

# NoSQL

Mariano Beiró

Dpto. de Computación - Facultad de Ingeniería (UBA)

# Topics

- 1 Introducción
- 2 Bases de datos distribuidas
- 3 Bases de datos NoSQL
  - Bases de datos clave-valor
  - Bases de datos orientadas a documentos
  - Bases de datos wide column
  - Bases de datos basadas en grafos
- 4 El modelo MapReduce
- 5 Teorema CAP
- 6 Bibliografía

# 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor
- Bases de datos orientadas a documentos
- Bases de datos wide column
- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

# Introducción

## Contexto histórico

[ELM16 24.1.2]

- La necesidad de diseñar SGBD's no relacionales surgió alrededor del 2000, con la masificación de la Web y algunos cambios tecnológicos.
- Se buscaban nuevas soluciones al problema del almacenamiento de datos que tuvieran:
  - 1 Mayor **escalabilidad** para trabajar con grandes volúmenes de datos
    - Se necesitaba almacenar y procesar cantidades masivas de datos.
    - Esto iba de la mano con el crecimiento en la digitalización de la información y la disponibilidad de datos a través de la red.
    - Fue uno de los objetivos principales en el desarrollo de BigTable (Google Inc., 2005).
  - 2 Mayor **performance en aplicaciones Web**
    - Se buscaban formatos de datos que fueran fáciles de serializar, y que ayudaran a un desarrollo Web más ágil.
    - Surgen XML (1998) y JSON (1999), entre otros.

# Introducción

## Contexto histórico

### 3 Mayor **flexibilidad** sobre las estructuras de datos

- Queremos permitir que una estructura de datos evolucione en el tiempo.
- Los SGBD's relacionales son muy rígidos. Agregar una nueva columna puede ser muy costoso.
- El desarrollo Web busca también darle mayor libertad al desarrollador para organizar los datos en una forma semi-estructurada.

### 4 Mayor capacidad de **distribución**

- Los grandes SGBD's relacionales no habían sido concebidos con la distribución en mente.
- Se busca tener mayor disponibilidad y tolerancia a fallas de parte del SGBD.
- Para ello, se requieren mecanismos de replicación y fragmentación automática de los datos.
- Se prioriza la capacidad de procesamiento distribuido.

# Introducción

## Limitaciones de las bases relacionales

- Los SGBD's relacionales tienen varias limitaciones cuando se trata de escalar los sistemas:
  - Los *joins* de tablas son costosos.
  - El manejo de transacciones en forma distribuida no escala.

Lectura sugerida: " Starbucks Does Not Use Two-Phase Commit",  
G. Hohpe, 2004.

[http://www.enterpriseintegrationpatterns.com/ramblings/18\\_starbucks.html](http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html)

# Introducción

NoSQL

[BERK19]

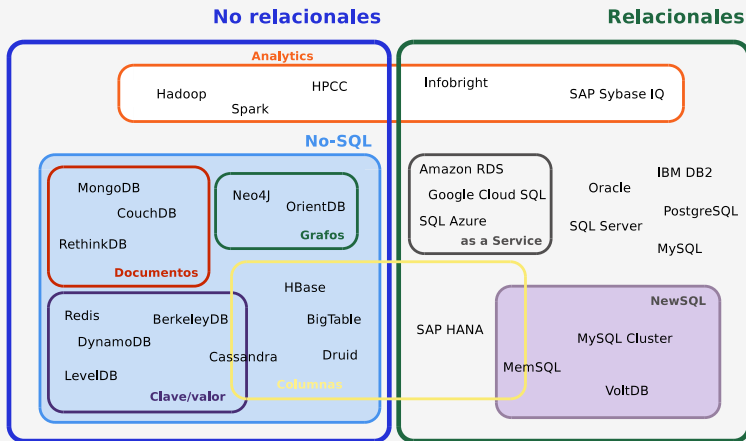
## ■ Contexto de los últimos 15 años:

	2000	2010	2020
Almacenam. (vel.)	+50MB/s (HDD)	+50MB/s (HDD)	+5000MB/s (SSD)
Almacenam. (latencia)	10ms	10ms	25 $\mu$ s/250 $\mu$ s
Almacenam. (\$)	10US\$/GB	0,10US\$/GB	0,20US\$/GB
Red	100Mbps	1Gbps	10 Gbps
CPU	$\approx$ 1GHz	$\approx$ 3GHz	$\approx$ 3GHz
DRAM (vel.)	2 GB/s (DDR)	10 GB/s (DDR3)	20 GB/s (DDR4)
DRAM. (latencia)	100ns	100ns	100ns
DRAM	800 US\$/GB	30 US\$/GB	5 US\$/GB

- Redes cada vez más rápidas ✓ y almacenamiento más barato ✓, pero velocidad de procesamiento estancada ✗.
- En este escenario se diseñaron nuevos SGBD's que rompían con el paradigma del modelo relacional, y por ende con SQL.
- De allí el concepto de “NoSQL” como todo aquello en el mundo de las BD's que no estuviera basado en el modelo relacional ó SQL.

# Introducción

## El universo de los SGBD





# Clasificación de SGBD's No-SQL

## Agregados

[FOWL13 2.1; ELM16 24.1.3]

- Las bases de datos No-SQL se clasifican en distintos tipos:
  - Bases de datos **clave-valor**
  - Bases de datos **orientadas a documentos**
  - Bases de datos ***wide column***
  - Bases de datos **basadas en grafos**
- En cada uno de ellos cambia la definición de **agregado**, es decir, de cómo conjuntos de objetos relacionados se agrupan en colecciones para ser tratados como unidad y ser almacenados en un mismo lugar. Ejemplos de agregados podrían ser:
  - El conjunto de datos personales de un cliente de una empresa
  - Un post de Facebook junto con todos sus comentarios
- Las bases de datos relacionales y las basadas en grafos carecen de la noción de agregado.

## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor
- Bases de datos orientadas a documentos
- Bases de datos wide column
- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

# Bases de datos distribuidas

## Contexto

- Las bases de datos NoSQL buscan aumentar la velocidad de procesamiento y la capacidad de almacenar información, explotando las ventajas que brindan las redes de computadoras y en particular Internet. Para ello implementan la funcionalidad de un **sistema de gestión de bases de datos distribuido**.

### SGBD distribuido

Un *sistema de gestión de bases de datos distribuido* es aquel que corre como sistema distribuido en distintas computadoras (nodos) que se encuentran sobre una red (ya sea local, a través de Internet, o como un servicio de *cloud computing*), aprovechando las ventajas de la computación distribuida y brindando a las aplicaciones la abstracción de ser un único sistema coherente.

# Bases de datos distribuidas

## Temas

- Para comprender las prestaciones de las bases de datos NoSQL introduciremos algunos conceptos sobre SGBD's distribuidos.
- Hablaremos de:
  - Fragmentación
  - Replicación
  - Búsqueda (*lookup*)
    - Tablas de hash distribuidas (DHT's)
  - Modelos de consistencia
    - Consistencia secuencial
    - Consistencia causal
    - Consistencia eventual
  - Métodos de acceso
    - *B-trees* vs. *LSM-trees* y estructuras diferenciales

# Bases de datos distribuidas

## Fragmentación

[ELM16 23.2.1; CONN15 24.4.2]

- La **fragmentación** es la tarea de dividir un conjunto de **agregados** entre un conjunto de nodos.
- Se realiza con dos objetivos:
  - Almacenar conjuntos muy grandes de datos que de lo contrario no podrían caber en un único nodo.
  - Paralelizar el procesamiento, permitiendo que cada nodo ejecute una parte de las consultas para luego integrar los resultados.
- Según la manera de fragmentar, podemos distinguir entre:
  - **Fragmentación horizontal**: Los agregados se reparten entre los nodos, de manera que cada nodo almacena un subconjunto de agregados. Generalmente se asigna el nodo a partir del valor de alguno de los atributos del agregado.
  - **Fragmentación vertical**: Distintos nodos guardan un subconjunto de atributos de cada agregado. Todos suelen compartir los atributos que conforman la clave.
- Muchas veces se utiliza una combinación de ambas.

# Bases de datos distribuidas

## Replicación

[ELM16 23.2.1; CONN15 26.2]

- La **replicación** es el proceso por el cual se almacenan múltiples copias de un mismo dato en distintos nodos del sistema.
- Nos brinda varias ventajas:
  - Es un **mecanismo de *backup***: permite recuperar el sistema en caso de fallas de disco o catastróficas.
  - Permite repartir la carga de procesamiento si permitimos que las réplicas respondan consultas o actualizaciones.
  - Garantiza cierta **disponibilidad** del sistema aún si se caen algunos nodos.
- Cuando las réplicas sólo funcionan como mecanismo de *backup* se denominan **réplicas secundarias**. Cuando también pueden hacer procesamiento, se las conoce como **réplicas primarias**.
- La replicación nos genera un nuevo problema a resolver: la **consistencia** de los datos.
  - Puede darse la situación de que distintas réplicas almacenen (al menos, temporalmente) distintos valores para un mismo dato.

## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor
- Bases de datos orientadas a documentos
- Bases de datos wide column
- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

### ■ Bases de datos clave-valor

■ Bases de datos orientadas a documentos

■ Bases de datos wide column

■ Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía



# Bases de datos clave-valor

[ELM16 24.4]

- Las **bases de datos clave-valor (key-value stores)** almacenan vectores asociativos ó diccionarios, es decir conjuntos formados por pares de elementos de forma (*clave*, *valor*).
  - Ejemplo: (“nombre”, “Luis Poretti”), (“saldo”, 5100)
- Las claves son únicas (es decir, no puede haber dos pares que posean la misma clave), y el único requisito sobre su dominio es que sea comparable por igual (=).
- Algunos ejemplos de *key-value* stores son:
  - Berkeley DB (Oracle)
  - Dynamo (Amazon)
  - Redis

# Bases de datos clave-valor

- Este tipo de bases de datos tiene cuatro operaciones elementales:
  - Insertar un nuevo par (**put**)
  - Eliminar un par existente (**delete**)
  - Actualizar el valor de un par (**update**)
  - Encontrar un par asociado a una clave particular (**get**)
- Sus ventajas son:
  - Simplicidad
    - No se define un esquema.
    - No hay DDL's, restricciones de integridad, ni dominios.
    - El agregado es mínimo, y está limitado al par.
    - El objetivo es guardar y consultar grandes cantidades de datos, pero no de interrelaciones entre los datos.
  - Velocidad
    - Ya que se prioriza la eficiencia de acceso por sobre la integridad de los datos.
  - Escalabilidad
    - Generalmente proveen replicación (ya sea maestro-esclavo ó distribuida) y permiten repartir las consultas entre los nodos.

