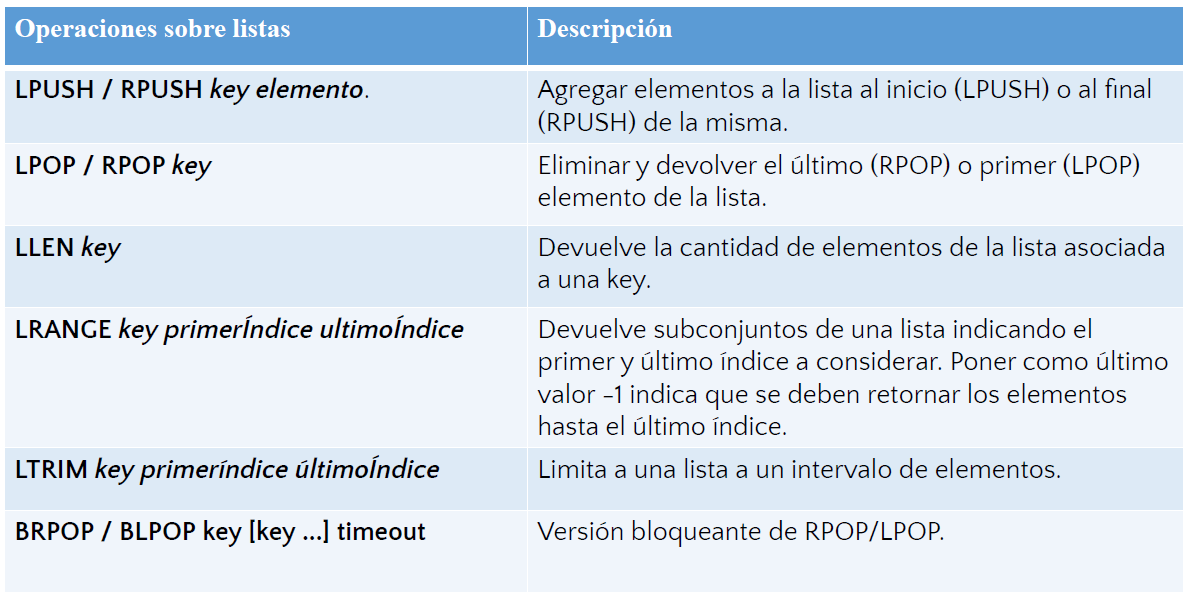
8) "0"

> **hincrby post:1000 likes 1**

OK

### Value como List

* Una lista es una serie de valores ordenados, que se almacenan como value para una key determinada.
* Implementadas como “Linked lists”:
  + Las operaciones de agregar elementos al principio o al final se realizan a tiempo constante, independientemente del número de elementos de la lista.
  + El acceso a elementos intermedios no es tan performante como en las listas implementadas con arrays.
  + Solamente **puedo eliminar primeros o últimos elementos, no del medio**.
* Ejemplo de cuándo usarlas:
  + Conocer las **últimas actualizaciones posteadas por un usuario** en una red social.
  + **Comunicación entre procesos usando el modelo del consumidor-productor**.



### Value como List OpsBloqueantes

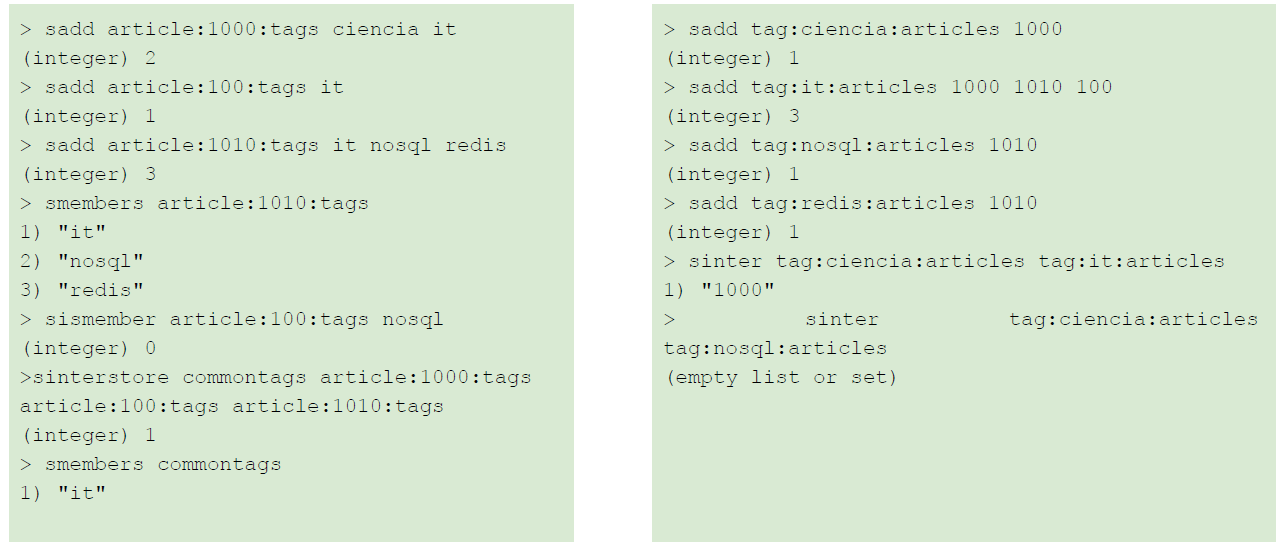
* BLPOP /BRPOP pueden ser utilizados para dar soporte a manejo de c**olas en un esquema Productor - Consumidor. BLPOP lista (o listas) tiempoTimeout**
  + Evita la necesidad de polling.
* El **timeout especifica el tiempo máximo que esperará el cliente para obtener un valor.** Luego de pasado dicho tiempo obtendrá un nil. Si este es 0 (cero), el cliente esperará indefinidamente.
* Si alguna de las listas indicadas posee elementos el cliente recibirá un elemento y no se bloqueará.
* Si varios clientes se bloquean ante una key, al agregar un elemento a su lista, se desbloqueará el primer cliente que se bloqueó ante la misma.
* Le puedo decir al consumidor “bloqueate hasta cierto tiempo” para que no se quede esperando para siempre.

### Value como Set

Es similar a una lista, pero se diferencia en que los elementos NO están ordenados y NO pueden repetirse.

* Algunas de las operaciones que se pueden hacer son:
  + Agregar al set un elemento: **SADD key value**
  + Eliminar del set un elemento **SREM key value**
  + Validar si un value está dentro del set: **SISMEMBER key value**
  + Conocer todos los elementos del set para una key determinada: **SMEMBERS key**
  + Obtener la cantidad de elementos de un set: **SCARD key**
  + Obtener un miembro cualquiera en forma random: **SRANDMEMBER key**
  + Entre conjuntos obtener:

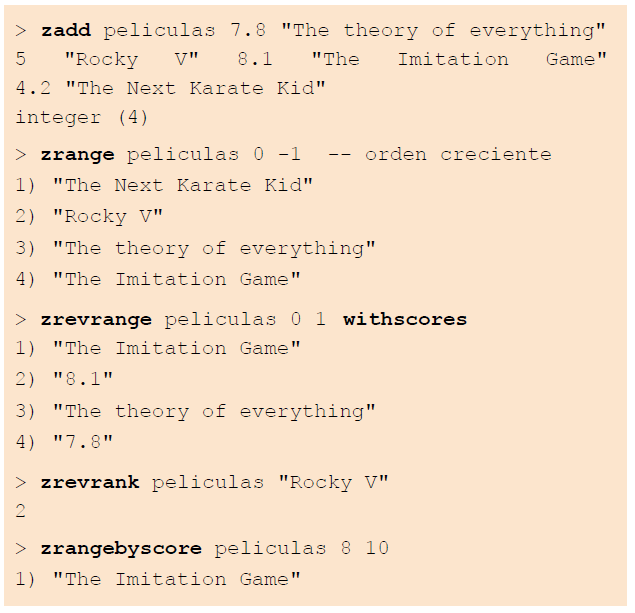
SINTER key [key ...] SUNION key [key ...] SDIFF key [key ...]



**sinter: graba la intersección en un nuevo SET.**

### Value como SortedSet

* Los sets son muy útiles pero, al ser desordenados, no funcionan bien para algunas situaciones.
* Por este motivo surgen los SORTED SETS: es similar a un set a secas, pero con la diferencia que c**ada value tiene un score (valor numérico) asociado**.
* Los elementos son ordenados según las siguientes reglas:
  + Si A y B son dos elementos con un score diferente, A > B si A.Score > B.Score
  + Si A y B tienen **igual score, se ordenan lexicográficamente**. Recordar que no pueden ser iguales porque los elementos NO se repiten!!
* Los valores se almacenan ordenados.
* Ejemplo de aplicación:
  + Juegos que guardan el ranking de usuarios y su puntaje.



### Value como BitMap

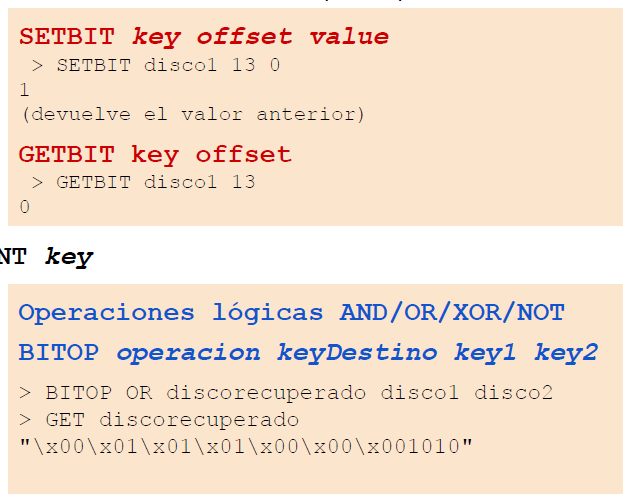
* No son realmente una estructura de datos, sino una nueva forma de interpretar a los Strings gracias a un conjunto de operaciones.
* Como los Strings son “Binary Safe” y su longitud puede ser de hasta 512 MB, los bitmaps se pueden aplicar en hasta 2(32) bits diferentes.
* Las operaciones se dividen en dos grupos:
  + Operaciones de un único bit. son de tiempo constante.

Por ejemplo, **setear un bit u obtenerlo**.

* + Operaciones de grupos de bits.

Por ejemplo, contar la cantidad de bits de un rango - **BITCOUNT key**

* Ventajas: Ahorro de espacio. Ej: tengo IDs de usuarios y quiero saber quiénes entraron en la plataforma en las últimas 24hs. **Puedo cambiar un bit rápidamente. Tengo que conocer la ubicación del bit.**
* Pueden ser divididos en múltiples claves.
* Ejemplos de uso: Real time analytics.
* Almacenamiento de información booleana



### Índices Geoespaciales

Los datos se **almacenan como un sorted set** de forma que luego se pueden realizar consultas por radio. Se usa **GEOADD, GEORADIUS, GEODIST, GEORADIUSBYMEMBER**.

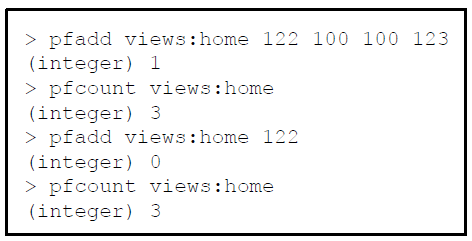
### Redis Value como HLL

* Es muy común el problema de querer contar la c**antidad de ocurrencias de “cosas” ÚNICAS.**
  + Suele consumir mucha memoria, ya que hay que recordar elementos pasados (cantidad proporcional al número de elementos a contar).
* HyperLogLog (HLL):
  + **Estructura de datos probabilística.**
  + Útil para determinar la **cardinalidad de un set.**
  + Tamaño fijo y pequeño de memoria (12KB como máximo).
* Uso similar a usar un Set, pero en HLL realmente **no se almacena elementos.**

**PFADD key val1 val2 val3**

**PFCOUNT key**

* HLL, permiten bajar los requerimientos de memoria a cambio de menor precisión en los resultados. En el caso de HLL el error estándar es < 1%.
* HLL es una estructura de datos que se codifica como un String → se pueden aplicar operaciones SET y GET.
* Ejemplo de uso
  + **Contar la cantidad de búsquedas distintas realizada por un usuario** por día. Eventos diferentes.
  + Contar la cantidad de u**suarios únicos que visualizaron una página**.

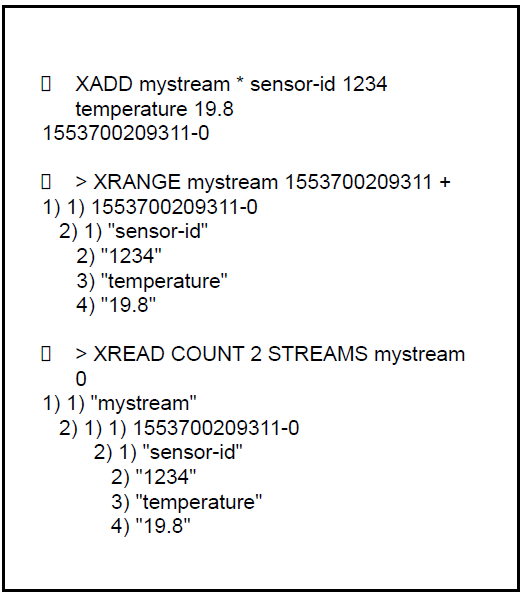


### Redis Streams

Los datos se almacenan en una **estructura append only** mediante la cual tendremos algunos clientes actuando de proveedor y otros como consumidor.

Para ello se utilizarán las siguientes operaciones:

* Para agregar pares clave valor dentro de un stream utilizaremos **XADD key \* field value**
* Para consumir un rango de valores se utilizara **XRANGE mystream - +**
* Además, si quisiéramos agregar una cantidad de valores a consumir podremos definir **XRANGE mystream - + COUNT nro**
* Para leer valores de un stream XREAD COUNT 2 STREAMS mystream 0
* Podremos también bloquear la operación de lectura con XREAD BLOCK 0 STREAMS mystream $



### Redis Operaciones sobre KeySpace

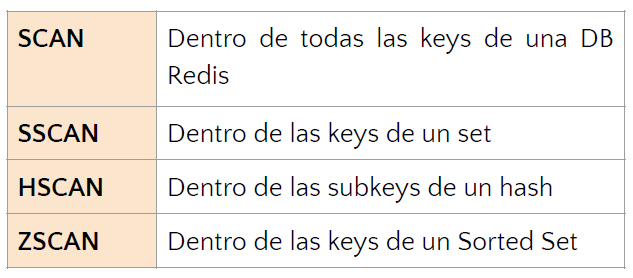
Todos los comandos mencionados en esta sección son INDEPENDIENTES DEL TIPO

DEL VALUE!!!

* Verificar si existe una clave con un valor: **EXISTS key**
  + Devuelve 1 si existe, 0 si no existe.
* Obtener el tipo de un value: **TYPE key**
* Determinar un TTL a una clave. Esto significa que cuando expira, es borrada la clave automáticamente “como si se hubiera realizado un DEL”.
  + Modificando el value con **SET key value PX/EX ttl**
  + Sin modificar el value EXPIRE key segundos
  + Se puede conocer el TTL de una clave con **TTL key**
  + Devuelve el TTL , -1 si no lo tiene seteado o -2 si la key no existe.

### Redis SCAN

Iterar incrementalmente sobre conjuntos de elementos.



Ofrecen garantía limitada de los elementos obtenidos.

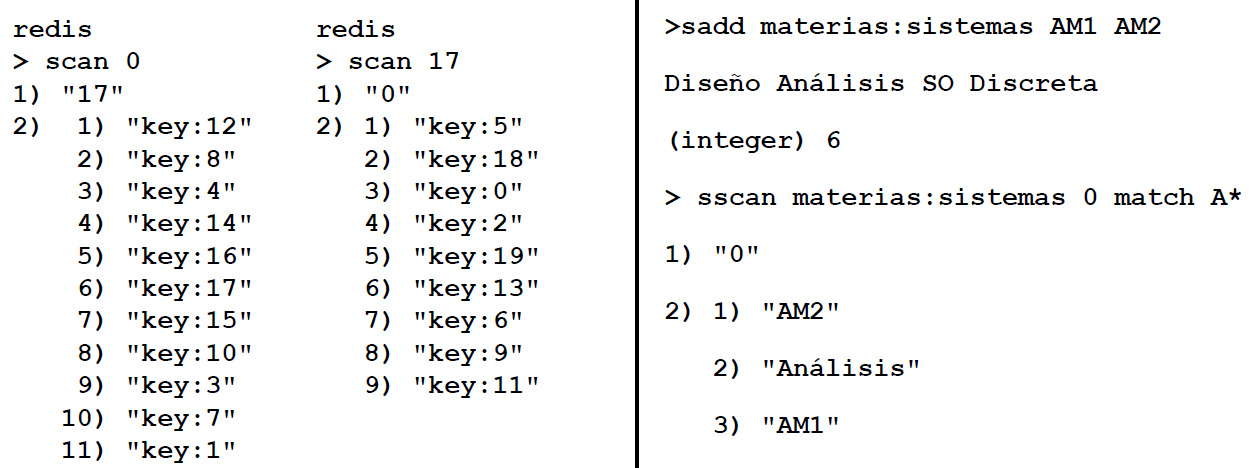
Sirve para **saber qué keys existen en la DB o subkeys dentro de un SET etc.**

Cuando hago un SCAN, Redis devuelve los datos de a bloques. Tengo que ir mandando el cursor para que me de el siguiente bloque de datos. Cuando **termina me devuelve 0.**

**SSCAN start** devuelve todos los elementos

**SSCAN key start**

**Match** permite **matchear por Key con expresiones regulares.**



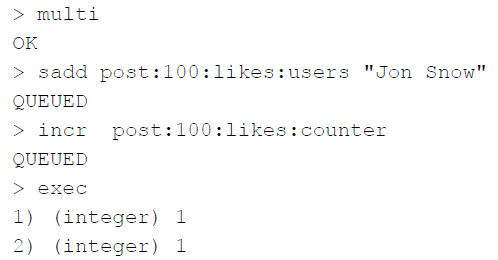
### Redis Blocks

* Estructuras de datos se **almacenadas en memori**a y se ejecutan en la base de datos, haciéndolas extremadamente rápidas.
* Pensar el modelado en base a los patrones de acceso o consultas a la DB.
* Las estructuras de datos pueden ser usadas como Legos, o building blocks, **ahorrando tiempo y esfuerzo a los desarrolladores.**
* Combinar estructuras de datos para conseguir nueva funcionalidades de forma simple: “Obtener el email del usuario con mayor puntaje en el partido que comenzó el 23 de junio a las 11pm.”

### Redis Transacciones

Redis permite ejecutar un conjunto de comandos como un bloque garantizando:

* **Aislamiento: No se procesan otras operaciones durante una transacción.**
* **Atomicidad: Se ejecutan todos los comandos o ninguno, si el desarrollador es responsable.**
* Si ocurriera un error programático en la ejecución y se hubieran realizado acciones antes, lástima. Re**dis no realiza rollbacks**.



**Check and Set =** comando **watch**

Se usa para "monitorear" keys y detectar cambios sobre las mismas.

* Si alguna de las keys con WATCH son modificadas por otro cliente antes de ejecutar EXEC toda la transacción aborta.
* Requiere el trabajo extra de reintentar la transacción si es que es abortada por esta causa.

### Scripts LUA

* Redis scripts es una transacción por definición: Todo lo que se hace con una transacción, se puede hacer con un script.
* Redis no necesita recompilar el script cada vez que es ejecutado.
* Para minimizar el ancho de banda consumido → se puede usar EVALSHA
* Hay que tener cuidado con que los scripts no sean muy lentos → bloquea al server

**redis.call(“lua”;param)**

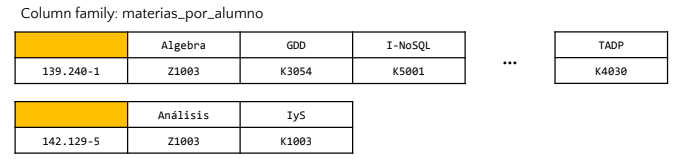
### Redis Modules

* Extienden la funcionalidad de Redis para atacar casos de usos comunes.
* En formato de Add-ons: **Librerías dinámicas que se cargan al iniciar el motor.**
* Pueden ser creados por cualquier persona.
* Utilizan el Redis Core mediante una API Alto y Bajo nivel, en C.
* Cada **módulo aporta nuevas comandos al set de comandos default de redis.**
* Compatibilidad binaria: Librerías no están acopladas a una versión específica del motor.

# Column Family

* Agrupación lógica de columnas
  + Pueden estar bajo el mismo prefijo o nombre
  + Con configuraciones idénticas
* También conocida **como wide row.**
* Agrupadas dentro de una row key o partición
* Es un caso especial de las key-value.
* Dos rows de una misma column family no tienen necesariamente las mismas columnas ni tampoco cantidad de columnas

Ejemplo

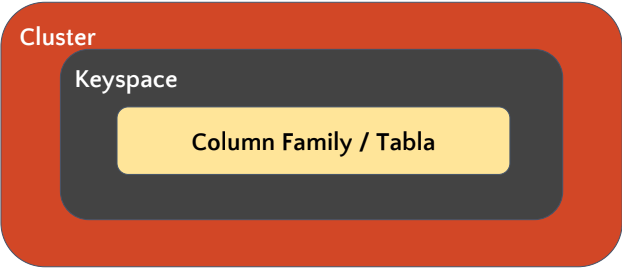


Por cada una de las materias que cursó el alumno con un legajo particular, voy creando una columna con la materia.