# Dynamo

- **Dynamo** es el *key-value store* de Amazon.
- Está diseñado siguiendo una **arquitectura orientada a servicio (SoA)**: la base de datos está distribuida en un **server cluster** que posee servidores web, routers de agregación y nodos de procesamiento (Dynamo instances).
- Utiliza un método de *lookup* denominado **hashing consistente** que reduce la cantidad de movimientos de pares necesarios cuando cambia la cantidad de nodos *S*.
  - Esto hace que sea muy sencillo agregar nodos en forma dinámica, con un impacto mínimo.
- Utiliza un modelo de consistencia denominado **consistencia eventual**, que tolera pequeñas inconsistencias en los valores almacenados en distintas réplicas.
- Es totalmente descentralizado. Los nodos son *peers* entre sí.
  - Carece de un punto único de falla.

# Dynamo

## Hashing consistente

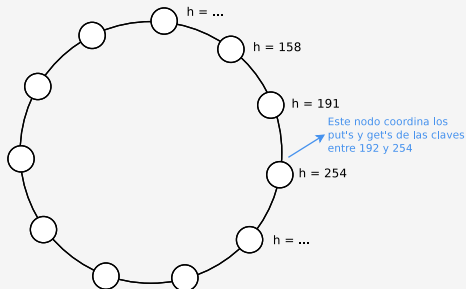
[ELM16 24.4.2]

- Disponemos de una **función de hash**  $h()$  que, dada una clave  $k$ , devuelve un valor  $h(k)$  entre 0 y  $2^M - 1$ , en donde  $M$  representa la cantidad de bits del resultado.
- El valor de la función de hash para un par dado es lo que determina en cuál de los  $S$  nodos el mismo será almacenado. Esto es lo que se conoce como una **tabla de hash distribuida (DHT)**.
- A diferencia de otras DHT's en que el nodo asignado se determina como  $h(k) \bmod S$ , en Dynamo se utiliza una técnica ligeramente distinta, conocida como *hashing consistente*.

# Dynamo

## Hashing consistente

- Al identificador de cada nodo de procesamiento (generalmente, su dirección IP) se le aplica la misma función de hash. A partir de los hashes, los nodos se organizan virtualmente en una estructura de anillo por *hash* creciente.



- **Regla:** Un par  $(k, v)$  se replicará en los  $N$  servidores siguientes a  $h(k)$ , que conformarán el *listado de preferencia* para esa clave.

# Modelos de consistencia

## Consistencia secuencial

[TURU17]

- En el estudio de la **replicación** en las bases de datos distribuidas se utilizan distintos **modelos de consistencia**.
- Uno de los modelos más fuertes, y que proviene de las bases de datos centralizadas, es el de **consistencia secuencial**.
- Partimos de una serie de procesos que ejecutan instrucciones de lectura,  $R_{P_i}(X)$  y de escritura,  $W_{P_i}(X)$ , sobre una base de datos distribuida. → *¡Los procesos están en distintas máquinas!*
- Se dice que una base de datos distribuida tiene consistencia secuencial cuando *“el resultado de cualquier ejecución concurrente de los procesos es equivalente al de alguna ejecución secuencial en que las instrucciones de los procesos se ejecutan una después de otra”*. → *Queremos ver si el conjunto de órdenes locales de las instrucciones se corresponde con algún hipotético orden global.*
  - **Atención!** Esto no quiere decir que los procesos se ejecuten uno después de otro, sino que una instrucción no comienza hasta que otra no haya terminado de aplicarse en todas las réplicas.

# Modelos de consistencia

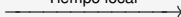
## Consistencia secuencial

- Para indicar el valor leído/escrito utilizaremos esta notación:
  - $R(X)a$  indica que el proceso leyó el valor  $a$  del ítem  $X$ .
  - $W(X)b$  indica que el proceso escribió el valor  $b$  en el ítem  $X$ .
- Ejemplo:
  - El valor inicial de  $a$  es 30, y el de  $b$  es 12.

$P_1$	$R(b)12$	$R(a)3$	$W(a)20$	
$P_2$	$R(a)30$	$R(b)12$	$W(a)3$	$W(b)8$
$P_3$	$R(a)20$			

Nota: Misma columna no implica mismo tiempo físico. Los procesadores pueden no tener sus relojes sincronizados!

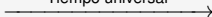
Tiempo local



- Esta ejecución es equivalente a la siguiente ejecución secuencial de las instrucciones:

$P_1$				$R(b)12$	$R(a)3$		$W(a)20$	
$P_2$	$R(a)30$	$R(b)12$	$W(a)3$				$W(b)8$	
$P_3$								$R(a)20$

Tiempo universal



- Por lo tanto, esta ejecución tiene consistencia secuencial.

# Modelos de consistencia

## Consistencia secuencial

- Pero en la siguiente ejecución, en cambio:

$P_1$	R(b)8	R(a)30	W(a)20	
$P_2$	R(a)30	R(b)12	W(a)3	W(b)8
$P_3$	R(a)20			



- No existe una ejecución secuencial equivalente. Evidentemente,  $P_1$  está leyendo un valor de  $a$  desactualizado.
- Esta ejecución no tiene consistencia secuencial.

Garantizar consistencia secuencial es costoso, ya que requiere de mecanismos de sincronización fuertes que aumentan los tiempos de respuesta.



# Modelos de consistencia

## Consistencia causal

[TURU17]

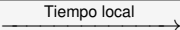
- En el modelo de **consistencia causal** se busca capturar eventos que puedan estar causalmente relacionados.
- Si un evento  $b$  fue influenciado por un evento  $a$ , la causalidad requiere que todos vean al evento  $a$  antes que al evento  $b$ .
- Ejemplo: Supongamos que un proceso  $P_1$  escribe un ítem  $X$ . Simultáneamente, un proceso  $P_2$  lee el ítem  $X$  y escribe el ítem  $Y$ . Las dos escrituras están causalmente relacionadas, porque operan sobre el mismo ítem. Entonces, el modelo requiere que todos las vean en el mismo orden.
- Dos eventos que no están causalmente correlacionados se dicen *concurrentes*, y no es necesario que sean vistos por todos en el mismo orden.
- En el modelo de consistencia causal, *“dos escrituras que están potencialmente causalmente relacionadas deben ser vistas por todos en el mismo orden.”*

# Modelos de consistencia

## Consistencia causal

- Analicemos la siguiente situación en que los valores iniciales son  $a = 5$ ,  $b = 7$  y  $c = 9$ :

$P_1$	W(a)5	R(c)20	R(b)7
$P_2$	W(b)10	R(b)10	R(c)9
$P_3$	W(c)20		



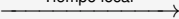
- Aquí las operaciones  $W(b)10$  y  $W(c)20$  son concurrentes, y por lo tanto es consistente que sean vistas por  $P_1$  y  $P_2$  en órdenes distintos.
- La ejecución tiene entonces consistencia causal. Sin embargo, no tiene consistencia secuencial.

# Modelos de consistencia

## Consistencia causal

- Veamos en cambio este ejemplo con valores iniciales  $a = 5$  y  $b = 7$ :

$P_1$	W(a)20		
$P_2$		R(a)20	W(b)3
$P_3$	R(b)3	R(a)5	



- Ahora las operaciones  $W(a)20$  y  $W(b)3$  tienen relación causal, y deben ser vistas por todos en el mismo orden.
- Sin embargo,  $P_3$  las ve en el orden  $W(b)3 \rightarrow W(a)20$  mientras que  $P_2$  las ve en el orden  $W(a)20 \rightarrow W(b)3$ .
- Esta ejecución no tiene consistencia causal.

# Modelos de consistencia

## Consistencia eventual

[TURU17]

- El modelo de **consistencia eventual** está basado en la siguiente observación:
  - En la mayoría de los sistemas reales, son pocos los procesos que realizan modificaciones o escrituras, mientras que la mayor parte sólo lee. ¿Qué tan rápido necesitamos que las actualizaciones de un proceso que escribe sean vistas por los procesos que leen?
- Estas situaciones pueden tolerar un grado bastante más alto de inconsistencia.
- Decimos entonces que una ejecución tiene consistencia eventual cuando *“si en el sistema no se producen modificaciones (escrituras) por un tiempo suficientemente grande, entonces eventualmente todos los procesos verán los mismos valores”*.
- En otras palabras, ésto implica que eventualmente todas las réplicas llegarán a ser consistentes (guardarán los mismos valores).

# Dynamo

## Modelo de consistencia eventual

- Dynamo provee un modelo de **consistencia eventual**, que permite que las actualizaciones se propaguen a las réplicas de forma asincrónica.
- Gracias a ésto, las lecturas y escrituras pueden devolver el control rápidamente.
- Cuando un nodo recibe un *put* sobre una clave, no necesita propagarlo a las  $N - 1$  réplicas antes de confirmar la escritura.
- Dado que las operaciones de *get* pueden realizarse sobre cualquier réplica, es posible leer un valor no actualizado.

Cualquier nodo del listado de preferencias de una clave es elegible para una operación de *put* o *get*.

# Dynamo

## Modelo de consistencia eventual

- Se definen dos parámetros adicionales:
  - $W \leq N$ : *Quorum* de escritura
  - $R \leq N$ : *Quorum* de lectura
- *Quorum* de escritura  $W$ 
  - Un nodo puede devolver un resultado de escritura exitosa luego de recibir la confirmación de escritura de otros  $W - 1$  nodos del listado de preferencia.
  - $W = 2$  ofrece un nivel de replicación mínimo.
- *Quorum* de lectura  $R$ 
  - Un nodo puede devolver el valor de una clave leída luego de disponer de la lectura de  $R$  nodos distintos (incluido él mismo).
  - En muchas situaciones  $R = 1$  es ya suficiente.
  - Valores mayores de  $R$  brindan tolerancia a fallas como corrupción de datos ó ataques externos, pero hacen más lenta la lectura.

# Dynamo

## Consistencia eventual

- Algunos valores comunes de  $R$  y  $W$  son:

N	R	W	
3	2	2	Buena durabilidad y latencia (paper).
3	3	1	Lectura más lenta, pero pobre durabilidad. Escritura rápida.
3	1	3	Escrituras más lentas. Muy buena durabilidad y lectura rápida.

- Para mantener sincronizadas las réplicas, Dynamo utiliza una estructura llamada *Merkle tree* que consiste en un árbol en que cada nodo no-hoja es un *hash criptográfico* de los valores de sus hijos.

# Dynamo

## Aplicaciones

- Dynamo fue originalmente creado por Amazon como un SGBD distribuido para gestionar su sitio de comercio electrónico.
- Luego ha servido de base para la construcción de Dynamo DB, disponible a través de AWS (Amazon Web Services), la plataforma de *cloud computing* de Amazon.
- En Dynamo, los datos deben estar estructurados de manera que las búsquedas sean siempre por clave.
- Una forma de simular la estructura de tabla de los SGBD's relacionales es utilizando pares (clave, valor) con la siguiente estructura:

("NombreEntidad:ValorAtributoClave:NombreAtributo", ValorAtributo)



# Dynamo

## Aplicaciones

- Por ejemplo, para almacenar el precio del producto E34, haríamos:

```
put("PreciosProductos:E34:Precio", 499.00)
```

- Y para consultar la cantidad disponible del producto F80:

```
cantidad = get("PreciosProductos:F80:CantidadDisponible")
```

- ¡No hay funciones de agregación! Debemos implementarlas a mano con un *"file scan"*, ó utilizar herramientas como Spark o Hadoop sobre Dynamo.

## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor

- **Bases de datos orientadas a documentos**

- Bases de datos wide column

- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

# Bases de datos orientadas a documentos

- En las **bases de datos orientadas a documentos**, un **documento** es una unidad estructural de información (agregado), que almacena datos bajo una cierta estructura.
- Sin necesidad de definir un esquema rígido para la estructura del documento, estas bases de datos ofrecen la posibilidad de manejar estructuras un poco más complejas que un simple par **clave: valor**.
- Generalmente, un documento se define como un conjunto de pares **clave: valor** que representan los atributos del documento y sus valores. Se admiten atributos multivaluados, y también se admite que el valor de un atributo sea a su vez un documento.
- La estructura de un documento típicamente se describe con un **lenguaje de intercambio de datos (data exchange language)**.
  - Ejemplos: HTML, XML, JSON, BSON, YAML...

# Bases de datos orientadas a documentos

- Las bases de datos orientadas a documentos comparten algunas características con las bases de datos relacionales, como hacer consultas de selección o agregar datos.
- Algunos ejemplos son:
  - MongoDB
  - RethinkDB
  - CouchDB
  - RavenDB

A continuación tomaremos **MongoDB** como ejemplo para describir su funcionamiento en mayor detalle.



# Bases de datos orientadas a documentos

MongoDB

[ELM16 24.3]

- Basada en *hashes* para identificar a los objetos.
- No utiliza esquemas (*schema-free*). No existe un DDL.
- Los documentos tienen un formato **JSON**.
- Almacena por clave/valor.
- Desarrollada en C++. Tiene APIs en múltiples lenguajes: Python, Java, C#, ...
- Su implementación de la operación de junta es limitada.
- Organiza los datos de una base de datos en **colecciones** que contienen **documentos**.

Modelo relacional	MongoDB
Esquema	Base de datos
Relación	Colección
Tupla	Documento
Atributo	Campo

# MongoDB

## JSON

```
1  Hamburguesa = {  
2      "nombre": "BigBacon",  
3      "ingredientes": ["pan", "carne", "lechuga", "salsa", "pan"  
4          "carne", "tocino", "queso", "pepinillos", "salsa", "pan"  
5          ],  
6      "precio": 129.99,  
7      "calorías": 930  
    }
```

- JSON es una codificación que permite describir estructuras de datos.
- Fue creada para Javascript, pero hoy en día representa en parte una alternativa a XML.
- Un objeto JSON se delimita con `{ }` y posee un conjunto de atributos que se identifican como `clave: valor`.

# MongoDB

## JSON

```
1  Hamburguesa = {  
2      "nombre": "BigBacon",  
3      "ingredientes": ["pan", "carne", "lechuga", "salsa", "pan"  
4          "carne", "tocino", "queso", "pepinillos", "salsa", "pan"  
5          ],  
6      "precio": 129.99,  
7      "calorías": 930  
8  }
```

- Las claves en JSON son siempre *strings*.
- Los valores pueden ser de distintos tipos: *strings*, números (enteros ó de punto flotante), vectores, booleanos, objetos JSON, ó NULL.
- Por lo tanto, los atributos en MongoDB pueden ser multivaluados.

# MongoDB

## Creación de documentos

---

```
from pymongo import MongoClient
conn = MongoClient()

conn.database_names()

# Creamos una nueva base de datos
bd_empresa = conn.base_empresa

# Le agregamos una colección
col_clientes = bd_empresa.clientes

# Y le agregamos un documento a la colección
cliente1 = {
    "nombre": "Mario",
    "apellido": "Wilkerson",
    "domicilio": "Av. Entre Ríos 1560" }
id_cliente1 = col_clientes.insert_one(cliente1).inserted_id
```

---



# MongoDB

## Creación de documentos

---

```
# Insertamos tres clientes más
cliente2 = {
    "nombre": "Horacio",
    "apellido": "Fonseca",
    "localidad": "Morón" }
id_cliente2 = col_clientes.insert_one(cliente2).inserted_id

cliente3 = {
    "apellido": "Gandría",
    "localidad": "Caballito" }
id_cliente3 = col_clientes.insert_one(cliente3).inserted_id

cliente4 = {
    "apellido": "Findo",
    "nombre": "Diego",
    "localidad": "Morón" }
id_cliente4 = col_clientes.insert_one(cliente4).inserted_id
```

---

# MongoDB

## Creación de documentos

---

**#Y definimos algunos productos**

```
col_productos = bd_empresa.productos
```

```
producto1 = {  
    "código": "FD2910",  
    "nombre": "Amoladora",  
    "precio": 2200 }  
producto2 = {  
    "código": "G49",  
    "nombre": "Cinta aisladora",  
    "precio": 40 }  
producto3 = {  
    "código": "EA315",  
    "nombre": "Pinza" }
```

```
id_producto1 = col_productos.insert_one(producto1).inserted_id  
id_producto2 = col_productos.insert_one(producto2).inserted_id  
id_producto3 = col_productos.insert_one(producto3).inserted_id
```

# MongoDB

## ObjectId's

- Los documentos dentro de una colección se identifican a través de un campo `_id`.
- Si no lo indicamos, MongoDB asignará como `_id` un *hash* de 12 bytes. La función `ObjectId(h)` convierte un *hash* en una referencia al documento que dicho *hash* identifica.
- El *hash* también asegura que no se pueda insertar dos veces el mismo documento en una colección
- Cuando hacemos `insert_one()` ó `insert()`, obtenemos una lista de los *hashes* de los documentos creados.

---

```
pprint.pprint(id_producto1)
```

---

```
1 ObjectId('59209152975790214370fcf1')
```

---

# MongoDB

## Consultas básicas

- Las consultas se realizan con la función *find()* sobre la colección.
- El resultado es un **cursor** que debe ser iterado.

---

```
#Buscamos todos los clientes que son de Morón
respuesta_query = col_clientes.find({"localidad": "Morón"})
```

```
for c in respuesta_query:
    pprint.pprint(c)
```

---

```
1 {'_id': ObjectId('59208626975790214370fc98'),
2   'apellido': 'Fonseca',
3   'localidad': 'Morón',
4   'nombre': 'Horacio'}
5 {'_id': ObjectId('59208626975790214370fc9a'),
6   'apellido': 'Findo',
7   'localidad': 'Morón',
8   'nombre': 'Diego'}
```

---

# MongoDB

## Documentos embebidos vs. referenciados

- Podemos utilizar los *ObjectId*'s para **referenciar** objetos:

```
pedido1 = {
    "cod_pedido" : 78303,
    "cliente" : id_cliente2,
    "productos" : [ {"producto": id_producto2, "cantidad": 3} ],
    "fecha_entrega_limite": datetime.datetime(2017, 6, 18),
    "entregado" : False }
pprint.pprint(pedido1)
```

```
1  {'_id': ObjectId('59209fd2975790214370fcff'),
2   'cliente': ObjectId('59209fcd975790214370fcf9'),
3   'cod_pedido': 78303,
4   'entregado': False,
5   'fecha_entrega_limite': datetime.datetime(2017, 6, 18, 0, 0),
6   'productos': [{ 'cantidad': 3,
7                   'producto': ObjectId('59209fcf975790214370fcfd') } ] }
```

# MongoDB

## Documentos embebidos vs. referenciados

```
1  {'_id': ObjectId('59209fd2975790214370fcff'),  
2   'cliente': ObjectId('59209fcd975790214370fcf9'),  
3   'cod_pedido': 78303,  
4   'entregado': False,  
5   'fecha_entrega_limite': datetime.datetime(2017, 6, 18, 0, 0),  
6   'productos': [{ 'cantidad': 3,  
7                   'producto': ObjectId('59209fcf975790214370fcfd') } ] }
```

- En este caso, el documento relativo al “cliente” queda **referenciado** dentro del pedido.
- En cambio, el atributo “productos” contiene como valor un vector de documentos que se encuentran directamente **embebidos (anidados)** dentro del pedido.
- A su vez, cada uno de dichos documentos tiene referenciado un “producto”.

# MongoDB

## Consultas sobre documentos embebidos y referenciados

```
#Buscamos los clientes que pidieron el producto 3:
result = col_pedidos.find({"productos.producto": id_producto3})

for cliente_id in result.distinct("cliente"):
    result2 = col_clientes.find({"_id": cliente_id})
    for cliente in result2:
        pprint.pprint(cliente)
```

```
1  {'_id': ObjectId('5920a5ae975790214370fd0f'),
2    'apellido': 'Fonseca',
3    'localidad': 'Morón',
4    'nombre': 'Horacio'}
5  {'_id': ObjectId('5920a5ae975790214370fd10'),
6    'apellido': 'Gandría',
7    'localidad': 'Caballito'}
```

# MongoDB

## Consultas sobre documentos embebidos y referenciados

---

```
#Buscamos los clientes que pidieron el producto 3:
result = col_pedidos.find({"productos.producto": id_producto3})

for cliente_id in result.distinct("cliente"):
    result2 = col_clientes.find({"_id": cliente_id})
    for cliente in result2:
        pprint.pprint(cliente)
```

---

- Como “productos” está embebido, utilizamos `.` para indicar el subatributo.
- En cambio, como “cliente” es un documento referenciado, debemos hacer una subconsulta para ir a buscar sus datos.
- A qué se parece este tipo de consulta? Junta en SQL!!