## Cassandra

* Es una BD Column family. Tenemos un esquema como en SQL, pero no tenemos que tirar un alter table (es más **desestructurado**).
* Es **Peer to Peer** (no existe maestro esclavo) todos los nodos son iguales y pueden tomar cualquier papel. Simplemente se crea un nodo sin ninguna tarea particular. Evita **SPOF**
* Masiva y **linealmente escalable**, en general. En teoría por cada nodo que agregue mi rendimiento va a crecer proporcionalmente hasta el infinito. Aunque en la práctica si le meto 100 nodos no va a ser totalmente lineal (de todas formas, la **tendencia** es bastante lineal)
* Es altamente disponible y ubicada en el **sector AP**. Sacrifica consistencia, cuando no quede otra alternativa.

### Cluster



Seleccionar un **keyspace** donde crear sus tablas/column families. Sobre este se definen ciertas propiedades a cómo va a tratar las tablas dentro de ese keyspace

### Anillo de Cassandra (didáctico)

* Todos los nodos comunican entre sí su estado interno constantemente (**gossip**)
* **Cada nodo** es responsable de almacenar **una parte de los datos**. Esto escala muy bien porque se tarda prácticamente lo mismo más allá de donde esté el dato.
* A cada **row-key se le aplica una función de hash** que **determina a qué nodo pertenece**. A esa **row key y todas sus columnas**, se las llama **partición**
* La **unidad básica** de replicación y particionamiento es la **partición**
* **NUNCA VAMOS A TENER UNA COLUMN FAMILY DISTRIBUIDA ENTRE MÁS DE UN NODO (tengo todo en un nodo o nada)**
* En Cassandra, a una column family se la llama **Table**. Pero es **absolutamente distinta a una tabla normal de SQL**
* **No existe join**, usa el lenguaje **CQL**. **No existe integridad referencial (no hay FK)**
* **No existen transacciones como las conocemos en el mundo relacional**

### Modelado

**Basarnos en las queries y no en las entidades.** El modelado de Cassandra solo tiene sentido frente al pedido de una query.

#### Sintaxis CQL (Cassandra Query Language)

**Creación de keyspace**

CREATE KEYSPACE nosql

WITH REPLICATION = {‘class’: ‘SimpleStrategy’, ‘replication\_factor’: ‘1’};

**Creación de tablas**

CREATE TABLE nombre\_tabla (

nombre\_columna tipo\_dato,

...

nombre\_columna tipo\_dato,

PRIMARY KEY (**(nombre\_columna,...,nombre\_columna)**,**nombre\_columna,...,nombre\_columna**));

##### Ejemplo 1

CREATE TABLE materias\_por\_alumno (

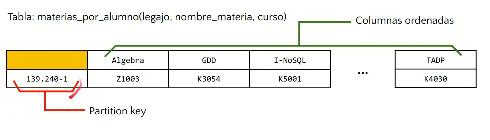
legajo TEXT,

nombre\_materia TEXT,

curso TEXT,

PRIMARY KEY (**(legajo)**, **nombre\_materia**)

);



**Columnas ordenadas = clustering columns → nombre\_materia**

**Partition key → legajo**

La **primary key** es el elemento más importante, porque ahí se define la partition key y clustering column. Si no las defino bien **hay muchas consultas que no voy a poder hacer**.

##### Ejemplo 2

CREATE TABLE materias\_por\_alumno (

legajo TEXT,

nombre\_materia TEXT,

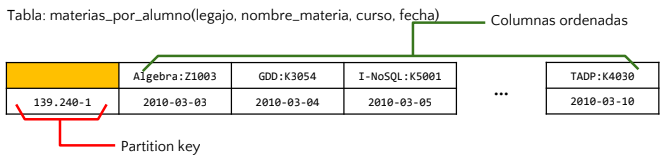
curso TEXT,

fecha TIMESTAMP,

PRIMARY KEY (**(legajo)**, **nombre\_materia, curso**)

);

Agregamos una nueva **clustering column curso**:



Notar que el único valor es la fecha.

##### Ejemplo 3

CREATE TABLE materias\_por\_alumno (

legajo TEXT,

nombre\_materia TEXT,

curso TEXT,

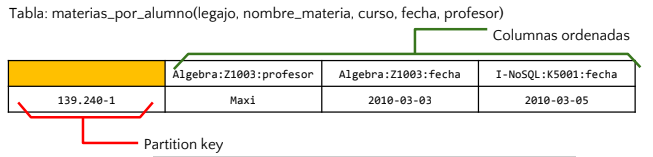
fecha TIMESTAMP,

profesor TEXT,

PRIMARY KEY (**(legajo)**, **nombre\_materia, curso**)

);

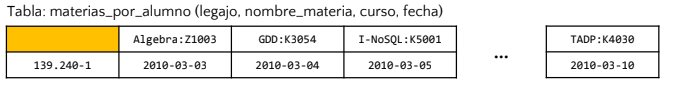
Agregamos un nuevo valor **profesor:**



Notar que si le agregamos el profesor además de la fecha… Cassandra lo agrega también a la columna!

#### CRUD

Operaciones básicas con Cassandra. Modelo de tabla del ejemplo:



##### Create

Es igual a la de SQL normal

INSERT INTO materias\_por\_alumno(legajo, nombre\_materia, curso, fecha)

VALUES(‘139.240-1’, ‘Algebra’, ‘Z1003’, ‘2010-03-03’);

INSERT INTO materias\_por\_alumno(legajo, nombre\_materia, curso, fecha)

VALUES(‘139.240-1’, ‘GDD’, ‘K001, ‘2010-03-04’);

INSERT INTO materias\_por\_alumno(legajo, nombre\_materia, curso, fecha)

VALUES(‘139.240-1’, ‘I-NoSQL’, ‘K5001’, ‘2010-03-10’);

##### Read

Opciones válidas:

SELECT \* FROM materias\_por\_alumno;

SELECT \* FROM materias\_por\_alumno WHERE legajo = ‘139.240-1’;

SELECT \* FROM materias\_por\_alumno

WHERE legajo = ‘139.240-1’ AND nombre\_materia = ‘Algebra’;

SELECT \* FROM materias\_por\_alumno

WHERE legajo = ‘139.240-1’ AND nombre\_materia = ‘Algebra’

AND curso > ‘A0000’;

Estas opciones son correctas porque Cassandra puede saber en que nodo del cluster se encuentra (al limitar correctamente en el where por la primary key). El primer caso también es válido porque trae todo.

Opciones inválidas:

SELECT \* FROM materias\_por\_alumno

WHERE legajo > ‘139.240-1’;

SELECT \* FROM materias\_por\_alumno

WHERE nombre\_materia > ‘Algebra’;

SELECT \* FROM materias\_por\_alumno

WHERE legajo = ‘139.240-1’

AND curso > ‘A000’;

SELECT \* FROM materias\_por\_alumno

WHERE fecha > ‘2010-03-05’;

* No tengo forma de comparar (operación <, >, !=) los hash… o sea si lo quisiera hacer tendría que ir a cada nodo con un full scan. Podría, pero por ser no performante lo bloquea directamente
* Lo mismo me pasa con Álgebra, que es una clustering column. Por lo tanto también lo bloquea. En este caso el orden me afecta, ya que es el primer atributo dentro de la clustering column.
* Si entro a una partition key específica, y quiero los cursos mayores a tales. No la va a ejecutar porque curso es el segundo atributo dentro de una clustering column. Tendría que escanear todas las clustering (es decir, todas las materias) de un alumno particular y por lo tanto no lo permite
* Las consultas en Cassandra están siempre condicionadas por la primary key. No puedo hacer nada sin eso, mucho menos analizar contra un value.

**Cassandra suele brillar sobre queries que ya andan lentas y no se pudieron optimizar, cosa de ya tener todo claro. Es decir, reemplazar algo que ya existe pero anda mal.**

##### Update

UPDATE materias\_por\_alumno

SET fecha = ‘2016-03-26’

WHERE legajo = ‘139.240-1’

AND nombre\_materia = ‘Algebra’

AND curso = ‘Z1003’;

Si no está lo inserto, si está lo actualizo (UPSERT) → Lo mismo en el insert

UPDATE → Localizar una celda exacta

¡Cuidado!

* Los update generan inserts si las condiciones no se cumplen (upsert)
* Las condiciones del update deben localizar una celda de manera exacta

##### Delete

DELETE fecha FROM materias\_por\_alumno

WHERE legajo = ‘139.240-1’

AND nombre\_materia = ‘Algebra’

AND curso = ‘Z1003’;

Los nulos no ocupan espacio, por como Cassandra almacena los datos. Cassandra es una base de datos **sparse**.

#### Tipos de datos

##### COUNTER

Columnas contadores, como cantidad de parciales y cantidad de finales

NO SE PUEDEN USAR DENTRO DE LA PRIMARY KEY

Por qué implementa contadores Cassandra? Es muy probable que 2 clientes me están actualizando un contador. Tener un tipo de dato que lo resuelva es mucho más fácil

CREATE TABLE examenes\_por\_alumno\_materia (

legajo TEXT,

nombre\_materia TEXT,

año INT,

parciales COUNTER,

finales COUNTER,

PRIMARY KEY ((legajo, nombre\_materia), año)

);

##### COLLECTION

Son lo mismo que JAVA: SET, LIST y MAP

###### SET<Tipo>

CREATE TABLE materias\_aprobadas\_por\_alumno (

legajo TEXT,

año INT,

materias SET<TEXT>,

PRIMARY KEY (legajo, año)

);

UPDATE materias\_aprobadas\_por\_alumno

SET materias = materias + { ‘I-NoSQL’ }

WHERE

legajo = ‘111111-1’ AND

año = 2010

Recordar que en el SET no pueden existir elementos duplicados y no hay un orden.

###### LIST<Tipo>

CREATE TABLE materias\_aprobadas\_por\_alumno (

legajo TEXT,

materias LIST<TEXT>,

PRIMARY KEY (legajo));

UPDATE materias\_aprobadas\_por\_alumno

SET materias = [‘Algebra’, ‘I-NoSQL’]

WHERE legajo = ‘111111-1’;

UPDATE materias\_aprobadas\_por\_alumno

SET materias = materias + [‘I-NoSQL’]

WHERE legajo = ‘111111-1’;

UPDATE materias\_aprobadas\_por\_alumno

SET materias = [‘I-NoSQL’] + materias

WHERE legajo = ‘111111-1’;

UPDATE materias\_aprobadas\_por\_alumno

SET materias[1] = ‘I-NoSQL’

WHERE legajo = ‘111111-1’;

ATENTI a la operación ‘+’ porque en la lista es distinto si sumo al principio o al final.

###### MAP<Tipo, Tipo>

CREATE TABLE fecha\_aprob\_por\_alumno (

legajo TEXT,

materias MAP<TEXT, INT>,

PRIMARY KEY (legajo));

UPDATE fecha\_aprob\_por\_alumno

SET materias = {‘Algebra’: 2010, ‘I-NoSQL’:

2011}

WHERE

legajo = ‘111111-1’;

UPDATE fecha\_aprob\_por\_alumno

SET materias[‘Algebra’] = 2011

WHERE

legajo = ‘111111-1’;

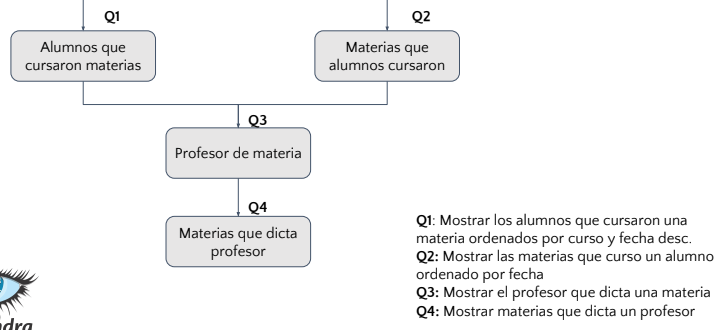
Dos tipos que no vimos: **uuid y timeUUID**

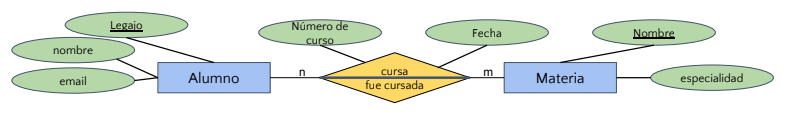
****

#### Principios de modelado en Cassandra

* Conocer tus datos: En general se utilizan DER para conocer las entidades del problema y sus campos. Se captura en el modelo conceptual. Es decir, no me alcanza “solo” con las consultas.
* Conocer tus queries: Conocer las queries y las interacciones entre las queries es crucial para un buen diseño. En general, la solución puede tener una tabla por cada query solicitada
* Anidar tus datos: Esto hace referencia al uso de colecciones dentro de una tabla, o a las particiones de múltiples row. Está bueno anidar nuestros datos con row keys
* Duplicar tus datos: Perderle el miedo.

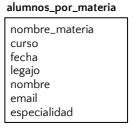
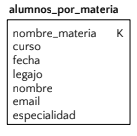
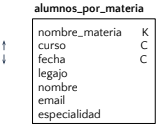
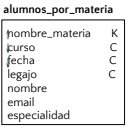
##### Workflow





Mi punto de entrada va a ser una query y tiene que ver con las opciones que le voy a dar al usuario.

Por cada query, voy a hacer estas 5 reglas para **pasar al modelo lógico.** El ejemplo se va a hacer tomando Q1

* MR1: Entidades y relaciones mapean a tablas.
  + Paso TODO atributo a una tabla
  + 
* MR2: Atributos de búsqueda por igualdad se convierten en partition key
  + Quiero buscar una **materia** en especial, por lo tanto convierto a **nombre\_materia** en Partition Key
  + 
* MR3: Atributos de inecuación se convierten Clustering columns
  + No quiero saber si algún campo es mayor a otro. Acá aplicaría si por ejemplo quisiera consultar por una fecha mayor a cierto punto en el tiempo (convirtiendo hipotéticamente a fecha en Clustering Column).
* MR4: Atributos de ordenamiento se incluyen en Clustering column con orden adecuado
  + Queremos un orden particular para los cursos y las fechas. Entonces los agregamos con el clustering column correspondiente (con la flechita)
  + 
* MR5: Atributos clave de entidades se convierten en Clustering column para garantizar **unicidad**
  + Con la regla 4 ya podemos hacer la query que queremos sin problemas. Pero es importante agregar el legajo, porque nos aseguramos que todas las combinatorias de datos que tenemos que formar se metan el PK (por ahí no resolver la query de Cassandra, simplemente para poder insertar todos los datos que queremos).
  + 

El último paso es en base a esa tabla lógica, pasarla a una tabla física:

CREATE TABLE alumnos\_por\_materia (

nombre\_materia TEXT,

curso TEXT,

fecha TIMESTAMP,

legajo TEXT,

nombre TEXT,

email TEXT,

especialidad TEXT,

PRIMARY KEY ((nombre\_materia), curso, fecha, legajo)

WITH CLUSTERING ORDER BY (curso DESC, fecha DESC, legajo DESC);

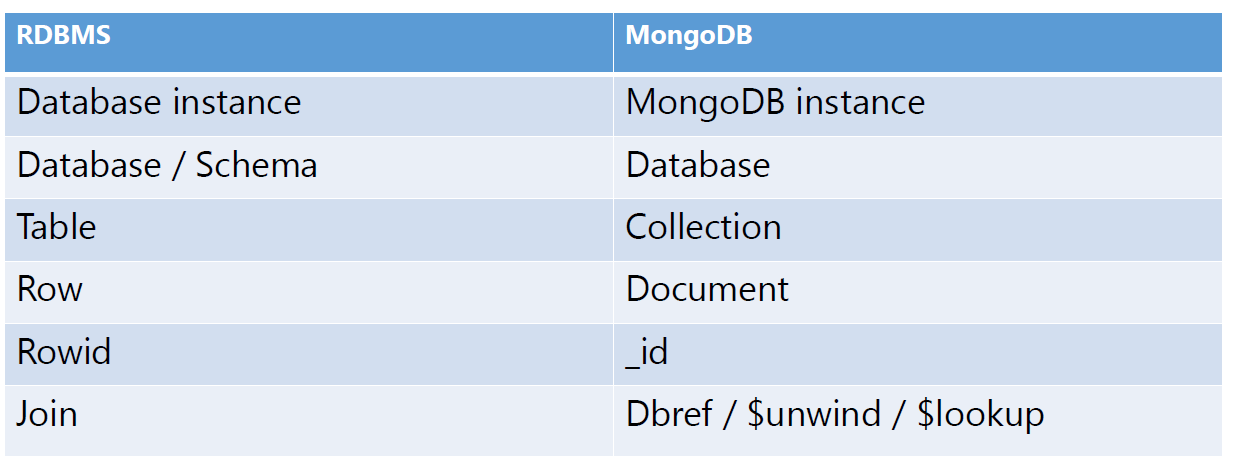
Con esto se resuelve la Query 1. Se debe ir iterando **por cada query** que se desee implementar, siguiendo el workflow y aplicando las mismas 5 reglas.

# Documentales

* Las bases de datos almacenan y recuperan documentos que **pueden ser XML, JSON, BSON, et**c.
* Estos documentos son estructuras de datos en forma de árbol jerárquico que consisten de mapas, colecciones, y valores escalares.
* Los documentos almacenados son similares unos con otros pero no necesariamente con la misma estructura.

Mongo

* MongoDBguarda estructuras de datos en documentos tipo JSON(JavaScript ObjectNotation) con un esquema dinámico.
* BSON está diseñado para tener un almacenamiento y velocidad más eficiente. Internamente MongoDB almacena los datos en formato BSON y con Btree con la aparición de WiredTiger



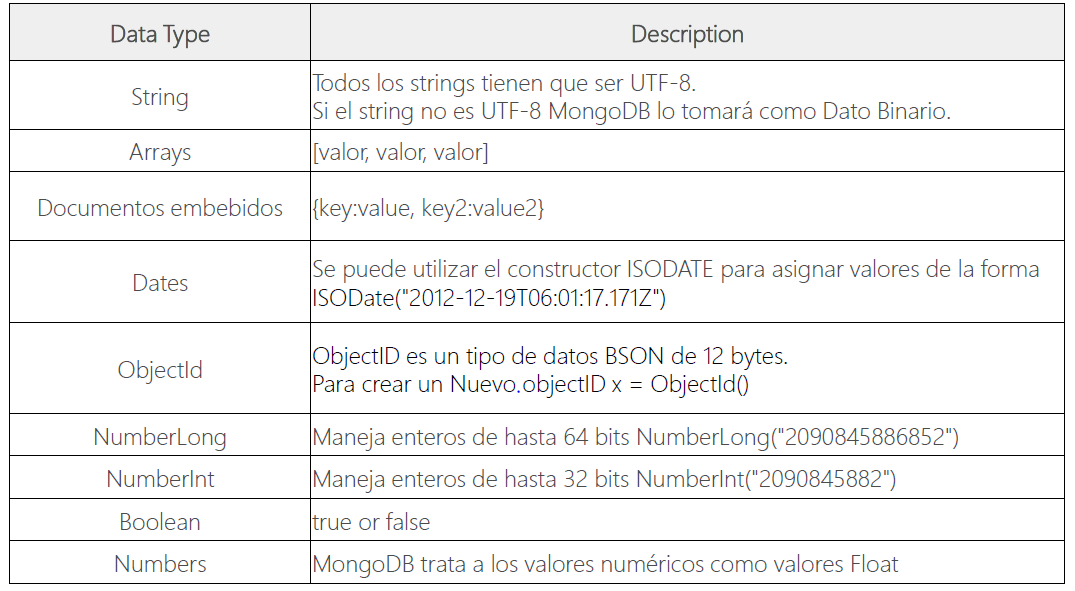
\_id: lo pone Mongo por default Mongo. este hace referencia al server en el que está guardado el dato.

joins: se hacen referenciando a otros \_id. No tienen tanto sentido, los evito.

schema: igual a una DBR.

* Los **documentos no pueden ser mayores a 16Mb**, pero existen objetos denominados **GridFS que permiten almacenar info de gran tamaño** (Gb/Tb). **Segmenta los archivos.** Se utiliza para fotos por ejemplo.
* Si hay información que frecuentemente es **consultada en conjunto,** lo conveniente es **almacenarla junta en un documento.**
* Los **JOINs se hace a mano, por aplicación.**
* Ejemplo de la editorial: Si cuando busco un libro quiero acceder a la información de la editorial, pero tengo 10mil libros entonces no puedo guardar los libros dentro de la editorial, pongo los datos que necesito de la editorial dentro del libro, + la referencia con el id a la editorial. Si en cambio los libros no fueran tantos, me convendría almacenarlos dentro de la editorial.

### Tipos de datos



JSon no tiene date.

### Cuándo usar DB documentales

MongoDB tiene un **lenguaje de consulta de datos basado en JSON.**

* Contiene construcciones para resolver la cláusula “where”, para ordenar los datos, o para mostrar el plan de ejecución.
* Contiene también la posibilidad de **expresar consultas mediante expresiones** regulares: db.orders.find({“items.product.name”:/Refactoring/})
* Puedo **acceder a los datos por cualquier atributo.**
* Utilización de **índices** para el rápido acceso.