# MongoDB

## Juntas

- MongoDB no está pensado para realizar operaciones de junta en forma eficiente.
- En general, cuando necesitamos hacer una junta la escribiremos a mano, como en el ejemplo anterior.
- Si debemos acceder muy frecuentemente al documento referenciado, hay que pensar si no sería conveniente tenerlo directamente embebido.
- Pero, ¿y la **no redundancia** de datos? ¿y la **normalización**?
  - Las sacrificamos, ya que NoSQL, y MongoDB en particular, rompen con el paradigma del modelo relacional.
  - La idea es resignar un poco de normalización para ganar velocidad en el procesamiento de los datos.

# MongoDB

## Juntas

- No debería ser frecuente tener que hacer operaciones de junta entre colecciones completas.
- A lo sumo deberemos ir a buscar documentos referenciados concretos para una consulta en particular.
- Esto último es resuelto en forma muy eficiente en MongoDB porque el *hash* ayuda a encontrar el documento en muy poco tiempo.
- De todas maneras, desde la versión 3.2 de MongoDB existe el comando **lookup**, que permite realizar la junta entre dos colecciones, aunque se recomienda utilizarla con cautela por su baja eficiencia.

# MongoDB

## Agregación

- MongoDB implementa la agregación a través de un *pipeline* secuencial que combina etapas de agrupamiento, selección, etc.
- La función `aggregate()` opera a partir de un vector de documentos JSON, en donde cada documento describe una operación (p. ej., *group* ó *match*) del *pipeline*.

```
#Calculamos la cantidad de clientes que viven en cada localidad,  
#y mostramos sólo aquellas en que vive a lo sumo un cliente:  
result = col_clientes.aggregate( [  
    { "$group": { "_id": "$localidad", "cantidad": { "$sum": 1 } } },  
    { "$match": {"cantidad": { "$lte": 1 } } } ] )  
  
for cliente in result:  
    pprint.pprint(cliente)
```

```
1  {'_id': 'Caballito', 'cantidad': 1}  
2  {'_id': None, 'cantidad': 1}
```

# MongoDB

## Agregación

- El *pipeline* de agregación de MongoDB ofrece las siguientes operaciones, entre otras:
  - **match**: Filtrado de resultados.
  - **group**: Agrupamiento de los resultados por uno o más atributos, aplicando funciones de agregación.
  - **sort**: Ordenamiento de resultados.
  - **limit**: Limitado de resultados.
  - **sample**: Selección aleatoria de resultados.
  - **unwind**: Deconstrucción de un atributo de tipo vector.
- El conjunto de resultados que devuelve una operación será utilizado como entrada por la siguiente operación del *pipeline*.
- Un mismo tipo de operación podría ser utilizado más de una vez dentro del *pipeline*.

# MongoDB

## Agregación: Restaurantes

- El siguiente ejemplo se desarrollará con una base de datos de puntajes de restaurantes extraída de:

`https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/primer-dataset.json`.

- Los datos se encuentran en un archivo con formato `.json`, que importaremos al servidor MongoDB desde la línea de comandos con la siguiente instrucción:

### Importación de documentos JSON

```
mongoimport --db test --collection restaurants --drop --file  
restaurants.json
```

- Dispondremos entonces de una nueva colección llamada “restaurants” en la base de datos “test”.

# MongoDB

## Agregación: Restaurantes

- Cada documento de la colección tiene el siguiente aspecto:

```
1 Restaurant = {
2   "address": {
3     "building": "1007",
4     "coord": [-73.856077, 40.848447],
5     "building": "Morris Park Ave",
6     "zipcode": "10462"
7   }
8   "borough": "Bronx",
9   "cuisine": "Bakery",
10  "grades": [
11    {"date": {"$date": 1393804800}, "grade": "A", "score": 2},
12    {"date": {"$date": 1358985600}, "grade": "A", "score": 10},
13    .....
14    {"date": {"$date": 1322006400}, "grade": "A", "score": 9}],
15  "name": "Morris Park Bake Shop",
16  "restaurant_id": "30075445"
17 }
```

# MongoDB

## Agregación: Restaurantes

**#Ejemplo: Encontrar los 3 barrios en que hay más restaurantes italianos**

```
result = col_restaurants.aggregate( [  
    { "$match": { "cuisine": { "$eq": "Italian" } } },  
    { "$group": { "_id": "$borough", "cantidad": { "$sum": 1 } } },  
    { "$sort": { "cantidad": -1 } },  
    { "$limit": 3 }  
)  
  
for r in result:  
    pprint.pprint(r)
```

```
1  {'_id': 'Manhattan', 'cantidad': 621}  
2  {'_id': 'Brooklyn', 'cantidad': 192}  
3  {'_id': 'Queens', 'cantidad': 131}
```

# MongoDB

#Ejemplo: Hallar nombre y puntaje de los 4 restaurantes de mayor

#puntaje que hayan recibido al menos 5 calificaciones

```
result = col_restaurants.aggregate( [
  { "$unwind": "$grades" },
  { "$group": { "_id": "$restaurant_id", "nombre" : { "$first": "$name"},
    "puntaje": { "$avg": "$grades.score" },
    "cant_calificaciones": { "$sum": 1 } } },
  { "$match": {"cant_calificaciones": { "$gte": 5 } } },
  { "$sort": {"puntaje": -1 } },
  { "$limit": 4 },
  { "$project": {"nombre": "$nombre", "puntaje": "$puntaje" } } ] )$
```

```
1 { '_id': '403934', 'nombre': 'Bella Napoli', 'puntaje': 38.6 }
2 { '_id': '412673', 'nombre': 'Tenda Asian Fusion', 'puntaje': 37.4 }
3 { '_id': '416025', 'nombre': 'Red Chopstick', 'puntaje': 36.3 }
4 { '_id': '415504', 'nombre': 'El Mixteco', 'puntaje': 34.8 }
```



# MongoDB

## Ejemplo

- El código completo de los ejemplos anteriores se encuentra en el archivo `mongo.ipynb`.
- Quedan en el tintero...
  - Validación de datos
  - Creación de índices

# MongoDB

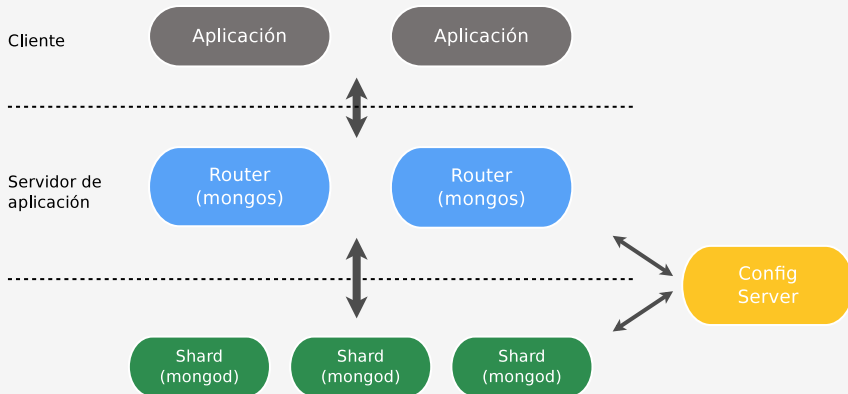
## Sharding

[ELM16 24.3.3]

- MongoDB utiliza un modelo distribuido de procesamiento, conocido como **sharding**.
- Se basa en el particionamiento horizontal de las colecciones en *chunks* que se distribuyen en nodos denominados **shards**. Cada *shard* contendrá un subconjunto de los documentos de cada colección.
- Un **sharding cluster** de MongoDB está formado por distintos tipos de nodos de ejecución:
  - Los **shards (fragmentos)**: Son los nodos en los que se distribuyen los *chunks* de las colecciones. Cada *shard* corre un proceso denominado *mongod*.
  - Los **routers**: Son los nodos servidores que reciben las consultas desde las aplicaciones clientes, y las resuelven comunicándose con los *shards*. Corren un proceso denominado *mongos*.
  - Los **servidores de configuración**: Son los que almacenan la configuración de los *routers* y los *shards*.

# MongoDB

## Sharding: Esquema



# MongoDB

## Sharding

- El particionado de las colecciones se realiza a partir de una **shard key**. La *shard key* es un atributo ó conjunto de atributos de la colección que se escoge al momento de construir el *sharded cluster*.
- La asignación de documentos a *shards* se hace dividiendo en rangos los valores de la *shard key* (**range-based sharding**), o bien a partir de una función de *hash* aplicada sobre su valor (**hashed sharding**).

---

```
sh.shardCollection("db_empresa.col_clientes",  
                  {"localidad": "hashed"}, unique = False)
```

---

# MongoDB

## Sharding

- En un contexto de *sharding* es posible tener algunas colecciones **sharded (fragmentadas)** y otras **unsharded (no fragmentadas)**. Las colecciones *unsharded* de una base de datos se almacenarán en un *shard* particular del cluster, que será el **shard primario** para esa base de datos.
- La realización de un *sharding* sobre una colección posee las siguientes restricciones:
  - 1 Es conveniente que la *shard key* esté definida en todos los documentos de la colección.
  - 2 La colección deberá tener un índice que comience con la *shard key*.
  - 3 Desde MongoDB 5.0, una vez realizado el *sharding* se puede cambiar la *shard key* y desde la versión 4.2 se puede cambiar (*update*) su valor.
  - 4 No es posible defragmentar (*unshard*) una colección que ya fue fragmentada (*sharded*) .

# MongoDB

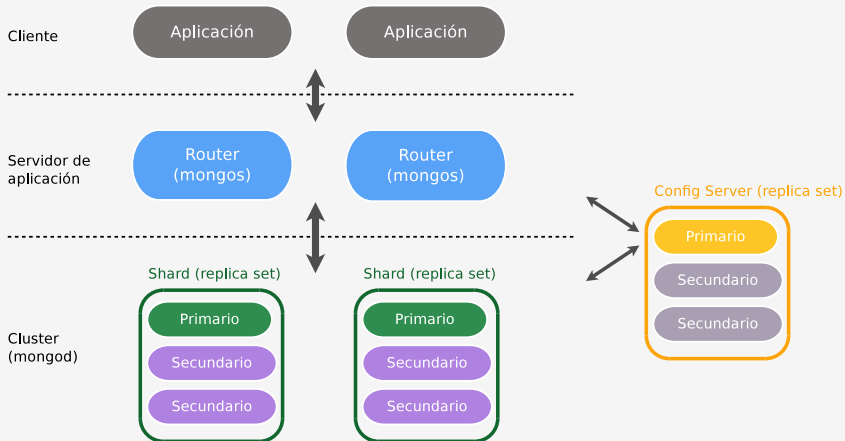
## Sharding: Objetivos

- El *sharding* permite:
  - Disminuir el tiempo de respuesta en sistemas con alta carga de consultas, al distribuir el trabajo de procesamiento entre varios nodos.
  - Ejecutar consultas sobre conjuntos de datos muy grandes que no podrían caber en un único servidor.
- El objetivo es que la base de datos sea **escalable** para proveer soporte al procesamiento de *Big Data*.

# MongoDB

## Replicación

- MongoDB también brinda **tolerancia a fallas** a través de un mecanismo de **replicación** de *shards*.



# MongoDB

## Replicación

- El esquema de réplicas es de **master-slave with automated failover** (**maestro-esclavo con recuperación automática**):
  - Cada *shard* pasa a tener un servidor *mongod* primario (*master*), y uno o más servidores *mongod* secundarios (*slaves*). El conjunto de réplicas de un *shard* se denomina **replica set**.
  - Las réplicas eligen inicialmente un *master* a través de un algoritmo distribuido.
  - Cuando el *master* falla, los *slaves* eligen entre sí a un nuevo *master*.
- También los servidores de configuración se implementan como *replica sets*.
- Todas las operaciones de escritura sobre el *shard* se realizan en el *master*. Los *slaves* sólo sirven de respaldo.
- Los clientes pueden especificar una **read preference** para que las lecturas sean enviadas a nodos secundarios de los *shards*.



## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor
- Bases de datos orientadas a documentos
- **Bases de datos wide column**
- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

## Bases de datos *wide column*

- Son una evolución de las bases de datos clave/valor, ya que agrupan los pares vinculados a una misma entidad como columnas asociadas a una misma clave primaria.
- Un valor particular de la clave primaria junto con todas sus columnas asociadas forma un agregado análogo a la fila de una tabla. Pero además, estas bases *permiten agregar conjuntos de columnas en forma dinámica a una fila*, convirtiéndola en un agregado llamado fila ancha (*wide row*)<sup>1</sup>.
- Esta dinámica podría representar las interrelaciones de la entidad con otra entidad. (**Ejemplo:** Un cliente de una librería y todos los libros que ha comprado.)
- Las más conocidas son:
  - Google BigTable
  - Apache HBase (basado en las ideas de BigTable) [ELM16 24.5]
  - Apache Cassandra (open-source; híbrida wide column/clave-valor)

<sup>1</sup>Aunque por motivos históricos su nombre quedó como *wide column*.

# Cassandra

**Cassandra** es una base de datos NoSQL de tipo *wide column*. Es un híbrido entre las bases de datos wide column y clave-valor.



- ¡No es estrictamente *orientada a columnas*! La organización física de los datos es por fila o *wide row*.
- Surgió en Facebook en 2008, y en 2009 fue adquirida por Apache.
- Actualmente utilizada por: Facebook, Twitter, Netflix.
- No es *libre de esquema*.
- Arquitectura *share-nothing*.
  - No existe un estado compartido centralizado, ni un controlador central, ni una arquitectura maestro-esclavo. Todos los nodos son pares.
  - Esto permite ofrecer una muy alta escalabilidad.
- Está optimizado para ofrecer una *alta tasa de escrituras*.

# Cassandra

## Key spaces y column families

SGBD's relacionales	Cassandra
Esquema	<i>Keyspace</i>
Tabla	<i>Column family</i>
Fila	Fila
-	<i>Wide row</i>

- En Cassandra, el concepto análogo al de *tabla* es el de *column family* (familia de columnas).
- Una *fila* está formada por:
  - Una clave compuesta (atributo ó conjunto de atributos)
  - Un conjunto de pares clave-valor ó *columnas*
- Para Cassandra, cada *columna* no es más que un par clave-valor asociado a una fila.
- Cada *keyspace* (esquema) puede estar distribuido en varios nodos de nuestro *cluster*.

# Cassandra

## *Key spaces y column families*

- Creamos un nuevo *key space* con la siguiente instrucción en CQL:

```
CREATE KEYSPACE empresa_db
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 1 };
```

- Asignamos este *key space* como aquél por defecto:

```
USE empresa_db;
```

- Creación de un *column family*:

```
CREATE COLUMNFAMILY clientes (
  cuit int,
  nombre text,
  domicilio text,
  primary key (cuit));
```

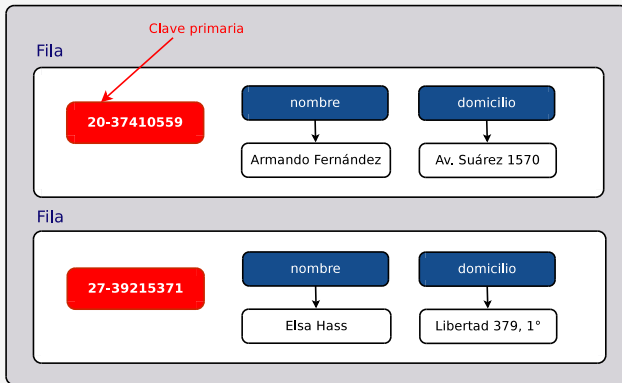
- ¡Es obligatorio definir una clave primaria!

# Cassandra

## Esquema lógico de una fila

[CASS20 4]

### Familia de columnas



# Cassandra

## Wide rows

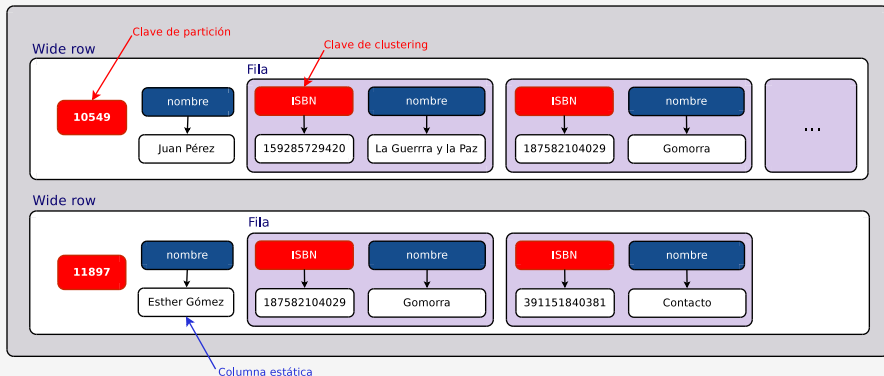
- Sin embargo, la estructura de datos de Cassandra es un poco más compleja.
- La idea es que las columnas de una fila puedan variar dinámicamente en función de las necesidades.
  - **Ejemplo:** Si un cliente nos compra Libros, quisiéramos agregar por cada libro comprado el ISBN y el nombre.
  - Pero, ¿cómo es ésto posible? Durante la creación del *column-family* tenemos que definir en el esquema cuáles van a ser las columnas.
  - **X** En realidad, lo que definimos es un listado de columnas, que cada fila puede instanciar muchas veces.
- Ésto se resuelve seleccionando a *una o más de las columnas como parte de la clave*.
- Cuando en una fila las columnas se repiten identificadas por el valor que toman las columnas clave, se dice que la fila se convirtió en una **wide row (fila ancha)**.

# Cassandra

## Esquema lógico de una *wide-row*

[CASS20 4]

### Familia de columnas



- Nuestra clave primaria queda ahora dividida en dos partes: una **clave de partición** y una **clave de clustering**.
- Adicionalmente, podemos tener **columnas estáticas**, que sean únicas por cada *partition key* (por ejemplo, el nombre del Cliente).



# Cassandra

## Clave primaria: Definición en CQL

- La definición de esta *column family* es:

```
CREATE COLUMNFAMILY clientes (  
    nro_cliente int,  
    nombre text static,  
    ISBN bigint,  
    nombre_libro text,  
    primary key ((nro_cliente), ISBN));
```

# Cassandra

## Clave primaria: Esquema lógico

- La clave primaria en Cassandra se divide en dos partes:
  - La clave de particionado (*partition key*)
  - La clave de clustering (*clustering key*)
- Ambas partes de la clave primaria pueden ser a su vez simples ó compuestas.
- Al igual que en las bases relacionales, pediremos que la clave primaria permita identificar a la fila. Es decir, no puede haber dos filas de una *column family* con igual valor en la clave primaria.
- Pero además, la clave de particionado por si sólo debe alcanzar para identificar a la *wide-row* (en el ejemplo anterior, al Cliente).
- Veremos que **no siempre vamos a pedir que la clave sea minimal**. Podemos agregar a ella los atributos necesarios para las búsquedas que tengamos que hacer en esa *column family*.

# Cassandra

## Clave primaria: Esquema físico

- La correcta definición de la **clave primaria** es fundamental para el funcionamiento de la base de datos en Cassandra. Está muy relacionada con el uso que le vamos a dar a la *column-family* para responder consultas.
- La clave de particionado determina el/los nodo/s del *cluster* en que se guardará la *wide-row* (se utiliza **hashing consistente** para el *lookup*).
- Toda la *wide-row* se almacenará contigua en disco, y la clave de clustering nos determina el ordenamiento interno de las columnas dentro de ella.

# Cassandra

## Clave primaria: restricciones

- El diseño físico de los datos en Cassandra impone algunas restricciones sobre la elección de la clave primaria de cada *column family*<sup>2</sup>:
  - 1 Las columnas que forman parte de la *partition key* deben ser comparadas por igual contra valores constantes en los predicados.
  - 2 Si una columna que forma parte de la *clustering key* es utilizada en un predicado, también deben ser utilizadas todas las restantes columnas que son parte de la *clustering key* y que preceden a dicha columna en la definición de la clave primaria.
  - 3 En particular, si una columna que forma parte de la *clustering key* es comparada por rango en un predicado, entonces todas las columnas de la (*clustering key*) que la preceden deben ser comparadas por igual, y las posteriores no deben ser utilizadas.

<sup>2</sup>Ver: “A Big Data Modeling Methodology for Apache Cassandra”, A. Chebotko, A. Kashlev, S. Lu, IEEE International Congress on Big Data, 2015.