**Logging de Eventos**

* las bases de datos basadas en documentos puede loguear cualquier clase de eventos y almacenarlos con sus diferentes estructuras.
* Pueden funcionar como un repositorio central de logueo de eventos.

**CMS, blogging**

* su falta de estructura predefinida hace que funcionen bien para este tipo de aplicaciones.

**Web-analytics/ Real-Time analytics**

* Almacenar cantidad de vistas a una página o visitantes únicos.

**E-Commerce:**

* A menudo requieren tener esquemas flexibles para los productos y órdenes

### Cuándo no usar DB documentales

**Transacciones Complejas con diferentes operaciones**

* **Soportadas sólo en RavenDB y en MongoDB recién a partir de la versión 4.0**

**Consultas contra estructuras de agregados variables**

* Que los datos se almacenen con cualquier estructura no implica que sea óptimo consultar por cualquier clave. Si los agregados varían entre sí, las consultas debieran variar también. Puede llevar a normalizar los datos, que no es lo que queremos.

### Comandos en MongoDB

**Cursores**

Ejemplo 2 – Script para consultar ciertos documentos de una colección e insertarlos en otra colección:

{ var cursor= db.facturas.find().limit(100);

while ( cursor.hasNext() ) {

db.facturasElegidas.insert(

{nroFactura:cursor.next().nroFactura,

fechaEmision:cursor.next().fechaEmision,

estado:"PEND" } );

}

}

**Consultar todos los elem**

db.facturas.find();

**Primer documento**

db.facturas.findOne();

**Límite de documentos**

db.facturas.find().limit(2);

**Saltear los dos primeros y hacerlo lindo**

db.facturas.find().skip(2).pretty();

**Buscar todo el doc por condición**

db.facturas.find( { condPago : “CONTADO” } ,{})

devuelve todos los elementos que cumplen

**Buscar por condición mostrando X atributos**

db.facturas.find( { “item.precio”:490 } ,{nroFactura:1,item:1})

\_id:0 para que no aparezca el id

Ojo con los tipos de datos. Es importante al comparar tener el cuenta el tipo de dato

**Mayor que**

db.facturas.find( { nroFactura : {$gt:1465} } ).count()

**Mayor o igual**

db.facturas.find({fechaEmision:{$gte: ISODate("2014-02-24T00:00:00Z")} } )

$gt

$gte

$lt

$lte

$not

$or

$in

$nin

$exist

$regex

**No cumpla una condición**

db.facturas.find({ nroFactura : { $not: {$gte:1500} } } )

**Regex**

* **i la búsqueda No es Case Sensitive.** (se puede usar tbién en expresiones regulares de Javascript)
* **m chequea la expresión contra multilíneas.** (se puede usar tbién en expresiones regulares de Javascript)
* **x** “extended” capability. En este caso **$regex ignora todos los espacios en blanco entre caracteres.**(sólo se puede utilizar con $regex)
* **s habilita el character “.”** Para matchear todos los caracteres incluso el character de newline.

Ej: Ambos listan todas las facturas donde el apellido del cliente contiene los textos “lav” y “no” en cualquier parte del texto, siempre y cuando “Lav” está antes de “noi”, sin tener el cuenta el Case Sensitive (n=N).

db.facturas.find( { “cliente.apellido”: /Lav.\*noi/i } );

db.facturas.find({“cliente.apellido”:{$regex:'Lav.\*noi',$options:'i'}} );

**Sort**

db.facturas.find( {}},{nroFactura:1,fechaEmision:1}).sort({fechaEmision:1})

**Exists**

db.facturas.find( {fechaEmision:{$exists:true}}, {nroFactura:1,fechaEmision:1}).sort({fechaEmision:1})

**HasNext valúa si un cursor tiene próximo elemento**

**Next Devuelve el próximo elemento**

{

var cursor= db.facturas.find().limit(100);

while ( cursor.hasNext() ) {

print(‘Factura Nro: ‘+cursor.next().nroFactura);

}

}

db.**getLastError();** devuelve el último error.

**Update de documentos**

Permite modificar uno o más documentos de una colección. **Por default modifica sólo un documento.**

db.coleccion.update ( {clausula\_where},

{documento\_o\_expresión\_a\_modificar}

,{ upsert

, multi

, writeconcern }

)

* **upsert** (true o false)Si está configurado en “True” significa que **realizará un update si existe un documento que concuerda con el criterio, o un insert si no existe algún documento que concuerde con el criterio.** El valor default es “false”, en este caso no realiza un insert cuando no existe documento que concuerde con el criterio.
* **multi** (true o false) Es opcional. Si es configurado en true, el **update realiza la actualización de múltiples documentos que concuerdan** con el criterio cláusula where. Si es configurado en false, modifica solo un documento. El valor default es false. Sólo actúa en updates parciales con operadores $.
* **writeconcer**n Es opcional, lo veremos en la parte de consistencia.

Update Totales/Completos: Se realiza el update del documento completo, reemplazando el mismo.

Update Parciales Operadores

* Operadores sobre cualquier atributo
  + **$set Permite modificar el valor de un atributo**, o agregar un nuevo atributo al documento.
  + **$unset Permite eliminar un atributo** de un documento.
  + **$inc** Incrementa o decrementa el valor de un atributo ( n ó –n)
* Operadores sobre Arrays
  + **$push Agrega un elemento a un Array o crea un Array con un elemento.**
  + **$addtoSet Agrega un elemento al Array solo si no existe** en el Array.
  + **$pushAll Agrega varios elementos a un Array con los valores indicados o crea un Array con esos elementos.** (Operación Múltiple)
  + **$pop Elimina un elemento de un Array por sus extremos,** permitiendo eliminar el primer elemento (-1)o el último (1).
  + **$pull Elimina todos los elementos de un Array que contengan** el valor indicado.
  + **$pullAll Elimina todos los elementos de un Array que contengan alguno de los valores indicados**. (Operación Múltiple)

Update TOTAL: db.updtst.update({x:2},{"x" : 2, "y" : 999 })

Updateará el primer documento que cumpla con la condición.

**Save**

mydoc.\_id="NuevoDoc"

db.facturas.save( mydoc )

db.facturas.find({nroFactura:1449}).pretty

Update PARCIAL: db.updtst.update({\_id:100},{$set : {x:100}})

Modifica el atributo x a 100.

db.updtst.update({x:2},{$set : {z:”NW”}},{multi:true})

**Cambia el valor de z en todos los docs cuando x=2**

db.updtst.update({ \_id:300}, { $unset: { y: 1})

**elimina el atributo y.**

db.updtst.update({\_id:100},{$push : {items:”elemArray”}})

Si no existe el array, lo crea.

db.updtst.update({\_id:100},{$addToSet: {items:”elemArray”}})

set sin repetidos

db.updtst.update({ \_id:300 }, { $pop: { items: -1 }})

**elimina el primer elemento del array**

db.updtst.update({ \_id:300 }, { $pull: { items: 6 }})

**elimina el elemento del array con valor 6**

db.accesos.update({\_id:”/materias/NoSQl”},{$inc : {visitas:1}}, {upsert:true})

**upsert: inserta si no existe, update si sí.**

db.updtst.remove({\_id:100})

eliminar documento

**Índice**

db.facturas.**ensureIndex**({"cliente.region":1,condPago:-1})

La clave del índice estará compuesta por los atributos cliente.region y condPago, el primero ascendende y el segundo descendente.

**Explain plan**

Dice los pasos realizados para cumplir con una consulta

**Consultar índices**

db.facturas.getIndexes()

**Eliminar índices**

db.facturas.dropIndex({fechaEmision:1})

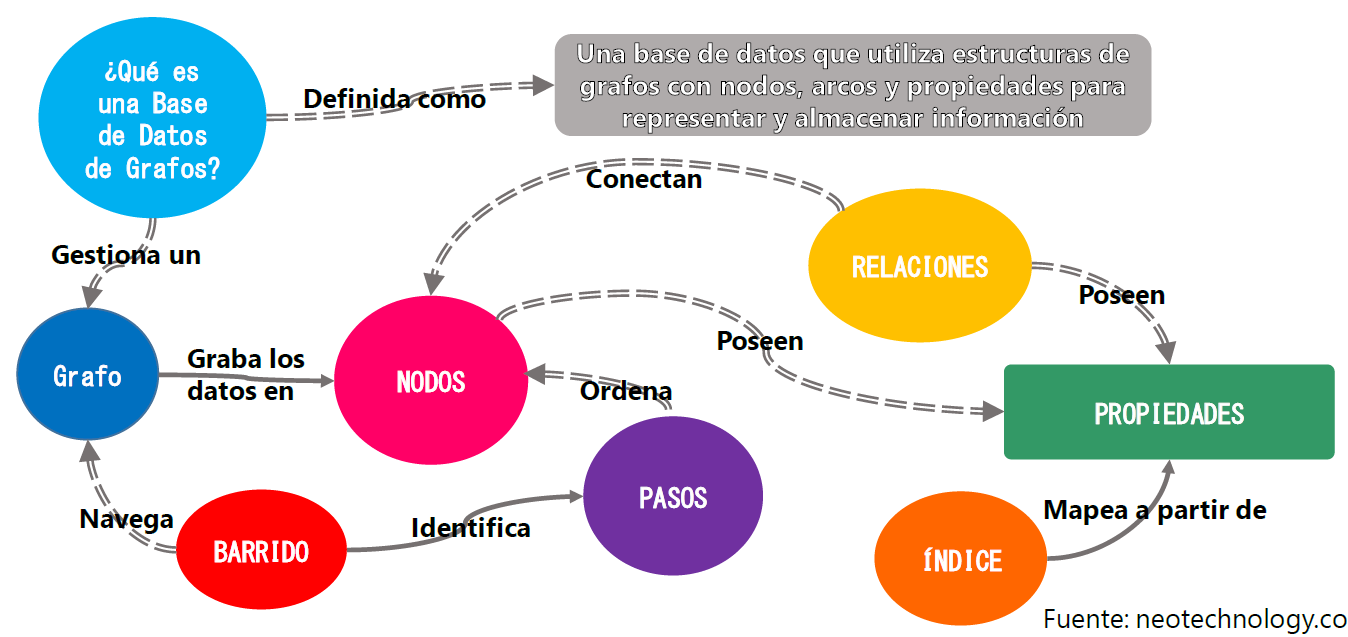
que la clave sea fechaEmisión:1

db.facturas.dropIndex(“fechaEmision\_1”)

Borrará el índice cuyo name sea igual a “fechaEmision\_1”

# Grafos

* **Presentan consistencia y disponibilidad**. **No se pueden escalar horizontalmente** pero se pueden hacer diferentes grafos con diferentes agrupaciones.
* Similares a las **struct con punteros, en donde puedo guardar información en la relación.**
* Podría representarlo con una matriz con 0 y 1 para representar las conexiones.
* Podría tener dos tablas, 1 de relaciones y otra de nodos.



* Las bases de datos de grafos nos permiten **almacenar entidades y relaciones sobre esas entidades.**
* Las entidades las denominamos **Nodos que tienen propiedades.** El nodo es el dato.

-Con buenas relaciones buenos índices puedo tener buenas búsquedas parecidas a las relacionales (?)

* Podemos hacer una analogía entre un Nodo y un objeto instanciado en una aplicación
* Las **relaciones denominadas Arcos pueden también tener propiedades.Son direccionales.**
* Neo4j almacena grafos y procesa grafos. **Hay algunas BD de grafos hechas sobre Cassandra.**
* Los nodos están organizados por relaciones que nos permiten encontrar patrones entre estos.
* La organización del Grafo permite que los datos sean **almacenados una vez** y luego **interpretados de distintas maneras basándonos en sus relaciones**.
* Las Bases de datos de Grafos soportan acceso a sus datos a través de lenguajes de consulta tales como Gremlin (lenguaje específico para recorrer grafos).
* Gremlin puede ser utilizado en todas las bases orientadas a Grafos que implementen “Blueprints property graph” (\*).

Cuándo usarlas

* **Datos interconectados:**
  + **Redes sociales** (likes, amigos, seguidores, etc.) ó laborales, por ej. representar a los empleados, sus conocimientos, y la relación de trabajo con otros empleados en diferentes proyectos
  + **Relaciones entre entidades de dominio de diferentes dominios** (por ejemplo, sociales, espaciales, comercio, redes y operaciones de IT, identidad y gestión de accesos) en una sola base de datos, puede hacer estas relaciones más valiosas, proporcionando la capacidad de recorrer a través de dominios.
* **Servicios de Ruteo, Despacho**
  + Cada ubicación en nuestra red de despacho es un nodo, las relaciones pueden contener la distancia entre las ubicaciones como propiedad.
* **Motores de Recomendaciones:**
  + Como los nodos y las relaciones se crean en el sistema, que pueden ser utilizados para hacer recomendaciones como "sus amigos también han comprado este producto" o "al facturar este artículo, estos otros artículos suelen ser facturados“ en tiempo real.
* **Sistemas con búsquedas recursivas con n niveles.**
* **Búsqueda de patrones en las relaciones para detectar el fraude** en las transacciones en tiempo real.
* **Gestión de datos maestros**: líneas de organización y producción que naturalmente se modelan como grafos.

Cuándo NO usarlas

* Sistemas que requieren de a**ctualizaciones masivas** sobre todas las entidades o un conjunto de entidades para un atributo o conjunto de atributos específicos.
* Sistemas que requieren una **alta distribución** de datos debido a su gran tamaño.

Neo4j

* Neo4J también cuenta con el lenguaje de consulta **Cypher** para recorrer grafos.
* Cypher usa las keywords **MATCH para búsqueda de patrones** en relaciones, **WHERE para filtrar propiedades en un nodo o relación,** **RETURN** especifica qué retorna una consulta.
* Cypher también provee métodos para operar los datos tales como **ORDER, AGGREGATE, SKIP, LIMIT**, entre otros.
* Se pueden crear procedures y hacer consultas en ambos sentidos.
* Aparte de estos lenguajes de consulta, Neo4J nos permite consultar las propiedades de los nodos, recorrer los grafos, o navegar por las relaciones de nodos utilizando “bindings” con diferentes lenguajes de programación (Java/.NET/JavaScript/Ruby/Python/PHP/R/Perl...).
* Índices. Se pueden crear índices sobre propiedades de un nodo.
* Se pueden aplicar **filtros direccionales** (entrantes, salientes ó ambos).
* Estructura de almacenamiento: guarda 3 archivos. **Levanta a memoria nodos y relaciones. Trae a memoria las propiedades a medida que se las pido**.
* El **tamaño del nodo es fijo**

RDBMS vs Grafos

* RDBMS: referencias a otras filas y tablas referenciando sus atributos clave mediante columnas de clave foránea.
  + Joins: uso intensivo de procesamiento y memoria y costo exponencial.
  + Si se usan relaciones de muchos a muchos, se debe introducir una tabla de join que guarde las FK de ambas tablas involucradas, lo cual incrementa más aún el costo de las operaciones de join.
* A pesar de su nombre, las BDs relacionales no guardan las relaciones entre los datos. Sino que **guardan referencias.** Por eso **no son adecuadas para los datos altamente conectados** de hoy en día. Señales de relaciones:
  + Gran número de Joins
  + Numerosos auto-joins(joins recursivos), comunes para representaciones de árbol o jerárquicas.
  + Frecuentes cambios de esquema
  + Queries lentos a pesar de haber sido tuneados
  + Resultados precomputados
  + Estos síntomas pueden indicar que se está intentando solucionar un problema de grafos con una BD relacional, donde el valor deriva de las relaciones entre los datos.
* Relaciones: **elementos privilegiados del modelo de grafos (sin necesidad de deducir las conexiones usando FK** o procesamiento externo.
  + El modelo de grafos permite construir modelos sofisticados que **mapean exactamente al dominio del problema.**
  + Próxima generación de las RDBMS, pero con soporte de primera clase para las relaciones. Equivalente a operación de **join: recorrido de listas de relaciones con acceso directo a los nodos conectados**.

Cypher

* Lenguaje **declarativo**: **enfoque en el dominio** y no en la BD (qué recuperar y no cómo).
* Cláusulas para lectura y para escritura, que se encadenan entre sí.
* **Actualización del grafo dentro de una transacción**.

### Nodo

Unidad fundamental de almacenamiento de información.

(identificador)

(identificador: etiqueta)

**(identificador: etiqueta { propiedades } )**

Ej.:

(nodo1: Persona { apellido:"Corrado", nombre:"Gustavo" } )

### Relaciones

Son claves para identificar la información relacionada en el grafo.

Tienen dirección y tipo.

(nodo1)-->(nodo2)

(nodo1)-[identificador]->(nodo2)

(nodo1)-[identificador:TIPO]->(nodo2)

**(nodo1)-[identificador:TIPO{ propiedades }]->(nodo2)**

(nodo1)--(nodo2)

Ej.: (Gonzalez)-[:ESTUDIO { estado: "Completo" }]->(CsEs)

### Consultas

Obtener la lista de personas que estudiaron carreras que no son de nivel “Universitario”, y los nombres de las carreras.

MATCH (a:Persona) -[:ESTUDIO]->(b:Carrera)

**WHERE b.nivel <> "Universitario"**

RETURN a, b

Obtener los nodos de todas las personas con nombre terminado en a.

MATCH (a:Persona)

**WHERE a.nombre =~ ".\*a"**

RETURN a

#### Create

Crear un nodo para la persona Analía Martinelli.

CREATE (n:Persona {nombre: "Analía", apellido: "Martinelli"})

**RETURN n**

#### Create unique

Si Analía Martinelli no lo posee, crear el conocimiento “Cálculo” asociado a ella.

**MATCH (n:Persona {nombre: "Analía", apellido: "Martinelli"})**

**CREATE UNIQUE (n)-[p:POSEE]->(c:Conocimiento {nombre: "Cálculo"})**

RETURN n, p, c

#### Create relación

Crear para Analía Martinelli una relación Estudio con la carrera "Lic en Sist de Inf", con estado “En curso”.

MATCH (n:Persona { nombre: "Analía", apellido: "Martinelli" }),

(c:Carrera { nombre: "Lic en Sist de Inf"})

**CREATE (n)-[r:ESTUDIO { estado: "En curso"}]->(c**)

RETURN n, r, c

CREATE UNIQUE es algo intermedio entre MATCH y CREATE. **Obtendrá la coincidencia que pueda, y creará lo que falta, haciendo el menor cambio posible al grafo.**

Analía Martinelli conoce a Verónica Mendez. Crear la relación asegurando que se cree una sola vez.

MATCH (a:Persona {apellido: "Martinelli"}),

(b:Persona {apellido: "Mendez"})

CREATE UNIQUE (a)-[r:CONOCE\_A]->(b)

RETURN r

#### Set

Agregarle a Analía Martinelli la etiqueta “Empleado” y el país Argentina.

MATCH (a:Persona {apellido: "Martinelli"})

SET a:Empleado,

a.pais="Argentina"

RETURN a

#### Remove

Eliminar de Analía Martinelli la fecha de nacimiento y la etiqueta Persona.

MATCH (a:Persona {apellido: "Martinelli"})

REMOVE a:Persona,

a.fechanac

RETURN a

#### Delete

Eliminar del grafo a Analía Martinelli y todas sus relaciones.

MATCH (a { nombre: "Analía", apellido: "Martinelli"})-[r]-()

DELETE r, a

#### Cualquier dirección

Se desea saber si mediante la relación CONOCE\_A es posible llegar directa o indirectamente **(con caminos en cualquier dirección)** desde Mario López hasta Jorge Lupis.

MATCH camino=(a {nombre:"Mario", apellido:"López"})-[:CONOCE\_A\*]-

(b {nombre:"Jorge", apellido:"Lupis"})

RETURN camino

#### Relationships

Relationships **devuelve las relaciones de un camino como colección**

MATCH camino=(a {nombre:"Mario", apellido:"López"})-[:CONOCE\_A\*]-

(b {nombre:"Jorge", apellido:"Lupis"})

RETURN relationships(camino)

**With**

La cláusula WITH **separa explícitamente partes del query**, permitiendo declarar qué identificadores arrastrar hacia la parte siguiente. Un uso habitual es para **limitar la cantidad de resultados que pasarán a otra cláusula MATCH**. Otro uso es para filtrar valores agregados.

Obtener la lista de personas y cantidad de conocimientos registrados de cada una de ellas, para las que registraron más de 3 conocimientos.

MATCH (a: Persona)-[:POSEE]->(c:Conocimiento)

**WITH a, count(c) AS conocimientos**

WHERE conocimientos > 3

RETURN a.apellido, a.nombre, conocimientos

# Distribución de datos

Problemas

* Sistemas de Base de Datos que administran grandes volúmenes de información.
* Sistemas que tienen alta interacción con diferentes y múltiples usuarios concurrentes.
* Muy alta tasa de consulta.
* Podría superar la capacidad de responder de una única CPU.
* Existencia de **grandes volúmenes de datos.**
* Pueden **exceder la capacidad de almacenamiento de una sola máquina**.
* El Tamaño de bloques de datos que se transfieren de disco a memoria.
* Pueden estresar la capacidad de E/S de los dispositivos de disco.

Clúster

**Grupo de servidores independientes interconectados a través de una red dedicada que trabajan como un único recurso de procesamiento**

Uno de los primeros requerimientos que justificaron la aparición del movimiento noSQL fue la necesidad de poder trabajar de forma eficiente en ambientes basados en clústers.

Los clusters

* **Distribuyen la carga entre los servidore**s interconectados.
* **Mejoran la disponibilidad** de los usuarios.
* **Mejoran la performance** total.
* **Mejoran la tolerancia a fallas**.
* Un servidor caído es inmediatamente inactivado y los usuarios redirigidos al resto.

No es solo útil para grandes empresas. La mayoría necesita garantizar disponibilidad.

Escalamiento vertical

* Escalamiento dentro de **un mismo servidor.**
* No trae problemas de concurrencia.
* **Incrementar capacidad de un Servidor** agregando CPU, almacenamiento, memoria.
* Las DB de grafos sólo pueden escalar verticalmente.
* Se puede particionar una misma aplicación

Escalamiento horizontal

* **Replicación.**
  + Replicación Maestro-Esclavo (Master/Slave)
  + Replicación Esclavo-Maestro-Esclavo (Slave-Master-Slave)
  + Replicación entre pares(peertopeer)
* **Particionamiento.**
* **Combinación** entre Replicación y Particionamiento.

### Replicación

* Proporciona **redundancia e incrementa la disponibilidad** de datos
* Con **copias múltiples en diferentes servidores**, la aplicación **protege a una base de datos de la caída** de un servidor.
* P**ermite recuperarse de las fallas de hw** e interrupción de servicio.
* Puede dedicar a una estrategia de recuperación ante desastres.
* Almacenamiento dedicado a la generación de reportes
* **Puede traer problemas de consistencia**

#### Master-Slave

* Un nodo maestro o primario
* El nodo **maestro es fuente de los dato**s y generalmente se encarga de **procesar cualquier actualización** de los mismos
* La r**eplicación sincroniza los nodos secundarios desde el maestro**
* La l**ectura** de datos puede realizarse desde c**ualquiera**
* Ante la c**aída del maestr**o, se puede **espera**r a que se recupere o designar un e**sclavo como maestro.**
* Maestros se pueden definir **manualmente o automáticamente** por el sistema.
* **No sirve para escalamiento de escrituras**
* Pueden haber **inconsistencias en las lecturas**
* Óptimo cuando se realizan **pocas actualizaciones y muchas lecturas**
* C**apacidad de recuperarse ante fallas para operaciones de lecturas.** Si cae el maestro, sigue andando.

#### Slave-Master-Slave

* **Todas las réplicas aceptan escrituras**
* La pérdida de algún nodo no evita al acceso a los datos
* Es f**ácil agregar n**odos que mejoren la performance y distribuyan la carga
* Cuando s**e escribe en secundario, se copia al primario y de ahí se replica a los otros secundarios.**
* Mejor performance, más paralelismo, menos latencia. Pero **Master sigue estando penalizado.**

#### Peer to peer

* Todas las **réplicas aceptan escrituras**
* La pérdida de algún nodo no evita el acceso a los datos
* Es f**ácil** **agregar nodos** que mejoren la performance y distribuyan la carga
* **Principal problema es la consistencia**
* **Conflicto escritura-escritura cuando dos nodos intentan escribir el mismo dato** simultáneamente.
* Pueden habe**r problemas de consistencia en las lecturas**

### Particionamiento (Sharding)

* Cada **partición es una DB independiente**
* Las particiones forman parte de una **única base de datos lógica**
* La carga se intenta b**alancear** entre todos los servidores disponibles
* Distribuye las distintas partes del conjunto de datos en distintos servidores que se denominan particiones
* En necesario garantizar que los datos que sea **accedan en conjunto estén también almacenados juntos en las particiones** = distribuir los datos de forma tal de garantizar los accesos.
* Se puede c**ombinar los agregados** que tienen mayor probabilidad de ser accedidos **juntos en conjunto.**
* También se pueden ubicar los agregados a ser leídos en conjunto uno a **continuación del otro**. Se **optimiza la lectura en bloques de datos.**
* Muchas DBs ofrecen el **autoparticionamiento: elige el motor**
* Muy importante: **puede mejorar los tiempos de lectura y escritura**
* Permite **escalar horizontalmente en aplicaciones con muchas lecturas y escrituras**
* La estrategia de particionamiento debe ser **decidida al momento de diseño del modelo de dato**s. Muy costoso de revertir un error.
* **No brinda una solución completa a la recuperación ante fallas. Debe combinarse con replicación.**

### Replicación y Sharding

* Tendremo**s múltiples maestros, pero cada ítem de dato estará en un único maestro**
* Un **nodo podría ser maestro para determinados datos y esclavo para otros**
* Estrategia usada en column-family

Replicación y distribución en Mongo

### Replicación en Mongo

* La **replicación** se realiza a través de **estructuras denominadas Replica Set**.
* Una Replica Set es un **grupo de procesos de mongo que mantienen el mismo conjunto de datos**. Las replica sets proveen **redundancia y alta disponibilidad.**

ReplicaSet:

* Utilizan un esquema derivado del **Maestro-Esclavo**
* Un n**odo (proceso mongodb)** se define como primario (maestro) y recibe todas las operaciones de escritura.
* Sólo puede haber un **único nodo primario**, por lo tanto se proporciona una consistencia estricta en la escritura.
* Los n**odos secundarios replican el log de operación del nodo primario y aplican las operaciones en sus datos asincrónicamente**.
* Si el nodo primario no está disponible, se **elige un secundario como primario (por votación).**
* Puede e**xistir un nodo árbitro que no almacena datos,** solo se encarga de **determinar el nuevo primario.**

#### Ejemplo

**Creamos un servidor** levantando una instancia

mongod--replSetrs\_clase6a –port 27058…..

use finanzas

db.facturas.insert({……..})

**Creamos 2 nuevas instancias** de mongoDB

mongod--replSetrs\_clase6a –port 27059…..

mongod--replSetrs\_clase6a –port 27060…..

**Configuramos un ReplicaSet con un servidor**

mongo --port27058

cfg={\_id:"rs\_clase6a",

members:[{\_id:0, host:“127.0.0.1:27058"}]}

rs.initiate( cfg)

**Agregamos al ReplicaSet nuevos servidores**

cfg={\_id:"rs\_clase6a",

members:[{\_id:0, host:"127.0.0.1:27058"},

{\_id:1, host:"127.0.0.1:27059"},

{\_id:2, host:"127.0.0.1:27060“} ,

]

}

rs.reconfig(cfg)

o también

Mongo --port 27058

rs.add({\_id:1,host:"127.0.0.1:27059"})

rs.add({\_id:2,host:"localhost:27060"})

**Le digo a los secondarys que son secondarys**

mongo --port 27059 --quiet

rs.slaveOk()

mongo --port 27060 --quiet

rs.slaveOk()

### Sharding en Mongo

MongoDB soporta particionamiento a través de **clusters particionados.**

Un cluster particionado tiene tres componentes:

* Particiones (mongod--shardsvr--replSet…)
* Ruteadores de Consultas (mongos --configdb…)
* Servidores de Configuración. (mongod--configsvr…)

#### Particiones

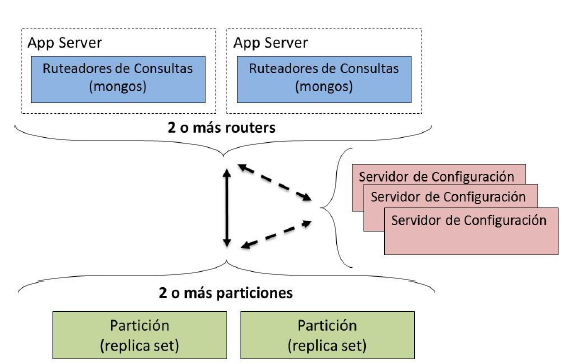
Cada partición almacena los datos. Para proveer alta **disponibilidad y consistencia cada partición debe ser un ReplicaSet**

#### Ruteadores de consultas

Son la interfaz con las aplicaciones, **procesan y dirigen cada operación a la partición correspondiente**. Un cluster puede tener más de un ruteador de consulta, para d**istribuir también la carga de los clientes a los ruteadores**.

#### Servidores de configuración

Almacenan la **metadata del clúster**, que contiene un **mapeo de los datos del clúster a cada partición**. El ruteador la utiliza para **direccionar las operaciones a la partición correspondiente**. En general los clústers de producción tienen 3.



* Para particionar es necesaria una clave de partición. La **shard-key** es un **campo indexado existente en cada documento de un agregado**.
* Mongo **divide los valores a través de la clave de partición en chunks** y **distribuye los chunks en cada partición**. Para ello utiliza dos esquemas:
  + Particionamiento basado en **rangos:**

Divide el conjunto de datos en **rangos disjuntos** determinados por los **valores de la clave de partición.**

Un chunk queda definido como: un rango de valores con un mínimo y un máximo.

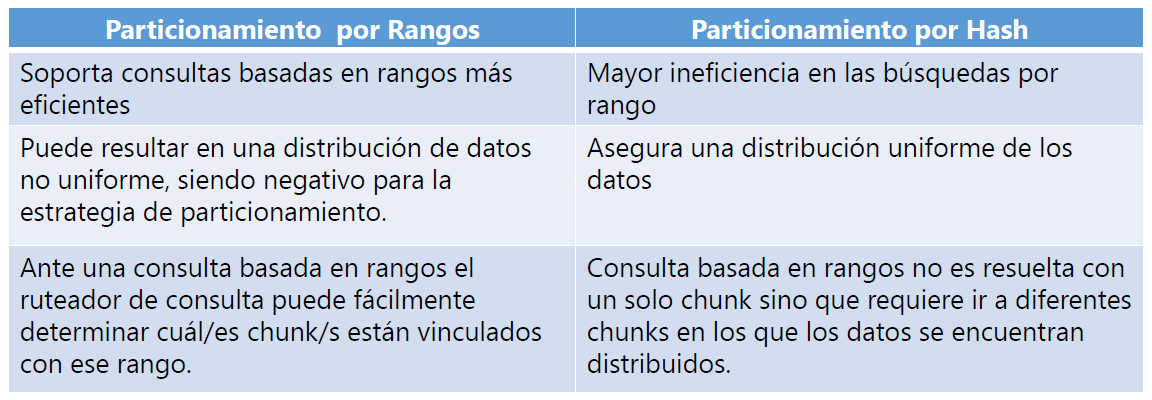
Los documentos que tiene **claves de particionamiento cercanos tiene más probabilidades de estar en el mismo chunk** y por ende en la misma partición.

* + Particionamiento basado en **hash**

Mongo **calcula el valor de hash de una clave** y utiliza ese valor para distribuir en chunks.

Dos documentos con valores de clave cercanos tienen baja probabilidad de estar en el mismo chunk

Asegura una **distribución aleatoria de los agregados en los clústers.**



#### Distribución balanceada

La a**dición de nuevos datos o de nuevos servidore**s puede resultar en una distribución de datos desbalanceada dentro de un clúster.

Una partición podría tener significativamente más chunks que otra.

Mongo asegura **clústers balanceados** con splitting y balancer.

* **Splitting**: Cuando un **chunk crece más del tamaño esperado, Mongo lo particiona en mitades**. Generadas por inserción y actualización. Para crearlas, Mongo no migra datos o afecta las particiones.
* **Balancer:** administra las migraciones de chunks. Se ejecuta, en un cluster, en los ruteadores de consultas. Cuando la distribución de una colección particionada en un clúster es no homogénea, **migra chunks hasta que el agregado se balancee.** Las particiones administran las migraciones de chunks como una operación que ocurre de fondo.

Ejemplo:

1. **Levantamos las instancias de MongoDB con el parámetro --shardsvr**

mongod –shardsvr --replSet r1 –port 27058…..

todas las instancias (3)

2. **Creamos y Levantamos un servidor de Configuración y creamos un ReplicaSet**

mongod--configsvr--port 27500 …..

**Nos conectamos al configsvr y creamos el RS**

(\*) mongo –port 27500

(\*) rs.initiate({\_id:"rs\_conf“, members:{\_id:0,host:"127.0.0.1:27500"}})

3. **Creamos y Levantamos un servidor de Ruteo**

mongos --configdb localhost:27500 (hasta versión 3.2)

(\*)mongos --configdb rs\_conf/127.0.0.1:27500

**4. Realizamos luego el agregado del primer shard**

mongo (nos conectamos al mongos al puerto default 27017)

sh.addShard(“r1/127.0.0.1:27058”)

(\*) Obligatorio a partir de versión 3.4

5**. Definimos la clave para realizar sharding y creamos un índice necesario.**

db.facturas.ensureIndex( { “cliente.region”:1, “cliente.cuit”:1 } )

**6. Activamos el Sharding para la base de datos finanzas.**

sh.enableSharding(“finanzas”)

**Ejecutamos Sharding sobre la Colección HistoriaSocio de la Base de datos GestionClub (mongos)**

sh.shardCollection(“finanzas.facturas”, {

cliente.region:1, cliente.cuit:1 },false)

Distribuye los datos de la colección en chunks.

Cada chunk "agrupa" los documentos entre un mínimo y un máximo.

**7. Configuramos un segundo Replica Set**

(idemr1) pasos 1 y 2

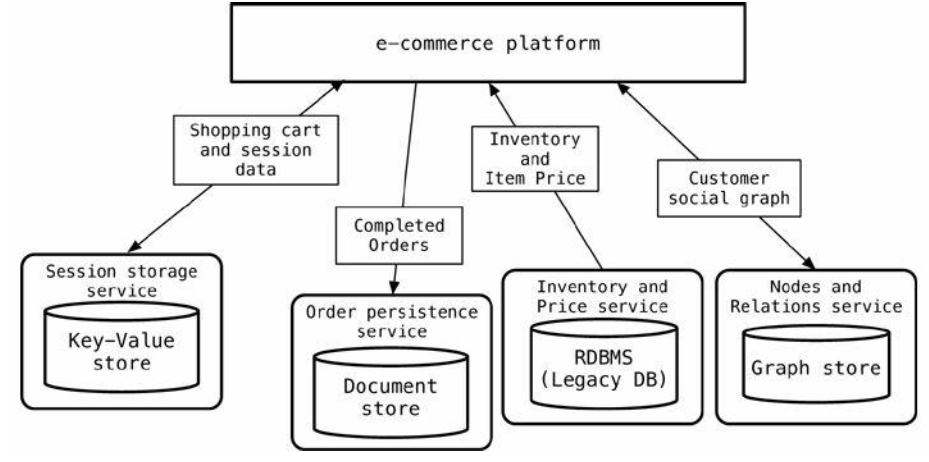
**8. Agregamos un nuevo Shard**

mongo

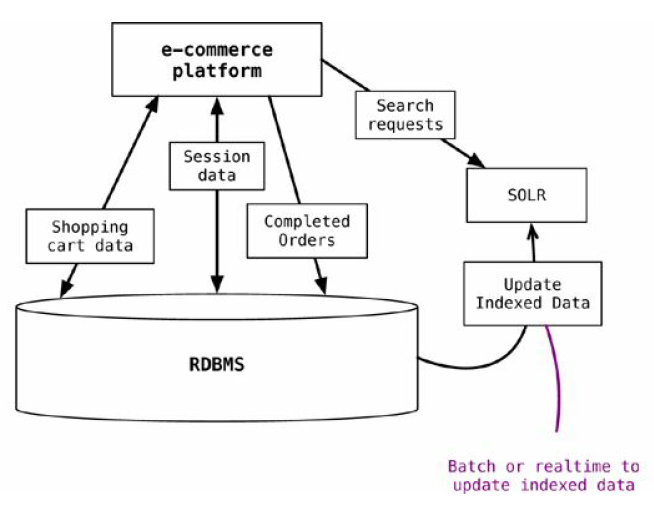
sh.addShard(“r2/127.0.0.1:27158”)

# Persistencia Políglota

* Diferentes bases de datos: **diferentes problemas**
* **Único motor** para todo: **soluciones de bajo rendimiento**
* Persistencia políglota: enfoque híbrido de la persistencia de los datos en distintos repositorios, optimizados para el tipo de problemas aplicable a cada parte. Usar las f**ortalezas de cada herramienta** en mi sistema.
* Como las bases de datos están construidas para satisfacer diferentes propósitos, no es necesario (ni eficiente) que una aplicación utilice un único tipo de repositorio para todas sus necesidades.
* El uso de RDBMS para OLAP y OLTP también puede ser visto como Persistencia Políglota.



Comunicarme a las BDs mediante APIs cada vez que quiero hacer una operación. Ventaja: que un equipo esté detrás de la API y se lo de a quien corresponda. Gobierno de datos.



BDsespecializadas

única BD (RDBMS) para todos los modelos de datos

soporte nativo de las soluciones

¿Podemos hacer recomendaciones mediante un RDBMS?: Al no ser un repositorio específico, de cambiar el recorrido, tendremos que migrar los datos y rearmar la base

Cuestiones a tener en cuenta

* C**ómo usar estos nuevos repositorios**
* C**ómo funcionan estas tecnologías**
* C**ómo monitorear estos sistemas**
* C**ómo hacer backup y restauración de datos**
  + Backups Incrementales
  + Restore point in time
* Cómo **manipular los datos**
* L**icenciamiento**
* **Soporte**: comunidad buena para resolver problemas? Importa cuán conocida es la DB.
* Herramientas complementarias
  + Importar/exportar archivos, Consulta Interactiva, etc.
* Actualizaciones: con qué frecuencia se actualiza la DB.
* **Drivers**
  + Drivers propios y de terceros, para diferentes lenguajes.
* Auditoría
* S**eguridad** de los datos
  + Crear Perfiles, Roles (built-in, user defined).
* Asignar privilegios/permisos para realizar diferentes acciones sobre los datos
* Validaciones contra Single sign on(LDAP, Kerberos, Active Directory)
* Encriptación de comunicación entre Servidores
* En general en sus comienzos las BD NoSQL, la seguridad yacía en la aplicación, aunque actualmente ya muchas BD introducen características de seguridad avanzada.
* Asegurar que las herramientas ETL y otros mecanismos puedan leer de los repositorios NoSQL.
* Asegurar que las herramientas de Explotación de Datos puedan utilizar correctamente Bases de Datos NoSQL.

Despliegue

* Todas las BDs **en producción al mismo tiempo**
* Todas estas BDs en los ambientes de Desarrollo, QA y UAT: probar las DBs en todos los ambientes

A evaluar

* **Costo / beneficio**
  + **Naturaleza del desarrollo**
  + **Forma de acceso a los datos**
* **Tipo de proyecto**
  + Frenos en el desarrollo
  + N**ecesidades intensivas de procesamiento** ¿justifica nueva tecnología? Tener en cuenta que si hago un cambio probablemente no tenga vuelta atrás.
* Proyecto nuevo o existente (menos importante)

Resumen

* Usar diferentes tecnologías de almacenamiento de datos para gestionar **necesidades variables** de almacenamiento.
* A través de toda la empresa o dentro de una aplicación.
* **Encapsular el acceso a los datos en servicios para reducir el impacto en otras partes del sistema. con APIs.** Me permite administrar los datos a mi gusto.
* Contrastar las ventajas de **elegir el mejor repositorio contra el incremento de la complejidad.**

# Consistencia

Uno de los Desafíos más grandes al pensar en Bases de datos Distribuidas en Clusters es la forma en la que se gestiona la consistencia de los datos.

Repasemos el Concepto de Cluster: Grupo de servidores independientes interconectados a través de una red dedicada que trabajan como un único recurso de procesamiento. Cada clúster tiene su hardware, su disco, su memoria.

Administrar la consistencia implica **mantener la información uniforme mientras se mueve a lo largo de una red o de varias aplicaciones.** Que la información sea **válida** según una serie de reglas predefinidas.

Las transacciones son el atributo de las DBR más apreciado. Pero las transacciones se quedaron cortas y se empezó a no pensar en la consistencia en términos absolutos. Se piensa en t**érminos de lo que es necesario** para cada aplicación particular.

Consistencia

El Acceso a datos en una base de datos distribuida es considerado **plenament**e consistente cuando **una actualización escrita en un nodo está inmediatamente** disponible en otro nodo del cluster.

La forma tradicional de lograr esta consistencia plena en un **RDBSs** mediante un esquema de **transacciones distribuidas.** Es decir, la o**peración se confirma cuando la escritura se publica en todos los nodos.**

Disponibilidad

Un sistema se define como de **alta disponibilidad** si g**arantiza respuestas** para todos los **requerimientos** que recibe, aún **inclusive** si uno o más **nodos** se encuentran c**aídos.**

Para cualquier base de datos con un solo nodo, esto es imposible de lograr ya que ese no es un punto único de falla (SPOF–Single Point of Failure).

Para **lograr la disponibilidad** de los datos en un cluster los datos deben replicarse a una cantidad de nodos, y **cada nodo debe estar listo para** reclamar t**ransformarse en maestro en cualquier momento**, ante la caída del mismo.

Tolerancia a particiones

Los nodos pueden estar físicamente separados uno del otro.

El lapso de **tiempo en el que no puedan alcanzarse entre sí**, sea por problemas de ruteo, problemas de interfaces de red o cuestiones de firewalls, se llama “partición de red”.

Durante la partición de red, t**odos los nodos deberían poder servir a los requerimientos de escritura y lectura**. Idealmente el **sistema automáticamente reconcilia las actualizaciones** tan pronto como cada nodo puede alcanzar al otro nuevamente.

Un sistema solo puede ofrecer a lo sumo dos de las 3 propiedades. Hay que tomar decisiones de cuál atributo sacrificar.