# Cassandra

## Tipos de dato

- Los tipos de dato básicos son similares a los de SQL:
  - **int**, **smallint**, **bigint**, **float**, **decimal**
  - **text**, **varchar**
  - **timestamp**, **date**, **time**
  - **uuid** (clave surrogada)
  - **boolean**
- Adicionalmente, Cassandra permite trabajar con colecciones como tipos de dato:
  - **set** (conjunto de elementos)
  - **list** (lista ordenada de elementos)
  - **map** (conjunto de pares clave/valor)
- (<https://cassandra.apache.org/doc/trunk/cassandra/cql/types.html>)

# Cassandra

## Reglas de diseño

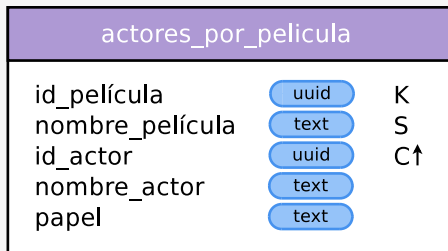
[CASS20 5]

- Para diseñar una base de datos en Cassandra debemos tener en cuenta los siguientes puntos:
  - 1 **No existe el concepto de junta.** Si para alguna consulta típica necesitamos el resultado de una junta, entonces debemos guardarla como una **tabla desnormalizada** más desde el comienzo.
  - 2 **No existe el concepto de integridad referencial.** Si la necesitamos, debe ser manejada desde el nivel de aplicación.
  - 3 **Desnormalización de datos.** En las bases de datos NoSQL el uso de tablas no normalizadas está a la orden del día, y básicamente por un único motivo: *performance*.
  - 4 **Diseño orientado a las consultas.** ¿Cómo saber qué tablas crear si no sabemos cuáles son las consultas a hacer?
    - En los SGBD's relacionales ésto no es un problema. El modelo lógico contiene toda la información, que luego extraeremos haciendo juntas.
    - Aquí en cambio las consultas preceden al modelo de datos: debemos pensar de antemano qué consultas haremos para poder diseñar las tablas.

# Cassandra

## Objetivos de diseño. Diagramas Chebotko

- Buscamos que:
  - Cada consulta se resuelva accediendo a una única *column family*.
  - Los resultados de una consulta estén en una única partición.
  - Se respeten las reglas del lenguaje CQL respecto al uso de la clave primaria.
- Se han propuesto distintos diagramas para el modelado lógico en Cassandra.
- Nosotros utilizaremos uno de los más conocidos: **Chebotko**.

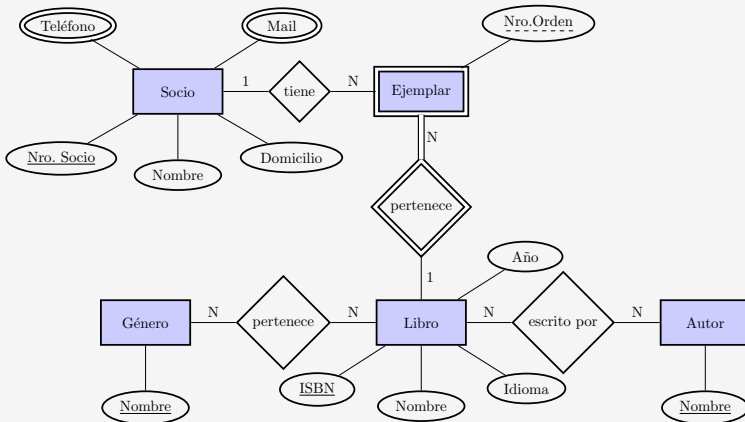


(partition key)  
(static attribute)  
(clustering key)

# Cassandra

## Ejemplo: Base de datos de una biblioteca

- Queremos desarrollar una base de datos para administrar los préstamos en una biblioteca. Partimos del siguiente diagrama:





# Cassandra

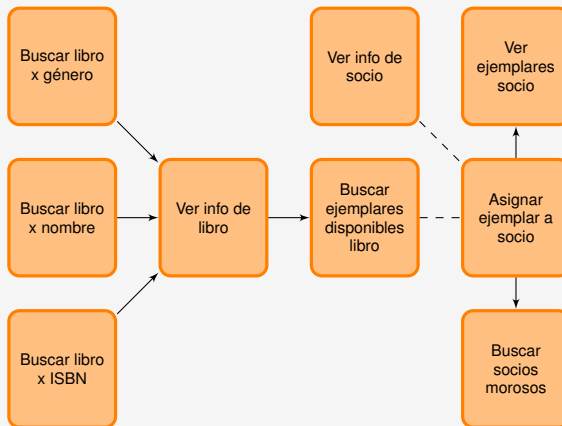
## Ejemplo: Base de datos de una biblioteca. *Workflow*

- El primer paso será definir el *workflow* de nuestra aplicación: cuáles son nuestras necesidades y qué consultas queremos responder.
- Supongamos que nos interesa responder las siguientes consultas:
  - Q1: Buscar libro por género
  - Q2: Buscar libro por nombre
  - Q3: Buscar libro por ISBN
  - Q4: Ver información de un libro
  - Q5: Ver información de un socio
  - Q6: Ver ejemplares disponibles de un libro
  - Q7: Asignar ejemplar a un socio
  - Q8: Consultar ejemplares que posee un socio
  - Q9: Encontrar socios morosos (con préstamos vencidos)

# Cassandra

## Ejemplo: Base de datos de una biblioteca. *Workflow*

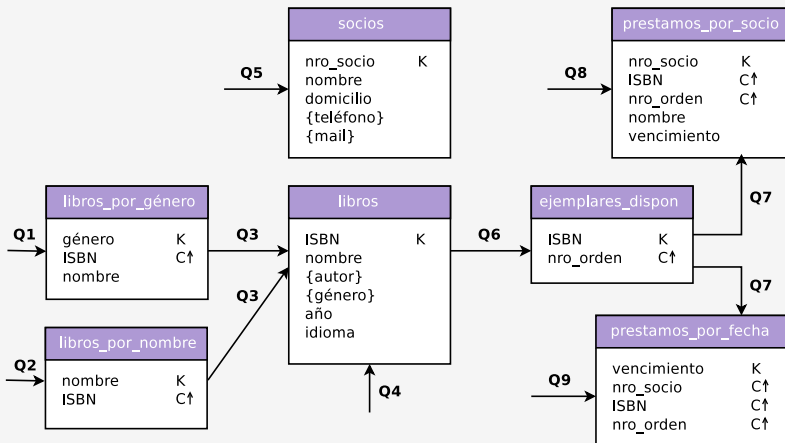
- El siguiente sería un posible *workflow* para nuestros datos:



# Cassandra

## Ejemplo: Base de datos de una biblioteca. Diagrama Chebotko

- Por cada consulta idearemos una *column family* que la resuelva:



# Cassandra

## Ejemplo: Base de datos de una biblioteca. Column-families

```
CREATE COLUMNFAMILY socios (  
    nro_socio int,  
    nombre text,  
    domicilio text,  
    telefono set<int>,  
    mail set<text>,  
    primary key (nro_socio)  
);  
  
CREATE COLUMNFAMILY prestamos_por_socio (  
    nro_socio int,  
    ISBN bigint,  
    nro_orden int,  
    nombre text,  
    vencimiento date,  
    primary key ((nro_socio), ISBN, nro_orden)  
);
```

# Cassandra

## Ejemplo: Base de datos de una biblioteca. Column-families

```
CREATE COLUMNFAMILY ejemplares_dispon (  
    ISBN bigint,  
    nro_orden int,  
    primary key ((ISBN), nro_orden)  
);  
  
CREATE COLUMNFAMILY prestamos_por_fecha (  
    vencimiento date,  
    nro_socio int,  
    ISBN bigint,  
    nro_orden int,  
    primary key ((vencimiento), nro_socio, ISBN, nro_orden)  
);  
  
...
```

# Cassandra

Ejemplo: Base de datos de una biblioteca. Ejercicios en CQL.

- 1 Dado un número de socio, encuentre los ejemplares que el socio posee, indicando el nombre del libro, el número de orden del ejemplar y la fecha de vencimiento del préstamo.

```
SELECT nombre, ISBN, nro_orden, vencimiento  
FROM prestamos_por_socio  
WHERE nro_socio = 751;
```

- 2 Dado el ISBN de un libro, encuentre los nombres de los autores del mismo.

```
SELECT autor  
FROM libros  
WHERE ISBN = 14292859338291;
```

# Cassandra

Ejemplo: Base de datos de una biblioteca. Ejercicios en CQL.

- 3 Dado el ISBN y número de orden de un ejemplar disponible de un libro, y dado un número de socio, elimine el ejemplar del listado de ejemplares disponibles y asígnelo en préstamo al socio.

```
DELETE FROM prestamos_por_socio
WHERE nro_socio = 751 AND
      ISBN = 55102184963921 AND
      nro_orden = 2;

INSERT INTO ejemplares_dispon (ISBN, nro_orden)
VALUES (55102184963921, 2);
```

# Métodos de acceso en Cassandra

- Cassandra está optimizado para altas tasas de escritura.
- Utiliza una estructura de búsqueda denominada **LSM-tree** (*log-structured merge tree*), que mantiene parte de sus datos en memoria, para diferir los cambios sobre el índice en disco.
- Se busca acceder en forma secuencial a disco, para mejorar el *trade-off* entre el costo de hacer un *disk seek* y el costo de un buffer en memoria. Esto ha sido bastante estudiado y se conoce como **regla de los cinco minutos** (*Five-minute Rule*).

## Five-minute rule [GRAE08]

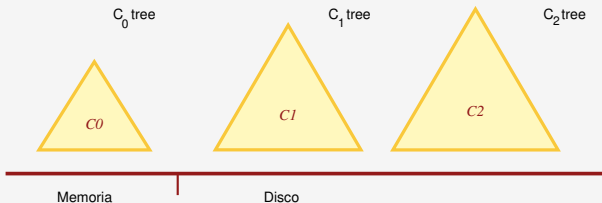
¿Cada cuanto tiempo debe accederse a un ítem de dato para que convenga mantenerlo en memoria?

- J. Gray y G. Putzolu, 1985 → 5 minutos
- Depende de las tecnologías de memoria y disco. La tendencia es que aumente (Ej. 2007, SSD a DRAM → 15 minutos, G. Graefe).