Muchas de las bases de datos NoSQL pueden configurarse para cambiar el tipo de CAP (por ejemplo, **MongoDB** está en el espacio **CP, pero podría configurarse para estar en el espacio CA**).

CA: DBR. No están distribuidas.

CP: En este espacio se privilegia la tolerancia a las particiones y la consistencia. Varios motores de bases de datos NoSQL ofrecen consistencia fuerte en desmedro de cierto nivel de indisponibilidad. Ellos son MongoDB, HBase, BigTable, entre otros. En este espacio, a**nte la eventualidad de una partición de red**, estas bases de datos pueden **no llegar a responder a cierto tipo de consultas**.

A**nte una partición de red** mongo lo resuelve, en donde h**aya más nodos**, voy a **permitir actualizar**, donde hayan **menos son read only**. Cuando grabo un dato, ese dato va a ser consistente, porque en mongo se graba por el master.

AP: Como Cassandra. Aunque estos motores ofrecen replicación de datos, no garantizan consistencia entre dos nodos.

Parámetros N R W

### N

* N es la cantidad de **réplicas requeridas** para considerar una **operación exitosa** en un clúster.
* Cada pieza de datos se replica a una cantidad de nodos N en un clúster.
* N representa el valor por defecto para el **clúster completo**, y en a**lgunas** DB puede definirse para **cada operación de lectura y escritura**.

### W

* Cuántas réplicas se **deben escribir** para que la **operación sea considerada exitosa. NO puede ser mayor a N** sino la operación resultaría en un error. Cada vez que se guarda un dato, cuántas ACK de que la operación fue exitosa tengo que recibir.
  + Valor de **W alto**
    - mayor **consistencia de los datos *escritos***
    - la operación **puede fallar** porque algunos **nodos pueden estar no disponibles o caídos.**
    - **afecta la velocidad** (latencia de red al haber más nodos y tener que esperar el ACK de todos)
  + Valor de **W bajo**
    - **Disminuye la consistencia del conjunto de datos**
    - Involucra menos nodos, requiere **tiempo menor**
    - Las lecturas en otras réplicas **pueden devolver valores desact.**

### R

* La cantidad de **nodos que se requiere devuelvan el mismo valor para que la operación** se considere **exitosa.**
* Los primeros R nodos en devolver el mismo valor formar parte del **quórum de lectura.**
* Valor de R
  + **R alto**
    - Nivel de **consistencia fuerte**
    - **afecta la performance**
    - con un **W bajo,** **fuerza al cluster a reconciliar piezas no actualizadas** que aparecen **escritas con cierta inconsistencia.**
  + **R bajo**
    - **lectura menos proclive a problemas de disponibilidad**
    - **baja la latencia de lectura**

### Ejemplos en RIAK

* Si W < N, una escritura puede considerarse exitosa aún inclusive mientras Riak está todavía copiando el valor en otro nodo.
* Si **R > N**(cantidad de copias disponibles), el requerimiento va a responder con un **error.**
* El valor **N**(Por default 3) puede c**onfigurarse para un Bucket (conj. de claves)** en particular.
* Puede definirse que la o**peración retorne inmediatamente y que Riak Siga replicando en background N nodos**. Ó se podría decidir e**sperar que Riak Replique** a los W nodos antes de retornar la operación.
* Lo mismo puede aplicarse a las lecturas con el valor R.
* Pero Riak Permite definir valores R ó W por Default para todas las operaciones, o definir valores R ó W para una operación particular.

### Tipos de consistencia

#### Consistencia por escritura

**W=N y R=1**

Esta forma establece **consistencia fuerte en la escritura** con lo que la c**onsistencia** en la **lectura está garantizada.**

Lo utilizan las DBR.

Garantiza lecturas más rápidas pero r**alentiza las escrituras.**

#### Consistencia por lecturas

**W=1 y R=N**

Se **pueden recuperar valores antiguos pero se elige el más reciente** (que va a ser el más reciente de todos).

#### Consistencia eventual

Garantiza que **si no existen nuevas actualizaciones que se hagan** eventualmente a un objeto, t**odos los accesos van a devolver el valor más actualizado en último término**.

El sistema más popular que define inconsistencia eventual es el de los DNS. Las actualizaciones a un DNS se distribuyen en función a coches controlados por tiempo.

Eventualmente el cliente puede ver o no la actualización, la consistencia no está garantizada aunque existen mecanismos que la intentan maximizar.

#### Consistencia por quorums

Resignar consistencia o disponibilidad no es una cuestión a todo o nada, presenta grises.

Cuanto mayor es la cantidad de nodos que se involucran en un requerimiento, más altas la chance de evitar inconsistencias.

Imaginemos datos replicados en tres nodos N=3.

No necesitamos verificar que en los tres nodos se haya realizado una escritura para lograr una fuerte consistencia(aunque no plena).

Lo que se necesita es que haya ocurrido en dos de ellos(una mayoría).

Esto es referido como el **quórum de escritura** y se expresa mediante la inecuación:

**W > N/2**

El **quórum de lectura** es un poco más complicado porque depende de cuántos nodos se necesitan para confirmar una escritura.

**R + W > N**

* Las inecuaciones se basan en general en un modelo de distribución entre pares.
* Si se tiene un modelo de distribución maestro-esclavo, se tiene evitan conflictos de escritura-escritura, y sólo con leer del maestro se evitan conflictos de lectura-escritura.
* No confundir el Factor de replicación N con la cantidad total de nodos en un cluster.
* Algunos afirman que N=3 es suficiente para tener una buena resistencia a fallas porque hace que un nodo pueda fallar dejando un backup para lecturas y escrituras hasta que se genera una terca réplica nuevamente. Es lo que hace Mongo.

Consistencia en Mongo

* Mongo da **ACK cuando se guarda en la memoria de Mongo**, pero podría no haberse persistido posta. Por eso agrega otro parámetro
* **J= true. hace que responda una vez persistido el dato**.

#### Defino w

cfg= rs.conf()

cfg.settings= {}

cfg.settings.getLastErrorDefaults= {w: "majority"}

rs.reconfig(cfg)

db.runCommand( { getLastError: 1, w: 2 } )

db.runCommand( { getLastError: 1, w: "majority" } )

db.runCommand( { getLastError: 1, w: 2, wtimeout:5000 } )

Yo puedo definir el **write concern** en un insert. Si inserto este dato con un **majority,** y de los 5 servers, por algún motivo n**o puede insertar en la mayoría**, me va **a avisar,** pero va a **grabar en los que pudo.**

Para una operación en particular

Write concern insert

db.products.insert(

{ item: "envelopes", qty: 100, type: "Clasp" },

{ writeConcern: { w: "majority", wtimeout: 5000 } }

)

#### Read concern

Para especificar un read concern para el metododb.coleccion.find()del shell de mongo, usar el método cursor.readConcern("<level>").

Levels:

* l**ocal:** Devuelve los **datos más recientes de la instancia a la que se le realiza la consulta** y no garantiza que haya sido escrita en una mayoría de los datos, por esto los datos pueden llegar a ser rollbackeados. Este es el comportamiento default.
* **majority**: El query devuelve los **datos más recientes de la instancia que haya sido escrita en una mayoría d**e los miembros del Replica Set.
* l**inearizable**: El query devuelve d**atos que fueron escritos en una mayoría de los miembros** del Replica Set. Para Replica Sets que corren con writeConcernMajorityJournalDefault en true, linearizable devuelve datos que nunca serán rollbackeados. Este tipo de read concern sólo puede ser especificado para lecturas sobre el Primary del Replica Set, y con queries que identifiquen a un único documento.

#### Read preference

Esto indica de qué nodo se leerá el dato que estemos pidiendo. Los disponibles son los siguientes:

* **Primary**(Default): Lee **únicamente del Primary**. En caso de no haber un Primary disponible, retornará error.
* **Primary Preferred**: Leerá **preferentemente** del Primary, pero si momentáneamente no hay ninguno leerá de un Secondary.
* S**econdary**: Leerá **únicamente de un Secondar**y. De no haber Secondaries, devolverá un error.
* **Secondary Preferred**: Leerá preferentemente de un Secondary, pero si sólo esta el Primary levantado, leerá de este.
* **Nearest:** Leerá del nodo que tenga **menor latencia de red.**

Utilizando el shell mongo, por default nos dejará leer únicamente del Primary, por lo que, al **querer leer de un Secondary, deberemos ejecutar rs.slaveOk()** indicando que a**ceptamos los datos del Secondary**, sabiendo que este puede no tener las últimas escrituras.

#### Schema validations

A partir de la versión 3.2. MongoDB provee la capacidad para validar documentos durante modificaciones e inserciones de documentos.

Mongo permite hacer **transacciones distribuidas entre N primarios.** Puedo hacer cosas en muchos nodos diferentes, y después committear. Mongo puede tener transacciones y si se manejan bien, no tener problemas de consistencia.

Las **reglas de validación son especificadas para una colección en particular** y a partir de creada serán **ejecutadas para los nuevos documentos modificados o insertado**s.

Restricciones

* No se puede especificar una validación “validator” para colecciones en las Bases de Datos admin, local, y config.
* No se puede especificar una validación “validator” para colecciones system.\*.

Cómo Eludir las validaciones de documentos

* Los usuarios pueden eludir las validaciones de documentos.
* En entornos con control de acceso habilitados, solo pueden eludir las validaciones(bypassDocumentValidation) los roles dbAdmin y restore.

Tipos de Acciones de Validación(validation Action)

* **off-** Deshabilita la validación entera.
* **•warn**- MongoDB loguea toda violación a la regla, pero permite que la inserción y modificación se lleve a cabo.
* **•error**(Default)- MongoDB impedirá toda inserción o modificación que viole a las reglas definidas.

Niveles De Validación(validation Level)

* **•strict**(Default)- MongoDB aplica las reglas de validación para **todos las inserciones o modificaciones.** no te hace nada. STRICT no te deja. Te dice que no podes porque tenés datos que no cumplen
* **•moderate-** aplica las reglas de validación para las i**nserciones o modificaciones a documentos que cumplan con el criterio de validación**. Con este nivel de validación las modificaciones a documentos existentes que No cumplan con el criterio de validación no serán tomados en cuenta para la validación. Entonces si tengo MODERATE, te deja hacer esa restricción, y los datos insertados antes no te hace nada.

Ejemplo1

db.createCollection("facturas",

{ validator: { $or:[ { "cliente.apellido":{ $type:"string"} },

{ "cliente.region" :{ $in:[ "CABA","CENTRO","NEA","NOA"] } } ] },

validationAction:"warn"}

)

dB.getCollectionInfos({name:"facturas"})

devuelve todas las reglas de validación que tenga la colección.

# Consistencia en Cassandra

* Base de datos column family
* **Peer to peer** (No existe el concepto de master o slave)
* ● Distribuida
  + **No posee SPOF** (Single Point Of Failure)
  + **Masivamente y linealmente escalable**
  + Puede escalar hasta miles de nodos con **mejoras casi lineale**s de performance
* Altamente disponible (ubicada en el sector **AP** del teorema CAP)
* Organizada lógicamente en Tablas, filas y columnas
* SQL... perdón, ¡CQL ! (Cassandra Query Language)

Todos los nodos pueden ejecutar cualquier rol. No existe primary, secondary, master, slave. Pero… cómo hace esto? **Es una Base Distribuida de forma NATIVA.** (a diferencia de otras bds que pueden laburar de esta forma, pero configurándola)

Ventajas: No tengo SPOF, todas las componentes son redundantes

**Arquitectura master-slave:** tiene el problema que **podrían fallar los nodos master** y que **queden muchos nodos slave inutilizados sin poder atender pedidos**. Como solución se ponen muchos subconjuntos de master con slave + un router.

**Router:** Le **quita la responsabilidad al usuario de saber a qué servidor dirigirse.** No puede haber un solo router porque SPOF. **Múltiples routers** pueden usarse para lidiar con este problema o para solucionar inconvenientes de performance.

Para evitar esa complejidad adicional y redundancia (costosa), Cassandra toma la postura peer to peer. Voy a poder acceder al nodo que sea y me va a poder atender.

Peer to peer

* No existen masters ni slaves
* **Cualquier nodo puede aceptar request de lectura o escritura**
* Los **nodos “conversan”** entre sí, para saber el **estado de sus vecinos**
* C**ada nodo tiene su propio storage (almacenamiento)**
* El nodo que **recibe la petición de escritura** se llama **coordinador** y es el encargado de **coordinar el request y comunicarse con el nodo correspondiente (N1) para finalizar el request**
* Para repartir la carga de escritura, es normal que los **clientes se comuniquen de manera roundrobin con todos los nodos del cluster,** eligiendo un nuevo coordinador en cada request

**Replication Factor (RF)**

* Cuando agregamos replicación al cluster, incrementamos la responsabilidad del coordinador
* Si deseamos un **nivel alto de consistencia**, el **coordinador** deberá **enviar la información a todos los nodos** que deberían tener el dato recibido
* Básicamente la definición es: **En cuantos nodos voy a escribir el dato**?

**Partición de red**

* **Cada partición de red escribe de manera independiente los datos que le llegan**. Cuando la partición de red **se soluciona l**os nodos vuelven a comunicarse y transmiten la información que cada uno recibió

La información que es recibida por cassandra debe consolidarse luego que una partición de red es solucionada. Este es un problema complejo en sistemas distribuidos en donde deben tomarse algunas concesiones.

Cassandra, **por defecto,** para sus tablas con datos normales **(no contadores)** se **queda siempre con el valor del último timestamp para una columna a la hora de mergear.** Esto puede traer (aunque poco frecuente) datos “corruptos o mal mergeados. Por esto, entre otras cosas, decimos que cassandra es AP

Cassandra decide **sacrificar “potencialmente” la consistencia de sus datos**, para brindar un sistema s**iempre disponible y tolerante a particiones de red** Existen otras maneras de lidiar con este problema

**Conflict-free Replicated Data Types (CRDT)**

Estos **tipos de datos** son **pensados** desde su concepción para **evitar posibles conflictos de mergeo** a la hora de realizar operaciones en sistemas distribuidos.

Decimos que c**assandra es eventualmente consistente.**

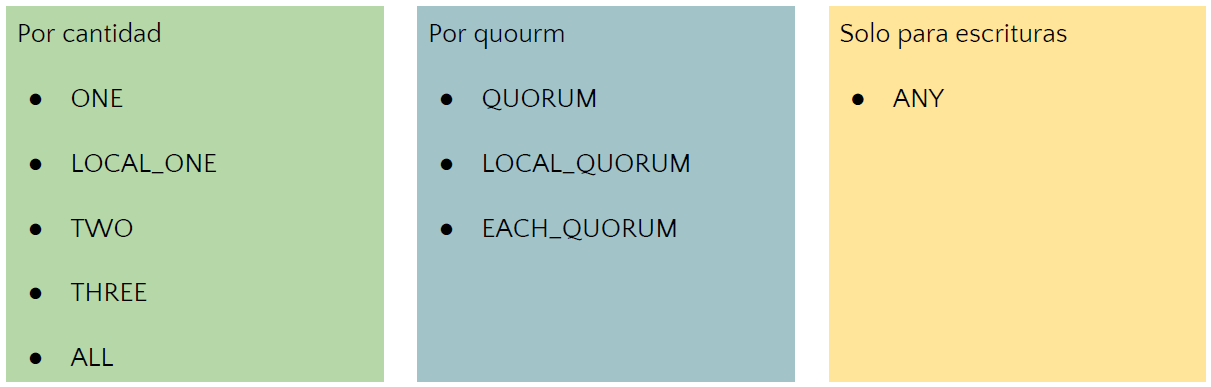
Esto quiere decir que dados eventos de red, actualizaciones concurrentes, o errores de sistema, l**os datos eventualmente convergerán al mismo valor**. Ese valor puede o no ser el correcto si existen conflictos.

Un sistema con consistencia eventual puede también tener **consistencia eventual fuerte** (strong eventual consistency) cuando aplica **CRDT para la resolución de conflictos.**

En sistemas con consistencia eventual fuerte, nodos pueden diferir en sus resultados frente a eventos de red, pero todos los nodos deberán converger en el mismo valor, y **tal valor debe ser el correcto**. Es el caso de counter. Lo que hace internamente es guardar las operaciones. Ejemplo: Si tengo un contador en 6, me llega un request a N1 para incrementarlo y otro request a N8 (que pudo quedar en otra partición). Con un tipo de dato normal, 7 con 7 se pisarían (con el valor más actual). Pero al ser un CRDT es más inteligente, lo que discuten en el nodo son **las operaciones.** Es decir, me guardo el +1 y al mergear los datos actualizo todos los nodos con todas las operaciones quedando en 2 que es el valor correcto :)

En la consistencia eventual **débil, no se garantiza el dato correcto.** Es el caso de los strings.

**Niveles de consistencia**



### One - Write

Con nivel de consistencia **ONE el coordinador es responsable de escribir la información en por lo menos uno de los nodos replica.** Si no pudiese escribir en **ninguno**, la **operación se considera fallida.** Luego de haber escrito en N1, el nodo re**plica su información en los nodos N5 y N6 de manera asincrónica**

### Quorum - Write

Quorum: QUORUM garantiza que la información debe escribirse en por lo menos **N/2 + 1 nodos.**

### Any - Write

En escritura ANY, **no es necesario esperar** a que el coordinador escriba a ninguna réplica para considerar la operación exitosa

### Quorum - Read

**Si en la lectura, los datos no coinciden,** se disparan o**peraciones anti-entropía.**

#### Operaciones anti entropía

* **Read-Repair:**
  + Sucede de **manera probabilístic**a al realizar consultas con ConsistencyLevel.ONE
  + Sucede cada vez que **se encuentran inconsistencia entre múltiples nodos** con cualquier otro ConsistencyLevel
  + Re**para el nodo con el dato más viejo**, pi**sandolo con el más nuevo** al **momento de hacer un read**
* **Repair:**
  + Comando de administración. Debe co**rrerse manualmente en cada nodo**
  + C**ompara los datos del nodo, con los datos de sus réplicas**, quedándose con los datos “correctos”
  + Se recomienda realizar esta operación de manera completa u**na vez por semana y de manera incremental una vez por día**

## Partition Ring

El **set completo de datos de una column family** en cassandra se compone de los hashes entre los números 2a la 63 y -2a la 63

A cada **row-key** se le aplica una **función de hash** que determina a **qué nodo** pertenece.

A esa **row key y todas sus columnas, se las llama partición**

Este espacio de hashes a su vez es dividido en subrangos más pequeños que son particionados en todo el cluster.

Todos los nodos deben **ponerse de acuerdo** sobre qué r**angos de particiones** van a operar.

Para mantenerse informados, los nodos se **comunican mediante gossip**.

### Quorum - Read (**gossip**)

Los nodos se comunican con hasta 3 nodos por segundo comunicándose información interna del cluster:

● Status

● Ip

● Schema version

● Datacenter

● Rack

● Load

● Severity

De **esta manera cualquier nodo puede enterar si, por ejemplo**

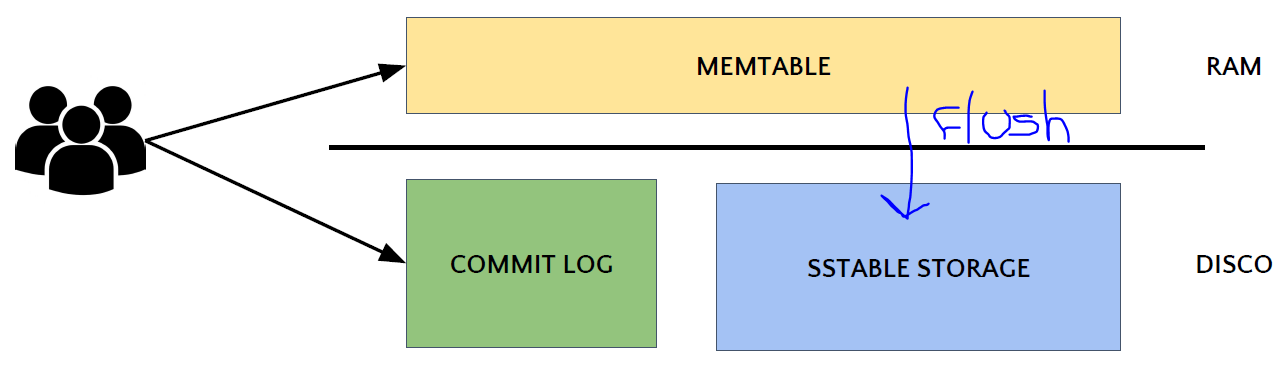
**● Se añade un nuevo nodo al cluster**

**● Se cambia el schema de las tablas**

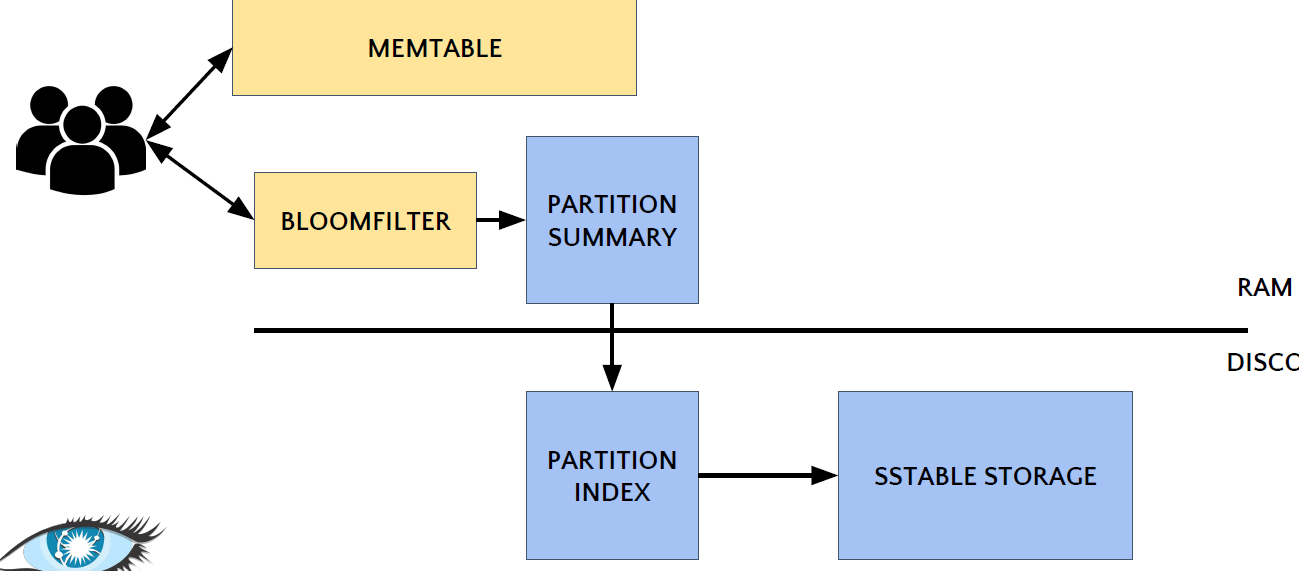
**● Se cae un nodo**

Es un protocolo bastante pandémico.

### Write path



* Los datos se guardan **ordenados** en la memtable por el **criterio de almacenamiento de los clustering columns**. Físicamente ordenados.
* Cuando llega un insert se graba tanto en la memtable como en el commit log.
* **El commit log es una estructura append only**. Los datos **nuevos se escriben al final del archiv**o. Frente a una catástrofe, **minimizo el riesgo de que quede corrupto**. En todo caso se borra el final. No se pueden actualizar datos viejos del log.
* Flush: **bajar los datos de memoria a disco.**
* Las **sstables** son estructuras i**ndexadas inmutables**. Son escritas por única vez mediante un flush y solo son reescritas en el proceso de compactación



Se **compacta y se crea una sstable con los datos más nuevos.**

No va a quedar guardada la historia porque se compacta.

Las **escrituras en cassandra son baratas**, pero las lecturas no tanto. Conviene escribir demás para leer menos.

El bloomfilter puede tener **falsos positivos.** Puede decir que existe en la partición o que puede existir.

Voy a leer la partition key 14

Bloom filter

→ NO existe (y listo)

→ **Puede existir**

Entra al **partition summary**, entra en este rango: usá este partition index

En este p**artition index**, entrá a tal posición de memoria en la S**STABLE**

Cuando leo tengo que leer de dos lados: **memtable si el dato es nuevo, y sino disco si ya se flusheo.**

Voy a buscar el dato a la memtable, si no esta, voy al patition summary, que me dice en que rango del partition index buscar, que me indica en que posicion de la sstable esta.

#### Replicación multi datacenter

Cada nodo sabe dentro de qué datacenter está y dentro de qué rack. Cuando replica, elige 3 datacenters lo más alejado posible.

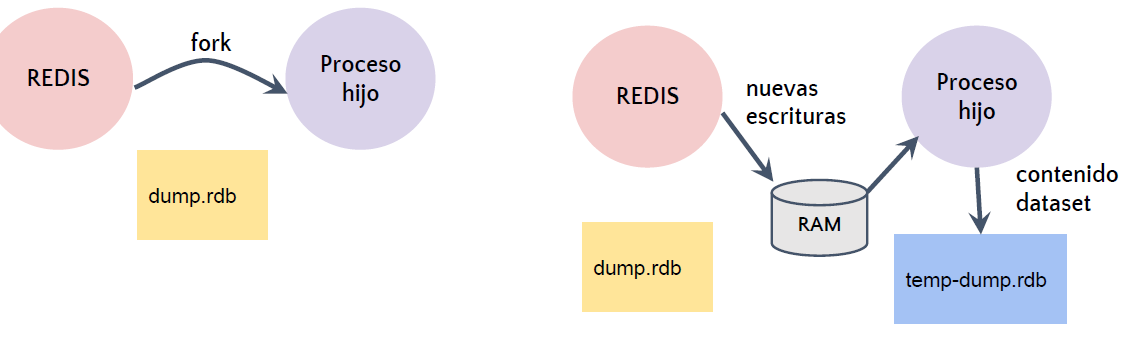
#### Snitches

Le permiten a cassandra entender la topología de la red

# Replicación y Sharding en Redis

* **RDB:** Se realizan **snapshots de los datos cada ciertos intervalos** y se almacenan en un **archivo de tipo .rdb** (Snapshotting). Es el e**stado actual en un momento de la DB.**
* **AOF:** Cada o**peración de escritura que se realiza se almacena en logs.** Los **cambios se aplican al inicio del server, reconstruyendo los datos originales**. Los comandos se loguean utilizando el protocolo Redis (Journaling). AOF es un archivo **append only** para restaurar el estado de la DB. Guarda todas las **operaciones y** cuando se levanta se ejecutan. Se va **reescribiendo para poder recuperar el estado con las menores operaciones posibles.**
* Se pueden combinar RDB y AOF en una misma instancia.
* La persistencia se puede deshabilitar en su totalidad.
* Además de **definir la frecuenci**a de actualización del .rdb en el archivo de configuración se puede realizar e**xplícitamente con SAVE (sincrónico) o BGSAVE (en background).**

### RDB



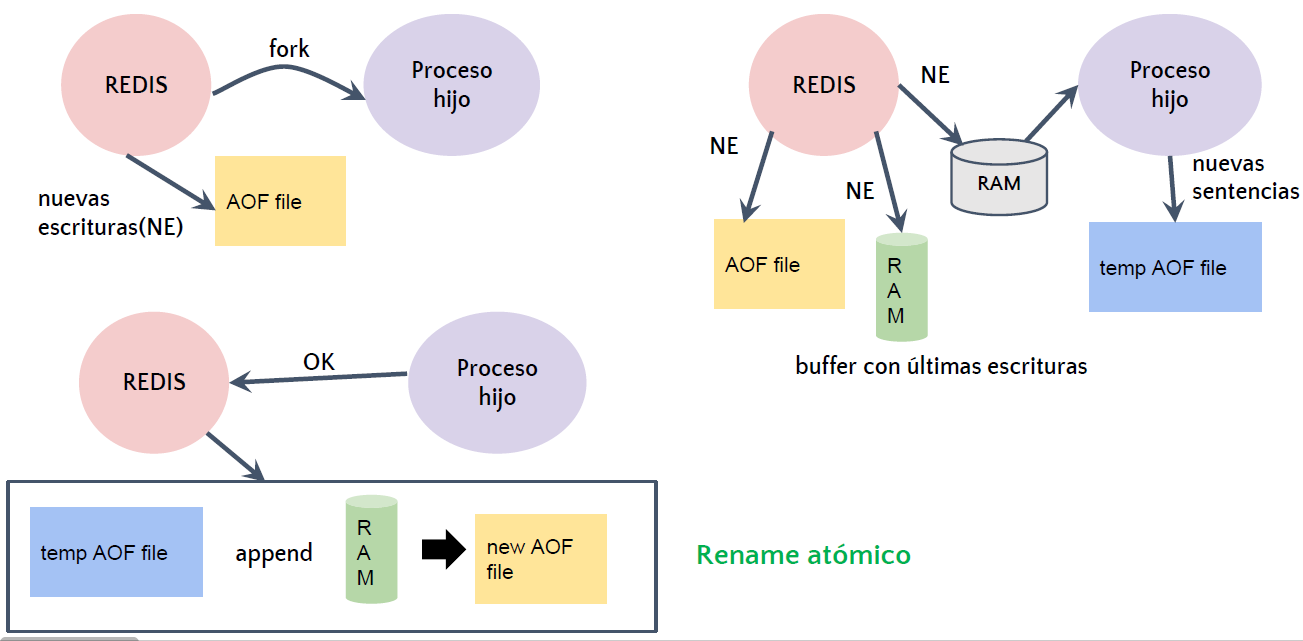
VENTAJAS

* El snapshot de los datos se almacena en un ú**nico archivo**, por lo que se puede u**tilizar para backup**s → útil para disaster recovery (puede ser fácilmente transferido a distintos datacenters)
* Se puede cambiar la frecuencia con que se guardan los snapshots. Se puede **restaurar fácilmente diferentes versiones de los datos**.
* Maximiza la performance: el proceso **padre no debe realizar ningún tipo de operación de disco** (un **hijo** se encarga de lo mismo).
* Permite que las **instancias con grandes datasets se inicien más rápido** (comparado con AOF).

DESVENTAJAS

* **No es bueno para minimizar pérdidas de datos** en caso de que una instancia se caiga.
* Necesita hacer un **fork() para persistir datos en discos** través de un proceso hijo. Esto **puede consumir mucho tiempo** si el dataset es muy grande, afectando la atención de peticiones de clientes. En AOF, en cambio, se puede configurar la frecuencia en la cual se reescriben los logs sin afectar la durabilidad de los datos.

### AOF



VENTAJAS

* Se puede configurar la política de actualización: sin actualización, actualización en cada segundo (default) o en cada query.
* Se utiliza **log** sobre el cual se realizan o**peraciones únicamente de append.**
* Redis reescribe el AOF cuando el mismo crece demasiado. Se **crea un nuevo AOF con la cantidad mínima de operaciones para generar el dataset actual.**

DESVENTAJAS

* Por lo general los archivos **AOF son más grandes** que los generados por RDB.
* **Puede ser más lento** a RDB dependiendo de la política de actualización que se utilice.
* En el pasado se detectaron ciertos bugs raros con algunos comandos específicos, causando que el AOF generado no reproduzca exactamente el dataset original.
* **AOF trabaja actualizando incrementalmente el estado actual mientras RDB crea todo desde cero una y otra vez** (lo cual es más robusto).

### Cuál usar?

* Si se requiere un grado de **seguridad** comparable a la de una BD relacional →  **RDB + AOF**
* Si se pueden tolerar **pérdidas de datos de unos pocos minutos → RDB**
* Si se quiere **minimizar la probabilidad de pérdida de datos** → **AOF (+ RDB ) (**realizar **un snapshot de vez en cuando es una buena idea para realizar fácilmente backups de la BD**)
  + No se recomienda utilizar únicamente AOF
  + Si se tienen a**mbas técnicas simultáneamente** al iniciar se utilizará el archivo AOF en lugar del RDB para recuperar los datos.

## Replicación

* La replicación es de tipo MASTER -SLAVE
* La a**ctualización de los slaves es asincrónica**
* La r**eplicación es no bloqueante** tanto del lado del master como del slave (pueden seguir atendiendo pedidos)
* Se puede utilizar para
  + **Escalamiento horizontal**
  + **Redundancia de datos**
  + **Ahorrar al master** la operación costosa de **escribir el dataset a disco** (se configura el master sin persistencia..en este caso no deberían reiniciar automáticamente el master sin que se sincrocnicen las réplicas).

Pasos cuando se settea un slave

1. Al setear una i**nstancia como un slave**, un comando **SYNC se envía al master**
2. Al recibir un SYNC, el **master comienza a guardar en background los datos**, mientras a**lmacena en un buffer los nuevos comandos de escritura que llega**n.
3. A**l finalizar, el master envía al slave el archivo**. **Este último lo guarda en disco y luego lo carga a memoria.**
4. El master luego le **envía al slave los comandos almacenados en el buffer** (utilizando Redis Protocol)

* Si el master recibe **múltiples peticiones de SYNC,** guarda el dataset u**na única vez y se la envía a todos.**
* La sincronización de los datos de un master a un slave puede ser:

○Full Resync

○Sincronización Parcial (a partir de 2.8)

REPLICACIÓN PARCIAL

* Permite que la r**eplicación se continúe sin requerir una sincronización completa** (PSYNC)
* Del lado del master se crea un **backlog del flujo de replicación en memoria.**
* El **master y el slave acuerdan el offset de replicación** y u**n master run id**
* Cuando el **link entre los mismos se cae**, luego el s**lave se reconecta** y pide que se continúe la replicación:
  + Si el **master run id es el mismo** y si el o**ffset indicado está disponible en el backlog de replicación** → la **replicación se continúa**
  + Si el **master id es distinto, o si el offset no está disponible** → se realiza una s**incronización completa.**
  + El master run id no se persiste en disco, por lo que cuando un **slave reinicia siempre debe realizar una sincronización complet**a

READ-ONLY slaves

* Se puede configurar el modo read-only para los slaves, de forma que las e**scrituras que lleguen a los mismos por error sean rechazadas.**

CONSISTENCIA “RELAJADA”

El master se puede configurar para que **sólo acepte escrituras si al menos N slaves están “actualmente” conectados al mismo.**

Los s**laves pinguean al master a cada segundo** indicando l**a cantidad de datos de replicación procesados.**

Los **masters guardan el timestamp del último ping de cada slave**

Se puede configurar el mínimo número N de slaves que tienen menos de M segundos de lag. No garantiza que los datos se vayan a escribir, pero minimiza la pérdida de datos.

## Sentinel

Redis Sentinel es un sistema diseñado para ayudar a **administrar instancias Redis.** Se conectan con los nodos:

* M**onitoring**: Continuamente chequea que todas las i**nstancias estén OK.**
* **Notification**: Puede notificar al administrador, o a un programa si es que algo está mal con las instancias monitoreadas.
* **Automatic failover:** Si un **master no está funcionando** correctamente, inicia un proceso de failover en el que un **slave se transforma en master y** el resto de los s**laves se reconfiguran para replicar del mismo**. A su vez, las aplicaciones que utilizan Redis son notificadas de la nueva dirección que deben usar en sus conexiones.
* **Configuration provider**: Los **clientes** se conectan a Sentinel para obtener la **dirección actual del master de Redis responsable de un determinado servicio.**
* Es un sistema distribuido → por lo general hay múltiples procesos Sentinel en una infraestructura Redis.
* Los procesos Sentinel utilizan g**ossip protocolo** para **determinar si un master está caído** y protocolos para ponerse de acuerdo de forma que un proceso esté autorizado para realizar un proceso de failover y asigne una nueva versión a la configuración.
* Un cluster de procesos Sentinel pueden realizar un proceso de failover sobre un master aún si algunos de los procesos están fallando.
* Un proceso sentinel **que no funciona bie**n no **podrá realizar un failover sin la autorización del resto de los procesos.**
* Los clientes se pueden conectar a cualquier Sentinel para obtener la configuración del master.
* Es OBLIGATORIO utilizar archivo de configuración
* En sentinel.conf se especifica la configuración del master. Luego el mismo será completado por Sentinel con información de los slaves.
* Luego de cierto **tiempo sin responder (**down-after-milliseconds) se puede considerar que la **instancia está SDOWN** (subjetivamente caída)
* SI una c**antidad suficiente (quórum) de Sentine**ls cree que la misma está SDOWN, entonces la in**stancia se marca como ODOWN** (objetivamente caída), lo cual dispara un proceso de recuperación.
* El **nodo q**ue quiere **iniciar el proceso de recuperación** debe pedir **autorización de la mayoría de los procesos Sentinel para poder realizarlo.**
* Si un proceso votó a otro proceso Sentinel para recuperar un determinado master, esperará cierto tiempo (failover-timeout ) antes de querer realizar la recuperación del mismo.

### Selección de nuevo master

Se evalúa sobre los slaves:

1. **Tiempo de desconexión al maste**r: menor a (down-after-milliseconds \* 10) + milliseconds\_since\_master\_is\_in\_SDOWN\_state).
2. **Prioridad**. (a menor valor, más prioridad)
3. **El offset de replicación procesado.** (si la prioridad es = se elige al que haya recibido más datos del master)
4. **Run ID.** (si tanto la prioridad y el offset son = se elige el que tenga un run ID lexicográficamente menor)

Una instancia de Redis puede tener slave-priority zero para indicar que nunca debe ser elegida como master

## Particionamiento

* **Client side partitioning:** El cliente directamente **selecciona el nodo** específico en el cual debe leer o escribir una key en particular. Muchos clientes de Redis implementan esta clase de particionamiento.
* **Proxy assisted partitioning:** Los clientes envían sus requests a un proxy que **se comunica a la instancia adecuada con protocolo Redis y luego envía la respuesta al cliente**. Twemproxy implementa este tipo de particionamiento.
* **Query routing:** Se puede enviar **cualquier consulta a cualquier instancia ya que la misma se encargará de enviarla al nodo indicado**. Redis Cluster implementa este tipo de particionamiento pero con ayuda de un cliente, ya que ya el mismo debe redireccionar la petición al nodo indicado.

### Problemas

**Operaciones con múltiples keys por lo general no son soportadas.**

**No se puede utilizar transacciones que involucren múltiples keys.**

**El agregado es el par key-value**, por lo que no sirve si nuestro dataset está formado por una o pocas keys con values muy grandes (**no podemos distribuir el value**).

El manejo de los datos es más complejo. Hay que manejar múltiples archivos RDB/AOF y para generar un backup de los datos es necesario combinar múltiples archivos de persistencia.

Modificar la capacidad (a**gregar eliminar instancias) puede ser complicado.**

Redis Cluster soporta el rebalanceo de datos prácticamente en forma transparente.

**Redis com**o **cache**:

si un n**odo no está disponible se puede utilizar otro sin problemas**, es decir, se puede **modificar el mapa de keys que posee una instancia con tal de aumentar la disponibilidad de los dato**s. Maneja un set de hash contenidos en instancias.

**Redis como data store** → una **key debe siempre estar mapeada a una misma instancia de Redis**. Por lo tanto, el **número de nodos es fijo y no puede variar.** No puedo redistribuir en una instancia nueva.

**Pre-sharding**: Como Redis es muy liviano, una “solución” consiste en **iniciar muchas instancias desde el principio,** por más que no se utilicen ( por ejemplo, 32 -64 instancias) Cuando los d**atos crecen y se necesitan más instancias**, lo único que hay que hacer es **mover las mismas de un server a otro.**

## Redis cluster

Permite **distribución automática** a través de múltiples nodos Redis.

Provee cierto grado de disponibilidad durante particiones de red.Sin embargo, si la mayoría de los masters no están disponibles, el cluster dejará de funcionar.

* C**ada key forma conceptualmente parte de un hash slot.**
* Hay 16384 hash slots en un Redis Cluster, y para saber cuál es el hash slot para una key se realiza: CRC16 de la misma y luego MOD 16384.
* Cada **nodo en el cluster es responsable de un subset de hash slots.**
* Como la operación de mover hash slots entre nodos no requiere parar las operaciones, se pueden a**gregar y remover nodos sin necesidad de downtime**.
* Soporta o**peraciones sobre múltiples keys** mientras que t**odas las keys involucradas** en la ejecución de un comando pertenezcan al **mismo hash slot.**
* Se puede **forzar para que ciertas keys pertenezcan al mismo hash slot** → utilizand**o hash tags (s**ólo realiza el hash sobre el tag)
  + key1 → user:100:comments{user100} // key2 → user:100:posts{user100}.
* Utiliza replicación, **cada hash slot tiene entre una (master) hasta N (n-1 slaves) réplicas**
* Por default la **replicación de las escrituras es asincrónica,** por lo que puede sufrir de pérdidas de datos, pero también puede ser configurada en forma sincrónica si es que es requerido.

# Consistencia en Neo4j

* Dado que las bases de datos de grafos están basadas en nodos conectados, la mayoría de ellas generalmente no **soportan la distribución de los nodos en diferentes servidores.**
* Neo4j es Full ACID-Compliant.
* Cuando se ejecuta Neo4j en un clúster. El mismo puede estar compuesto de **un** **servidor maestro y varios esclavo**s. Una escritura en el servidor maestro se sincroniza con los esclavos.
* Se **permite la escritura en los esclavos,** la misma **se sincroniza inmediatamente con el maestr**o; Con los otros esclavos no se sincronizará inmediatamente, van a tener que esperar a que los datos que **se propaguen desde el maestro hacia ellos**.
* Las Bases de datos de grafos aseguran la **consistencia a través de transacciones**.
* **No se permite dejar relaciones colgadas:**
* El nodo inicial y el nodo final siempre tienen que existir.
* Los nodos sólo se **pueden eliminar si no tienen ninguna relación unida a ellos.** Primero se debe eliminar la relación.