# Log-structured Merge Trees (LSM-trees)

[SILB19 24.2]



## ■ Consideraciones:


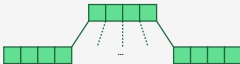
- Desde que se inserta una entrada en  $C_0$  hasta que se traslada a  $C_1$  habrá una demora.
- El costo de I/O de escritura en  $C_0$  es nulo.
- Cuando el tamaño de  $C_0$  alcanza un umbral se inicia un proceso de *rolling merge (flush)*.
- El árbol  $C_1$  suele tener una estructura similar a un B-tree.
- En cambio, como  $C_0$  está en memoria no es relevante minimizar su profundidad → suelen emplearse árboles balanceados como el *2-3 tree* o el *árbol AVL*.

# Métodos de acceso

## LSM-trees vs. B-trees

[SILB19 24.2]

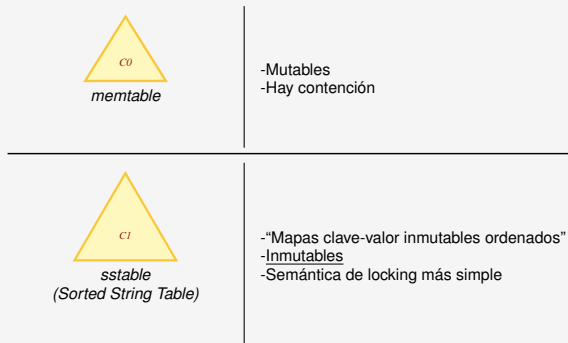
- Los LSM-trees fueron introducidos en 1995, y Google los utilizó en su sistema *BigTable* en 2005.
- Desde entonces fueron adoptados por Apache Cassandra y MongoDB (en el *engine* WiredTiger) entre otras bases NoSQL.

	LSM-trees	B-trees
		
Acceso a disco durante escritura	Secuencial ( <i>rolling merge</i> ).	Aleatorio.
Costo de lectura	Medio.	Muy bajo.
Costo de escritura	Bajo.	Alto.

# Métodos de acceso en Cassandra

## Implementación de LSM-trees

[PETR17]



### ■ Mejoras:

- Mantener más de un nivel en disco:  $C_1, C_2, \dots$
- Mantener varios archivos por cada nivel, y mergearlos cuando superan un umbral (*compaction*).
- Utilizar *Bloom filters* para verificar si una entrada *podría* estar en uno de los archivos.

## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor
- Bases de datos orientadas a documentos
- Bases de datos wide column
- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

# Bases de datos basadas en grafos

- En las **bases de datos basadas en grafos** los elementos principales son *nodos* y *arcos (ejes)*.
- Estas bases de datos resultan útiles para modelar interrelaciones complejas entre las entidades.
- Ejemplo: Las bases de datos relacionales modelan las relaciones entre entidades distintas utilizando claves foráneas.
  - Ejemplo:
    - Persona(dni, nombre, f\_nac)
    - HijoDe(dni, dni\_padre)
  - ¿Cómo hacemos si queremos encontrar a todos los descendientes de una persona?
  - En bases de datos relacionales esta consulta tiene alto costo porque requiere de múltiples juntas.
- Las bases orientadas a grafos utilizan una estructura en que cada nodo mantiene una referencia directa a sus nodos adyacentes.

# Bases de datos basadas en grafos

## Aplicaciones

- Organizar nuestra base de datos de esta forma nos provee ventajas para resolver problemas clásicos de grafos como:
  - Encontrar patrones de nodos conectados entre sí.
  - Encontrar caminos entre nodos.
  - Encontrar la ruta más corta entre dos nodos.
  - Calcular medidas de centralidad asociadas a los nodos.
- En general, es una buena idea utilizarlas cuando en nuestro modelo conceptual encontramos que las instancias de los tipos de entidades mantienen interrelaciones con otras instancias de su mismo tipo de entidad.

# Neo4j

[ELM16 24.6]

- Neo4j es una de las bases de datos orientadas a grafos más conocidas.
- Está desarrollada en Java.
- Es actualmente utilizada por empresas como Cisco, HP y Huawei.
- Incluye soporte para transacciones ACID y para bases de datos distribuidas.
- Posee APIs en distintos lenguajes: Python, Ruby, Java, ...
- Utiliza un lenguaje de consulta declarativo denominado **Cypher**.

# Neo4j

- El siguiente ejemplo fue desarrollado utilizando el servidor de Neo4j para Linux.<sup>3</sup>

## 1-Instalación del servidor de Neo4j

Descargar Community Edition desde

`https://neo4j.com/download/other-releases/  
tar-zxf-neo4j-community-4.3.13-unix.tar.gz`

## 2-Lanzamiento del servidor de Neo4j

```
cd neo4j  
./bin/neo4j start
```

## 3-Lanzamiento del cliente

`http://localhost:7474`

<sup>3</sup> Como alternativa se puede jugar con la consola online: <http://console.neo4j.org/>



# Neo4j

## Estructura: Nodos, *labels* y propiedades. **CREATE**

- Una base de datos Neo4j está formada por **nodos**.
- Un nodo puede tener distintos **labels**. Dentro de cada *label*, el nodo tendrá un conjunto de **propiedades** con determinados **valores**.

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

- No existe una estructura rígida respecto a qué propiedades deben tener los nodos con determinado *label*.

```
1 (edith:Persona {nombre: 'Tomás', color: 'Verde', prof: 'Músico'})
2 (maria:Persona {nombre: 'María', prof: 'Estudiante'})
3 (gaby:Persona {nombre: 'Gabriel', color: 'Verde', prof: 'Médico'})
```

- En Cypher, los nodos se crean con el comando **CREATE**:

```
1 CREATE (pepe:Persona {nombre: 'Pepe', color: 'Azul'})
```

# Neo4j

## Estructura: Consultas básicas con **MATCH** y **WHERE**

- Para buscar un nodo ó conjunto de nodos utilizamos el comando **MATCH**:

```
1 MATCH (p:Persona {nombre: 'María'}) RETURN p.nombre, p.prof
```

- El resultado de la consulta es un conjunto de **records**, que podemos representar con una tabla:

nombre	profesión
María	Estudiante

- También se pueden aplicar condiciones de selección sobre las búsquedas con el comando **WHERE**:

```
1 MATCH (m:Persona) WHERE m.color='Verde' RETURN m.nombre, m.prof
```

nombre	profesión
Edith	Músico
Gabriel	Médico

# Neo4j

## Estructura: Interrelaciones

- Podemos utilizar el comando **CREATE** para definir interrelaciones entre los nodos:

```
1 MATCH (juan:Persona {nombre:'Juan'}),
2     (lucas:Persona {nombre:'Lucas'}),
3     (edith:Persona {nombre:'Edith'}),
4     (maria:Persona {nombre:'María'}),
5     (tom:Persona {nombre:'Tomás'}),
6     (luis:Persona {nombre:'Luis'})
7 CREATE (juan)-[:AMIGO_DE]->(lucas),
8        (edith)-[:AMIGO_DE]->(maria),
9        (maria)-[:AMIGO_DE]->(lucas),
10       (lucas)-[:AMIGO_DE]->(tom),
11       (luis)-[:AMIGO_DE]->(edith),
12
13       (lucas)-[:ENEMIGO_DE]->(edith),
14       (tom)-[:ENEMIGO_DE]->(edith),
15       (tom)-[:ENEMIGO_DE]->(luis)
```

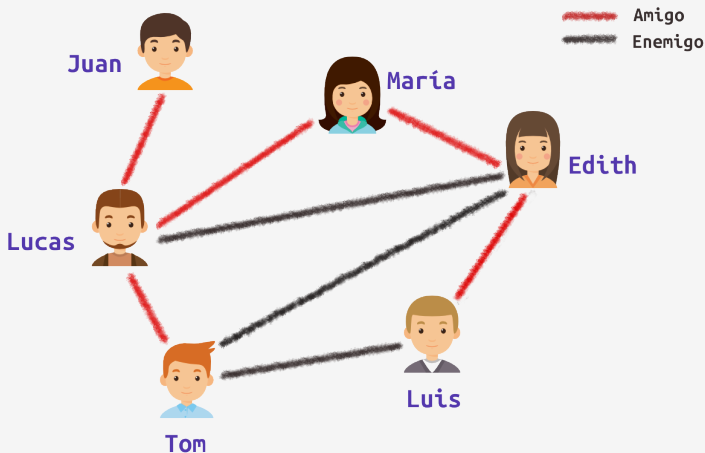
# Neo4j

## Direccionalidad

- En Neo4j, los ejes son siempre direccionales. (Los grafos son dirigidos.)
- Sin embargo, para trabajar con grafos no dirigidos no es necesario crear las interrelaciones en los dos sentidos.
- Es posible indicar en la consulta Cypher si queremos prestar atención a la dirección de los ejes en la navegación, o no.
  - `->`: Requiere que se respete la dirección del eje.
  - `-`: Pasa por alto la dirección del eje en la interrelación.

# Neo4j

## Estructura: Consultas básicas sobre interrelaciones



- Queremos descubrir, dentro de este grafo, si existen enemigos que tengan amigos en común.

# Neo4j

## Estructura: Consultas básicas sobre interrelaciones

### ■ Ayuda: hoja de referencia de Cypher

<http://neo4j.com/docs/pdf/neo4j-cypher-refcard-stable.pdf>

¿Existen enemigos con amigos en común?

```

1      MATCH (n:Persona)-[:AMIGO_DE]-(m:Persona),
2          (m:Persona)-[:AMIGO_DE]-(o:Persona),
3          (n:Persona)-[:ENEMIGO_DE]-(o:Persona)
4      RETURN n.nombre, o.nombre

```

n.nombre	o.nombre
Lucas	Edith
Edith	Lucas

# Neo4j

## Estructura: Consultas básicas sobre interrelaciones

- Con un **\*** en la interrelación podemos indicar una cantidad indeterminada de saltos.

¿A cuántos amigos de distancia están Juan y Luis?

```

1  MATCH (juan:Persona {nombre:'Juan'})
2      (luis:Persona {nombre:'Luis'})
3      p=(juan:Persona)-[:AMIGO_DE*]-(luis:Persona)
4  RETURN length(p)
5  ORDER BY length(p)
6  LIMIT 1

```

length(p)
-----------

4
---

- Lo que hicimos fue encontrar todos los *caminos*, ordenarlos por longitud y quedarnos con la longitud del menor.