## Transacciones

* Todas las **operaciones que acceden al grafo deben realizarse en el contexto de una transacción.**
* El nivel de aislamiento por defecto es READ\_COMMITTED.
  + Una o**peración de lectura leerá el último valor confirmado**, a menos que **exista una modificación local** dentro de la transacción en curso.
  + Las **lecturas no bloquean,** por eso p**uede haber lecturas no repetibles** (non-repeatable reads).
* La información devuelta por barridos no está protegida de ser modificada por otras transacciones.
* Se pueden adquirir bloqueos manualmente para incrementar el nivel de aislamiento.
* Los **bloqueos son a nivel de nodo y relación.**
* Al **agregar, modificar o eliminar una propiedad** en alguno de ellos, queda b**loqueado el nodo o la relación específica.**
* Al **crear o borrar un nodo queda bloqueado ese nodo**.
* Al **crear o borrar una relación queda bloqueada la relación y sus ambos nodos.**
* Es importante **finalizar las transacciones para que se liberen los bloqueos** o la memoria adquirida, ya que todas las modificaciones de la transacción se mantienen en memoria.
* El g**estor de transacciones detecta los deadlocks antes** de que ocurran, y arroja una excepción. Se elimina la transacción.
* La**s transacciones con deadlock se marcan para rollback.**
* Experimentar deadlocks frecuentemente es una indicación de que existen pedidos concurrentes de escritura, que ocurren de forma tal que no es posible ejecutarlos manteniendo el aislamiento y consistencia deseados.
* Una solución es asegurarse de que las actualizaciones concurrentes ocurren bajo un cierto criterio.
* Neo está pensado para tener **muchas consultas pero que las actualizaciones se hagan en batch.**
* Cuando se **borra un nodo o una relación, todas sus propiedades se eliminan** automáticamente.
* Un nodo eliminado no puede tener asociada ninguna relación cuando la transacción se valide.
* Es por eso que Neo4j fuerza una restricción al validar la transacción de que todas las relaciones deben tener un nodo de inicio y uno de fin válidos.

## Tipos de respaldo

* **Backup online** (respaldo en frío): Hay **archivos de backup**. Cuando falla un master, **otra instancia se transforma en master con esos archivos**.
* Backup online con alta disponibilidad (respaldo en caliente): Siempre hay **un master y una instancia de backup que se va actualizando con la información del mast**er y está lista en caso de que falle el master.
* Cluster de alta disponibilidad: cuando falla un **master, un slave se convierte en master.**

## Recuperación ante desastres

* **Prevención**
  + **Realizar Back**up De alta disponibilidad en otras ubicaciones fuera del data center actual.
  + Realizar backup online a diferentes ubicaciones en el file system.
  + Neo4j High Availability Cluster
* **Detección**
  + Monitoreo
* C**orrección**
  + Online backup: se inicia un nuevo servidor Neo4j a partir del backup y va tomando los nuevos requerimientos.
  + Neo4j High Availability Cluster: **un esclavo caído se puede reinsertar al cluster, obteniendo las últimas actualizaciones del master.**

## Alta disponibilidad

* Siempre hay **un solo maestro y cero o más esclavos.**
* Neo4j HA puede manejar **peticiones de escritura en todos los nodos,** no necesita direccionarlos específicamente al maestro. Un esclavo maneja las escrituras sincronizandose con el maestro para preservar la consistencia.
* Una escritura en el maestro intentará r**eplicarse a los esclavos antes de que se complete la transacció**n, pero **si falla, la transacción aún será considerada exitosa.**
* Todas las **modificaciones se propagarán del maestro a todos los esclavo**s eventualmente, así que pueden no ser inmediatamente visible a los otros esclavos.
* Cada **instancia monitorea a todas las demá**s para saber cuáles se unen o abandonan el cluster.
* Las **instancias eligen una para que sea el maestro y designan a las otras como esclavos.**
* Al configurar, se especifican las instancias que formarán el cluster. Al iniciarse cada una de ellas, si puede contactar a las demás formará el cluster, uniéndose como esclavo, de lo contrario se iniciará como el maestro de un propio cluster.
* Escritura en esclavos
  + Al realizar una **escritura en un esclavo, cada operación se sincronizará con el maestro** (y se adquirirán bloqueos tanto en el maestro como en el esclavo).
  + Al finalizar, **primero validará los cambios en el maestro y luego, si funcionó bien, en el esclavo.**
  + Para asegurar la consistencia, el **esclavo debe estar actualizado respecto del maestro** antes de realizar una operación de escritura.
* Escritura en el maestro
  + Se ejecutan de forma idéntica a la ejecución en el modo SIN alta disponibilidad.
  + Al fin**alizar con éxit**o, la transacción se **replicará hacia un número configurable de esclavo**s (uno por defecto).
  + Si esta réplica falla, de todos modos la transacción se considera que fue exitosa.
  + Se podría configurar el “pushfactor” en 0 para mayor rendimiento de escritura, aunque esto incrementaría el riesgo de perder transacciones no replicadas si se cae el maestro. Podría mandar al secondary antes de bajarlo a disco y se me cae todo.
  + También se pueden **configurar los esclavos para recuperar asincrónicamente los cambios.**
* **Falla en una instancia**
  + Las o**tras instancias en el cluster lo detectan y la marcan como caída** temporalmente.
  + Cuando la instancia caída se **recuper**a, automáticamente **se pone al día** con el cluster.
  + Si la caída fue del master, **se elegirá otro miembro para que pase de ser esclavo a maestro** una vez que se haya alcanzado el **quórum.** Entonces, el **nuevo maestro publicará su disponibilidad a los otros miembros. Las transacciones no validadas se deshacen.**
  + Hasta que se elige el nuevo maestro**, no se realizarán escrituras.**
* Árbitros
  + El cluster puede tener s**olamente un maestro, un esclavo y un tercer componente que sólo sea un árbitr**o.
  + El árbitro participará en la e**lección del nuevo master r**ompiendo los empates en caso de caída.
* Balanceador de carga
  + El frontend del cluster suele ser un servidor **proxy que realiza el balanceo de la carga,** y que puede distinguir transacciones de lectura y de escritura.
  + Este servidor puede realizar funciones de **“sharding”** cuando el volumen de datos es muy grande para caber en memoria, entonces almacena **temporalmente diferentes partes del grafo en cada máquina.**

## Escalamiento

* Sharding: Es muy difícil de implementar en las bases de datos orientadas a grafos, dado que estas **BD no están orientadas a agregados, sino que están orientadas a las relaciones.**
* Dado que cualquier nodo dado puede estar relacionado con cualquier otro nodo, el **almacenamiento de los nodos relacionados en el mismo servidor es mejor para el recorrido del grafo.**
* Recorrer un grafo cuando los nodos están en diferentes máquinas no es bueno para la performance.
* Existen varias formas de escalar una base de datos orientada a Grafos.
  + **Agregar suficiente memoria RAM al servido**r para que todo el esquema de BD con sus nodos y relaciones sea contenido en ella. Esta opción es útil si la BD utilizada tiene un tamaño posible para ser contenido en memoria RAM.
  + Podemos **mejorar las lecturas d**e la base de datos añadiendo **más servidores esclavos con acceso de sólo lectura** a los datos, con todas las escrituras en el servidor maestro.
  + Cuando el tamaño de la bd hace que la replicación sea poco práctica, podemos **fragmentar los datos desde el lado de la aplicación conociendo el dominio.**

# Map Reduce

* MapReduce es un modelo de programación y su implementación asociada para procesar y generar grandes sets de datos.
* La idea es que tengo que resolver una consulta teniendo en cuenta datos que se encuentran en distintas particiones. Entonces se hacen subcálculos y después se agrupan.
* Modelo de programación
  + Tiene abstracciones para expresar cálculos simples
* Librería
  + Se ocupa del trabajo pesado: Paralelismo, Fault Tolerance, Data Distribution y Load Balancing

## Operaciones básicas

**Map**

**Produce una lista de pares (claves, valor) desde una entrada estructurada como un par (key,value) de un tipo diferente**

MAP(agregado) { (k1,v1);(k2,v2);…;(kn,vn)}

**Reduce**

**Produce una lista de valores de una entrada que consiste en una clave y una lista de valores asociados con esa clave(ki, list(vi)) list(vi)**

## Librería

* Ag**rega todos los valores intermedios asociados con la misma clave intermedia**
* passes the intermediate key-value pairs to reduce function

### Map

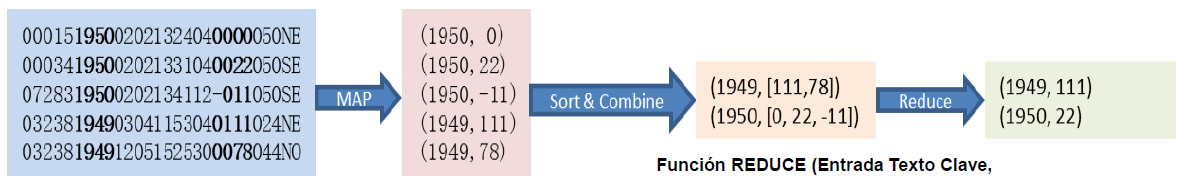
Se obtienen sólo los datos que se necesitan y los expresa como (key, value).

### Sort & combine

Ordena los datos y los deja en un formato más fácil de analizar. Es una **función de runtime.**

### Reduce

Realiza el cálculo final sobre los datos del sort&combine. Es una función **definida por el usuario.**



El **usuario especifica:**

**M: nro. de tareas map**

**R: nro. de tareas reduce**

FaseMap

La entrada se particiona en M splits

Las tareas map se distribuyen en múltiples máquinas

FaseReduce

Las tareas reduce se distribuye entre múltiples máquinas

Las claves intermedias se particiona para ser procesadas por la tarea reduce deseada

## Ambiente de ejecución

**No puede comenzar ninguna función reduce hasta que el Map esté completo**

Las tareas se programan en base a la ubicación de los datos.

Si un proceso map falla en cualquier momento antes de que termine el reduce, entonces la tarea debe correrse completamente de Nuevo.

El master debe comunicar ubicaciones de archivos intermedios

2. **Una De las copias del programa es especial –el maestro** -. El resto de los workers que son asignados trabajan según define el maestro. Hay M tareas map y R tareas reduce para asignar. **El maestro toma workers idle y asigna cada uno a una tarea map o una tarea reduce.**

3. Un worker al que se le asigna una tarea map lee los contenidos del split de entrada. **Parsea los pares clave/valor del ingreso de datos** y pasa cada par a la **función Map definida** por el usuario. Los **pares key/value intermedios producidos por la función Map se almacenan en un buffer en memoria.**

4. Periódicamente, los pares en **buffers se escribe a disco local**. Las ubicaciones de estos pares en el buffer en disco local se **pasan al proceso Master**, que es responsable de **reenviar esas ubicaciones a los workers reduce.**

5. Cuando Un **worker reduce es notificado por el maestro acerca de estas ubicaciones**, usa RPCs para leer los datos del buffer de disco local que dejaron los procesos Map. Cuando **Un worker reduce tiene todos los datos intermedios, los ordena por las claves intermedias (sort & combine)** de forma tal que todas las **ocurrencias de la misma clave se agrupan juntas**. El ordenamiento es necesario porque típicamente muchas claves diferentes se mapea a la misma tarea reduce. Si la cantidad de datos intermedios demasiado grande para entrar en memoria, se usa en sort externo.

6. El worke**r reduce itera sobre las claves intermedias ordenadas para cada clave única intermedia encontrad**a p**asa la clave y el correspondiente conjunto de valores intermedios a la función del usuario Reduce.** La **salida** de la función Reduce es **agregada al archivo salida final para esta partición de reducción.**

7. Cuando todas las tareas map y reduce se completan, el programa del usuario despierta al programa del usuario. En este punto, la llamada MapReduce del programa del usuario retorna al código del usuario.

Conclusión. Map reduce:

Modelo De programación de fácil uso y escalable para procesar grandes volúmenes de datos en clusters:

* P**ermiten a los usuarios focalizarse en los cálculos**

Oculta Cuestiones De:

* **paralelismo, fault tolerance, data partitioning y load balancing**

Lograr Eficiencia A través del uso de discos locales

# 

# HDFS

## Hadoop

* ●Hadoop Distributed File System
  + **File System Escrito en jav**a, basado en GFS (Google File System)
  + Puede montarse sobre casi cualquier file system(ext3, ext4, ntfs, fat32)
* Pensado para:
  + **Archivos grandes y volumen de datos** (órdenes de Terabytes)
  + **Trabajar con un esquema de Writeonce / readmanytimes.**
  + Los contenidos de los archivos no pueden modificarse. Solo **agregar datos al final de un archivo**
  + La latencia para recuperar un archivo entero es mucho más importante que para obtener una línea de un archivo

HDFS está diseñado para trabajar con un n**úmero modesto de archivos extremadamente grandes**

## Almacenamiento de datos

* Cuando se almacena un archivo en HDFS lo **parte en un conjunto de bloques individuales y almacena esos bloques en nodos esclavos**
* HDFS no sabe que está almacenado dentro del archivo, por lo que lo parte por el tamaño del bloque, sin ninguna regla que sea entendible por humanos
* Existe un tamaño de bloque predefinido para la instancia Hadoop. **128mb.** no todos los bloques tienen el mismo tamaño. El último es de 1MB.
* La falla es la norma y no la excepción. La **detección de fallas y la recuperación rápida es uno de los objetivos de Hadoop.**

### Namenode

**Nodo maestro d**e la operación con HDFS. Es el encargado d**e almacenar toda la metadata para los nodos del cluster: Qué bloques componen qué archivos y cual es la estructura de directorios del Filesystem** son las preguntas que este nodo debe responder.

Pueden crearse s**econdary namenodes:**

* Almacenan **periódicamente checkpoints de la metadata del cluster**
* Sirven para no sobrecargar al namenode en la creación de archivos en Local FS de potencialmente GBs de tamaño
  + Los secondary namenode **NO PUEDEN TOMAR** EL LUGAR DE UN NAMENODE.
  + Ante una caída irreversible del namenode un administrador debe
    - **Levantar un nuevo namenode**
    - **Hacer un -recovery con los checkpoints** del secondary
    - **Reconfigurar todos los data nodes para apuntar al nuevo namenod**e

En **Hadoop 2 se puede configurar un segundo namenod**e

### Datanode

**Nodos workers del cluster.** Los bloques de datos son almacenados y servidos por estos nodos. Periódicamente deben reportar al namenode la lista de los bloques que contienen.

No es posible utilizar el File System sin el namenode. Si el Namenode explotara, sería imposible reconstruir archivos desde los datanodes.

* También se los denomina **Slave Nodes**
* Ejecutan un **proceso en background denominado DataNod**e
* Este DataNode tiene información de las porciones de datos que el sistema almacena en ese nodo.
* **“Habla” regularmente** con el Master Server del sistema HDFS (llamado NameNode) para **reportar la salu**d y estado de los datos almacenados localmente.
* Los data blocks se almacenan como raw files en el file system local.
* Los **DataNode no saben que hay dentro de los bloques** de datos que almacenan
* El servidor **NameNode es el que tiene el mapeo** de cuales bloques de datos están almacenados en HDFS.
* HDFS es un file system montado sobre otro. Es importante optimizar el file system local para trabajar con HDFS
* Ext3 y Ext4 son los FS más comunes (no usar Linux LogicalVolumeManager –LVM).
* Cuando un usuario almacena un archivo en HDFS, el archivo se divide en bloques y 3 copias de los bloques se almacenan en nodos slave en el cluster.
* El N**ameNode sabe no sólo que bloques componen archivos individuales**, sino **donde están ubicados esos nodos y sus réplicas.**
* Si **NameNode está caído, las aplicaciones no pueden acceder los datos en HDFS.**
* NameNode corre en un servidor Master Node
* Toda la información de m**apeo entre los data blocks y los archivo**s correspondientes está almacenada en un archivo llamado **Fsimage.**
* Todos los cambios en los datos se registran en un edit journal que trackea los eventos desde el último checkpoint.
* Un **checkpoint ss** el **último momento cuando el editlog fue mezclado con fsimage**.

**Escritura de datos**

**1.El cliente envía un requerimiento al NameNode para crear un nuevo archivo**

El NameNode determina **cuántos bloques se** van a necesitar y le da al **cliente un “permiso” para crear estos nuevos bloques en el cluste**r. El cliente tiene un tiempo límite para usar ese permiso. Esto asegura que el espacio de almacenamiento no es tomado por aplicaciones clientes que están fallando.

2.**El cliente escribe las primeras copias de los bloques a los DataNode usando el “préstamo” otorgado por el NameNode**

-El NameNode administra los requerimientos de escritura y determina donde los bloques de archivos y sus réplicas se van a escribir, balanceando disponibilidad y performance.

-La primer copia del bloque se escribe en un rack, y la segunda y tercera copias se escriben en un rack diferente del primero, pero en DataNode diferentes del mismo rack.

-Esto minimiza tráfico de red, asegurando que no existan bloques de datos en el mismo punto de falla

3**.Mientras cada bloque se escribe en HDFS, un proceso especial escribe las réplicas restantes a los otros DataNode identificados por el NameNode**

4.Luego de que los daemons DataNod**e informan que se crearon las réplicas de los** bloques, la aplicación cliente cierra el archivo y notifica eso al NameNode, el cual cierra el préstamo.

* La ubicación de las réplicas saben sobre los racks de la instalación.
* En general en estas instalaciones hay muchos racks, y la comunicación entre racks se hace a través de switches.
* El ancho de banda de red entre máquinas del mismo rack es más grande que en racks diferentes.
* NameNode sabe cuál es el rack id de cada DataNode
* Las réplicas se ubican:
  + 1 en un DataNode del rack local
  + 1 en un DataNode de otro rack
  + 1 en un DataNode diferente del otro rack

1/3 de las réplicas están en el rack

2/3 de las réplicas están en otro rack

**Lectura de datos**

1.El cliente envía un **requerimiento al NameNode para leer un archivo**

-El NameNode determina qu**é bloques están involucrados y elige**, basado en la proximidad de los bloques uno con el otro y hacia el cliente, **el método de acceso más eficiente.**

2.El cliente accede a los bloques usando las direcciones dadas por el NameNode.

**Balanceo de datos**

* Con el tiempo, los patrones no uniformes de grabación de datos (donde algunos nodos esclavos pueden tener más datos escritos a ellos) o fallas de nodos, los **datos pueden estar distribuidos en forma no pareja e**ntre los racks y los nodos slave del cluster.
* Esto p**uede generar problemas de performance:** nodos con pocos datos pueden no usarse, mientras que los que tienen muchos datos pueden usarse mucho.
* HDFS incluye una **utilidad llamada balancer que redistribuye los bloques de DataNode sobreutilizados a subutilizados.**
* Esto se hace manteniendo la política de poner bloques en DataNode diferentes y racks.
* La utilización de esta herramienta es hecha por el administrador HDFS que chequea regularmente la salud del sistema.

## YARN

entorno de gestión de recursos y aplicaciones distribuidas donde se pueden implementar múltiples aplicaciones de procesamiento de datos totalmente personalizadas y específicas para realizar una tarea en cuestión.

### ResourceManager

En YARN, este componente es principalmente un planificador (scheduler) de resource requests y un ApplicationManager. Las aplicaciones pueden solicitar diversos recursos del cluster los que deben pasar por un proceso de validación de permisos, y luego son concedidos y priorizados de acuerdo a las políticas de planificación. Un resourcemanager concede container a través del applicationmanager y los aloca en un nodemanager específico.

### NodeManager

Procesos ejecutándose en cada nodo (datanodes), a cargo de llevar control de que los containers no consuman más recursos de los solicitados, informando al resourcemanager periódicamente. Envían heartbeats al resourcemanager para ser elegibles de alocación de container y dan la visión general de recursos del sistema.

### ApplicationMaster

Un ApplicationMaster es una instancia de una librería de terceros ejecutándose en el cluster. Es el responsable de negociar recursos con el resourcemanager y luego trabajar en conjunto con el nodemanager para monitorear los recursos de sus procesos. Es el responsable de controlar el failover, status y progreso de los procesos que ejecuta.

### Container

Abstracción lógica de recursos de sistema. Es el derechoa usar una cierta cantidad de recursos de una máquina específica (NodeManager).

### Tipos de Scheduler

Existen 3 tipos de schedulers: Fair, Capacity y FIFO.

* FIFO Scheduler: Mismo scheduler que estaba implementado en el JobTrackerde Hadoop1.
* CapacityScheduler
  + Se configuran una o más colas separadas, cada una pudiendo contener una fracción del total de los recursos
  + Cada cola puede tener una ACL asociada y configuración específica
  + Las particiones de recursos entre colas no pueden ser compartidas ondemand(Si una cola no se utiliza, sus recursos serán desperdiciados)
  + Se puede agregar y cambiar configuración de colas en runtime, pero no así eliminar queues.
  + Cada una de las colas internamente funciona como una cola FIFO

FairScheduler

* Todas las aplicaciones reciben la misma cantidad de recursos
* Acepta la creación de múltiples colas, cada una con su “fair share” de recursos
* Ideal para cuando no se conocen cuantos recursos se van a consumir por aplicación

Preemption

Si ingresa una nueva aplicación y no tiene recursos disponibles ya que algún otro container está usando más de lo que debería, es posible configurar el scheduler para matar containers que no son justos en el uso de sus recursos

## TEZ

Framework de procesamiento distribuido (en clusters) que modela su procesamiento en función a grafos de flujo de datos (DAG -directedacyclicgraph).

* Arquitectura Customizable Diseñada para extensibilidad y optimizaciones de performance definidos por los usuarios.
* Es un proyecto de código abierto de Apache.
* Es utilizado tanto por Apache Hive, como por Apache Pig.
* Actualmente, es una de las implementaciones de procesamiento distribuido utilizadas sobre YARN.

## Spark

* Un framework de computación clusterizado para el análisis de Big Data.
* Open Source originalmente desarrollado en la Universidad de Berkeley en California.
* Provee análisis de datos en memoria.
* Diseñado para ejecutar iterativamente algoritmos y análisis predictivos.
* Altamente Compatible Con los medios de almacenamientos en Hadoop.
* Permite desarrollar programas en múltiples lenguajes de programación.

## Hive

* Hiv Es un framework originalmente desarrollado por Facebook.
* Provee facilidades de datawarehouse sobre un cluster Hadoop.
* Para lograr esto, mapea los datos del HDFS de Hadoop a estructuras propias, aunque también puede utilizar las estructuras de Hbase.
* Además provee un lenguaje para realizar consultas llamado HiveQL(o HQL), muy similar a SQL, mediante el cual se puede escribir tareas de MapReduce, y explotar los datos guardados en el cluster.
* Puede utilizar TEZ/Map-Reduce ó Spark para la ejecución
* Utiliza HDFS para almacenamiento

## Pig

* Desarrollado por Yahoo!
* Plataforma Para el análisis de grandes conjuntos de datos.
* Lenguaje de alto nivel para expresar programas de análisis, junto con la infraestructura para la evaluación de los mismos.
* Programas MapReduce simples de desarrollar utilizando PigLatin como lenguaje de programación
* La estructura es favorable a la paralelización.
* Puede manejar enormes cantidades de información.
* No es necesario saber Java
* 200 líneas de Java → 15 líneas de Pig.

## Kafka

* Apache Kafka es un proyecto de intermediación de mensajes escrito en lenguaje Scala.
* Plataforma unificada, de alto rendimiento y de baja latencia para la manipulación en tiempo real de fuentes de datos.
* Cola de mensajes, bajo el patrón publicación-suscripción.
* Masivamente escalable concebida como un registro de transacciones distribuidas.
* Desarrollada originalmente por Linkedin, liberada en 2011.

## Sqoop

* Permite fácilmente importar y exportar datos desde:
  + Bases de Datos Relacionales
  + Enterprise Datawarehouses
  + NoSQLs
* Permite integración fácil con sistemas basados en Hadoop:
  + Hive
  + Hbase
  + Oozie
* Fue desarrollada por Apache Software Fundation(Grupo Apache)
* Basado en Java

## Hbase

* Base de datos
  + Distribuida
  + Del tipo Columnfamily
  + Diseñada para operar sobre el file system distribuido de Hadoop(HDFS) aprovechando sus características de escalabilidad, tolerancia a fallas, y alta disponibilidad
  + Flexibiliza el uso de un File System de append-only para escrituras y lecturas aleatorias
* HBase es un proyecto open source de Apache cuyo objetivo inicial fue proporcionar un mecanismo de almacenamiento para Hadoop.
* Los datos están organizados lógicamente en “tablas”, “filas” y “columnas”.

## Accumulo

* Proyecto Apache
* Base de datos
  + Del tipo Clave/Valor.
  + Distribuida, escalable y de alto rendimiento.
  + Basada sobre la especificación de Big Table.
  + Diseñada para operar sobre el file system distribuido de Hadoop(HDFS) aprovechando sus características de escalabilidad, tolerancia a fallas, y alta disponibilidad.
  + Construida sobre Hadoopy Zookeeper.
  + Escrita en Java

# Aggregation Framework

Sintaxis

db.collection\_name.aggregate

(

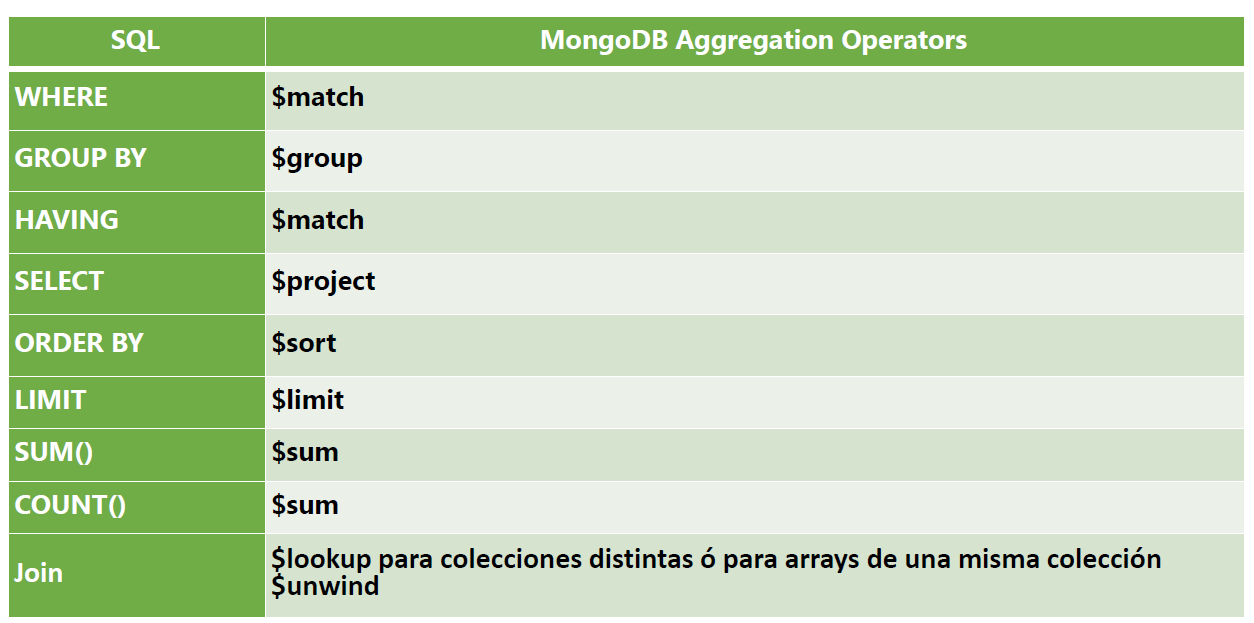
**[{$pipeline\_operator: “expression”},**

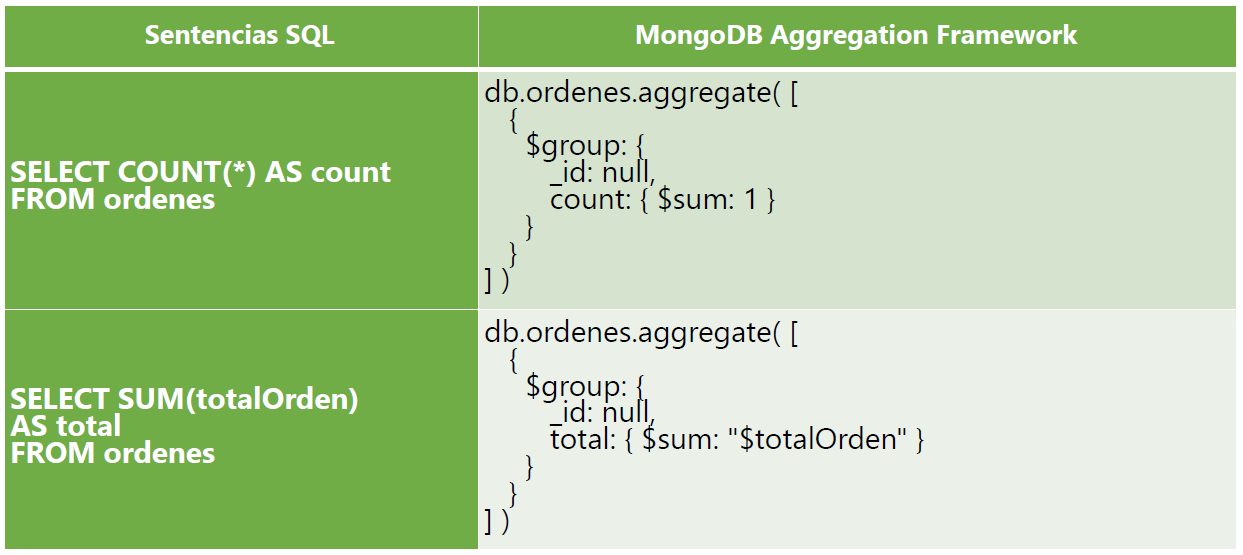
**{$pipeline\_operator: {document\_expression}},…..**

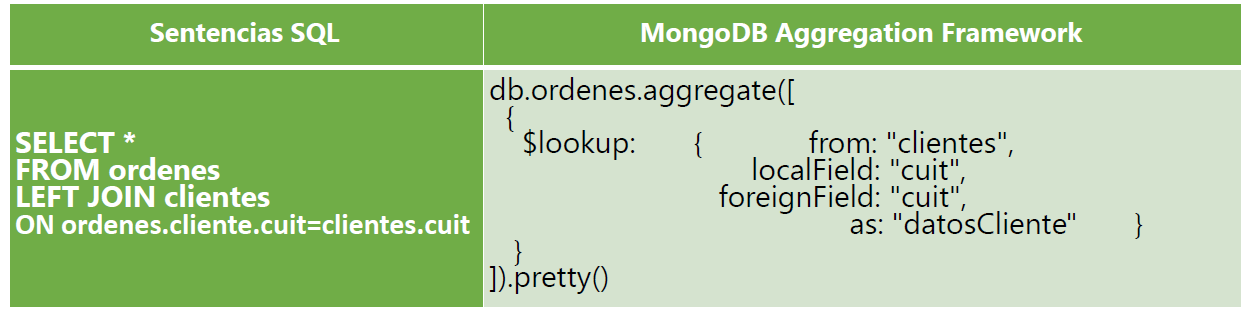
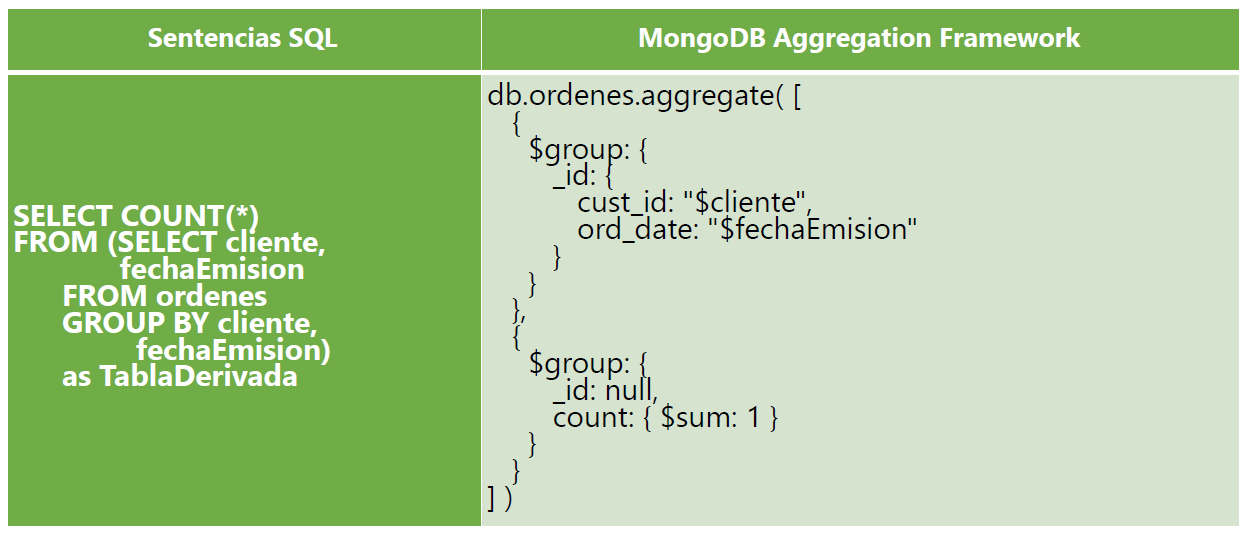
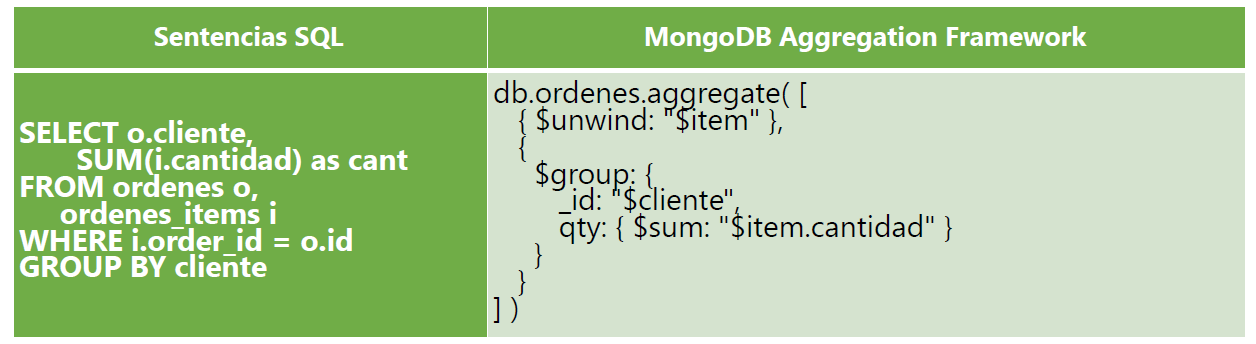
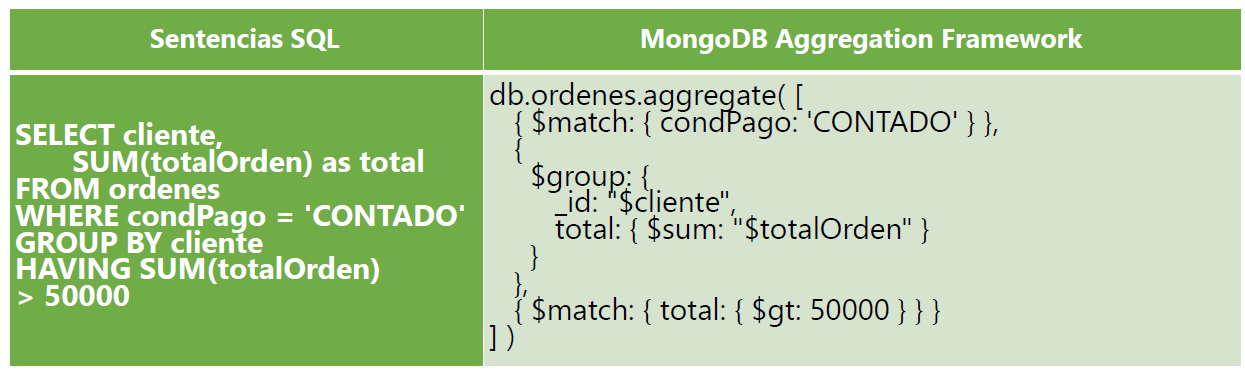
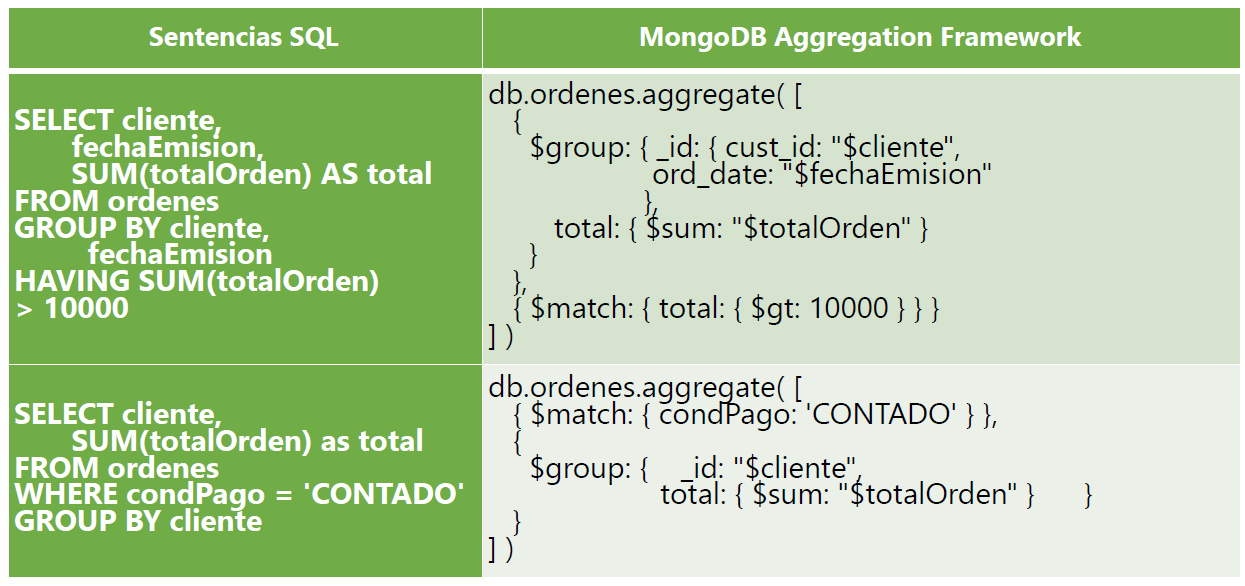
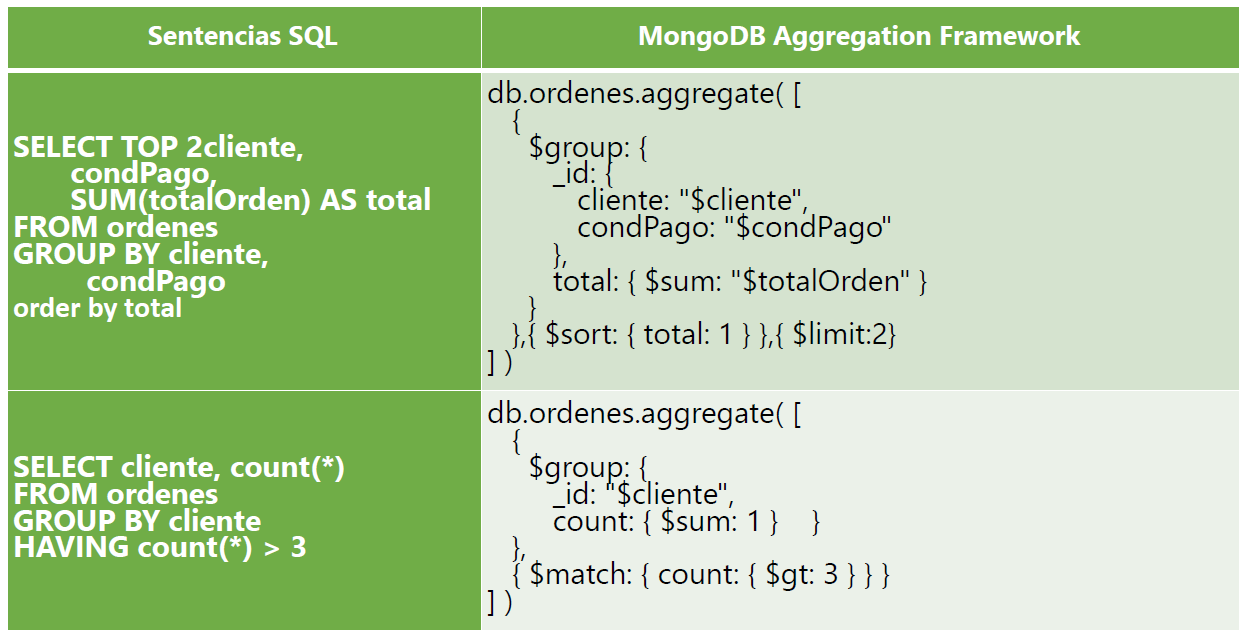
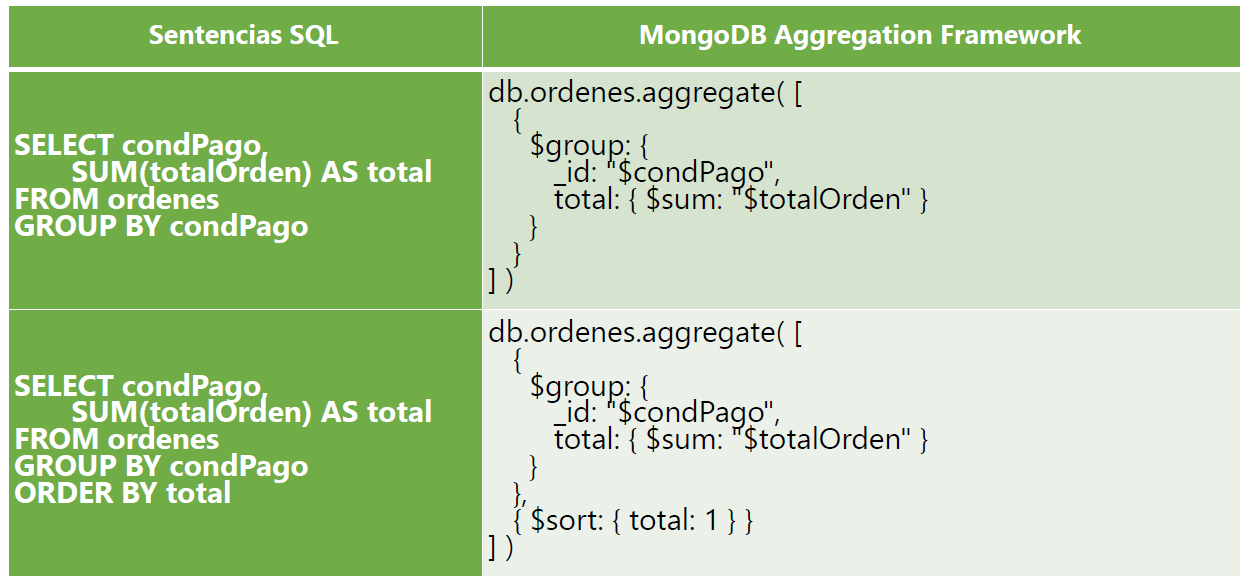
**]**

)

## Operadores







LOOKUP realiza un left join de las colecciones