# Neo4j

## Esquema general de consulta

- El esquema general de una consulta en Cypher es:

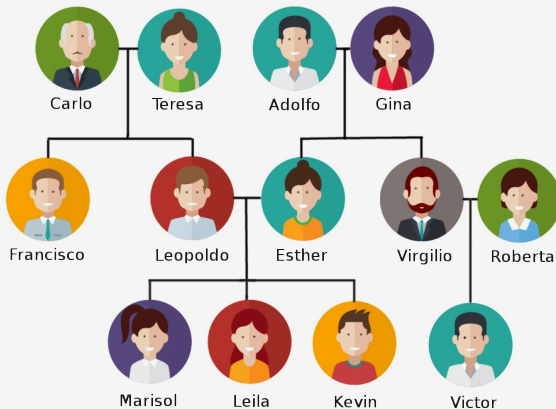
```
MATCH  $p_1 = pattern_1, p_2 = pattern_2, \dots, p_n = pattern_n$ 
WHERE  $cond_1, cond_2, \dots, cond_m$ 
RETURN  $p_{i_1}, \dots, agg(p_{j_1}), \dots$ 
ORDER BY  $p_{i_k}, agg(p_{j_{k'}}), \dots$ 
LIMIT N;
```

- Un patrón (*pattern*) puede especificarse a través de un nodo y sus propiedades, una interrelación y sus propiedades, o un camino y sus propiedades. A cada patrón podemos darle un nombre.
- La cláusula **WHERE** dá aún más flexibilidad para filtrar los resultados basados en alguna condición.
- Las operaciones de agregación se realizan siempre en el **RETURN**, aplicando funciones como `count(*)`, `sum()` ó `max()`.



# Neo4j

## Ejemplo: Árbol Genealógico



- En este grafo tenemos interrelaciones de tipo *HIJO\_DE* y *ESPOSO*.

# Neo4j

## Ejemplo: Árbol Genealógico

¿Cómo se llaman los primos de Victor?

```

1  MATCH (victor:Persona {nombre:'Victor'}),
2      (primo:Persona)-[:HIJO_DE]->(t:Persona)-[:HIJO_DE]->
3      (a:Persona)<-[:HIJO_DE]-(p:Persona)<-[:HIJO_DE]-(victor)
4  WHERE p<>t
5  RETURN DISTINCT primo.nombre

```

nombre
Leila
Marisol
Kevin

# Neo4j

## Ejemplo: Árbol Genealógico

¿Quiénes son ancestros de Victor?

```
1 MATCH (victor:Persona {nombre:'Victor'}),  
2   (p:Persona)<-[:HIJO_DE*]-(victor:Persona)  
3 RETURN DISTINCT p.nombre
```

nombre
Virgilio
Gina
Adolfo
Roberta

# Neo4j

## Ejercicio

### Ejercicio

En una base de datos Neo4J tenemos nodos con label “Producto” (que se identifican con un atributo “*producto\_id*”) y otros con label “Persona” (identificados con un atributo “*nombre*”).

Hay a su vez definida una interrelación “*COMPRÓ*” entre personas y productos, con un atributo “*fecha*”. Por último, existe una interrelación “*RECOMENDÓ*” entre personas, con un atributo “*fecha*” y un atributo “*producto\_id*”.

Nos gustaría saber qué personas –luego de comprar– recomendaron un producto a otra persona que luego también lo compró, para poder recompensarlas.

El formato de la salida debe ser (*nombre*, #*recom*).

# Neo4j

## Ejercicio

### Respuesta

```
1      MATCH (per1:Persona),
2          (per2:Persona),
3          (prod:Producto),
4          (per1)-[c1:COMPRO]->(prod),
5          (per2)-[c2:COMPRO]->(prod),
6          rec=(per1)-[r:RECOMENDO]->(per2)
7      WHERE c1.fecha<r.fecha AND
8            c2.fecha>r.fecha AND
9            r.producto_id = prod.producto_id
10     RETURN DISTINCT per1.nombre, count(rec)
```

## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor
- Bases de datos orientadas a documentos
- Bases de datos wide column
- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

# MapReduce

[ELM16 25]

- **MapReduce** es una técnica que brinda un marco flexible para el procesamiento paralelo de grandes volúmenes de datos.
- Fue propuesto por Google en un paper de 2004<sup>4</sup>, y luego tomado por Yahoo! para la implementación de Hadoop (2006).
- Sin embargo, MapReduce no introdujo ningún concepto nuevo. Es una reimplementación de conceptos de agregación que ya eran conocidos 30 años atrás.
- El concepto detrás de MapReduce es el de dividir una entrada de datos en porciones que puedan ser ejecutadas por distintas unidades de procesamiento, y luego integrar el resultado para generar una salida, que a su vez servirá de entrada a otra etapa de procesamiento.

<sup>4</sup>“*MapReduce: Simplified Data Processing on Large Clusters*”, J. Dean, S. Ghemawat. <https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf>.

# MapReduce

## Concepto

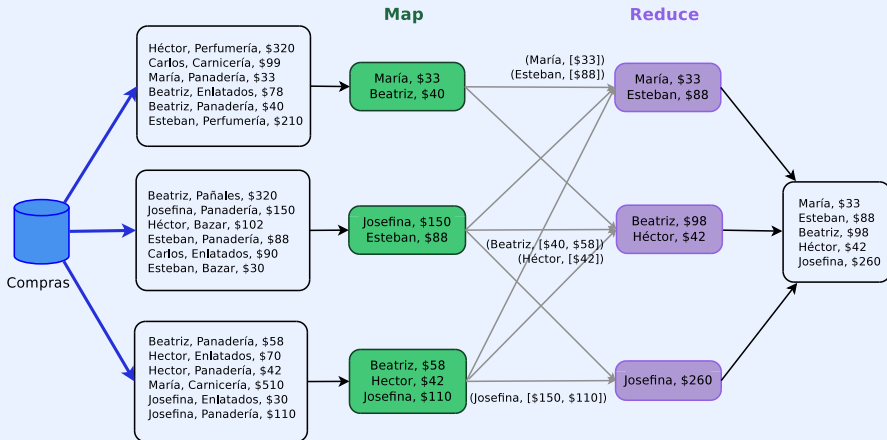
- Un pipeline MapReduce está formado por una secuencia de etapas de los siguientes tipos:
  - **Map:** Recibe un par clave/valor,  $(k, v)$ , y devuelve un conjunto de pares clave/valor,  $[(k_1, v_1), \dots]$ .
  - **Reduce:** Recibe un par clave/[valores],  $(k, [v_1, \dots])$  (es decir, una clave y un conjunto de valores asociados a esa clave) y devuelve un conjunto de valores  $[v_1, \dots]$  (típicamente es un único valor).
- Observaciones:
  - 1 La entrada del *pipeline* es un conjunto de pares  $(k, v)$  que se distribuyen en forma equitativa entre los nodos de procesamiento. En muchos casos,  $v$  no se utiliza en la entrada, y se le asigna directamente un valor por defecto.
  - 2 Entre medio de cada *map* y cada *reduce* hay una función de ordenamiento y *shuffling* interna que organiza la salida de los *map* para agruparlos por clave y luego llamar a *reduce*.
  - 3 Al escribir las funciones *map()* y *reduce()*, es el usuario quien termina diseñando todo el plan de su consulta.



# MapReduce

## Concepto

- En el siguiente ejemplo, queremos calcular cuál fue el gasto mensual de cada cliente en el rubro “Panadería”.



# MapReduce

## Concepto

- En este caso, las funciones map() y reduce() podrían ser:

---

```
def map(k, v):  
    l = list()  
    if (k.rubro=='Panadería'):  
        l.append((k.cliente, k.monto))  
    return l  
  
def reduce(k, list_v):  
    return (k, sum(list_v))
```

---

# MapReduce

## Ejercicio

### Ejercicio

Queremos saber qué palabras aparecen más de 100 veces en “El Ingenioso Hidalgo Don Quijote de la Mancha”.

Para ello leemos el texto de a una línea a la vez, de manera que cada línea se convierte en la entrada de un *pipeline* MapReduce.

La estructura de entrada al *pipeline* estará entonces compuesta por pares  $(k, v)$ , en donde  $k$  es un vector de palabras y  $v$  es igual a 0.

Escriba las funciones  $\text{map}(k, v)$  y  $\text{reduce}(k, \text{list\_}v)$  que permitan resolver el problema.

# MapReduce

## Ejercicio

### Solución en Python

---

```
def map(k, v):  
    d = dict()  
    for word in k:  
        if (word in d.keys()):  
            d[word] = d[word] + 1  
        else:  
            d[word] = 1  
    return d.items()  
  
def reduce(k, list_v):  
    freq_k = sum(list_v)  
    if freq_k > 100:  
        return (k, freq_k)
```

---

# Hadoop

- Hadoop es la implementación de MapReduce de Yahoo!.
- Está programado en Java, y almacena sus datos en un sistema de archivos *ad hoc* denominado HDFS (*Hadoop Distributed File System*).
- Hadoop escribe todos los datos de las etapas intermedias en disco. (*materialización intermedia*)
- Cada etapa del *pipeline* MapReduce lee desde la entrada estándar (*stdin*) y produce sus resultados en la salida estándar (*stdout*). Cuando escribimos un *pipeline*, podemos testearlo ejecutando:

## Ejercicio

```
cat entrada.txt | python mapper.py | sort -k1,1 |  
python reducer.py > salida.txt
```

## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor
- Bases de datos orientadas a documentos
- Bases de datos wide column
- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

# Teorema CAP

[ELM16 24.2]

- En 1998 el científico E. Brewer postuló la imposibilidad de que un sistema de bases de datos distribuido garantice simultáneamente el máximo nivel de:
  - (C) Consistencia (*consistency*)
  - (A) Disponibilidad (*availability*)
  - (P) Tolerancia a particiones (*partition tolerance*)
- En 2002 Seth Gilbert y Nancy Lynch presentaron una prueba formal del resultado.

# Teorema CAP

## Consistencia

- La **consistencia** la propiedad de que en un instante determinado el sistema muestre un único valor de cada ítem de datos a los usuarios.
- Su nivel máximo es la consistencia secuencial, en la que todas las operaciones de lectura/escritura distribuidas en el sistema pueden ordenarse de forma tal que toda lectura de un ítem siempre lea el último valor escrito en ese ítem.
- Alcanzar consistencia secuencial requiere de un alto nivel de sincronización entre los nodos.



# Teorema CAP

## Disponibilidad y tolerancia a particiones

- La **disponibilidad** consiste en que toda consulta que llega a un nodo del sistema distribuido que no está caído reciba una respuesta efectiva (es decir, sin errores).
- **Atención!** Quizás esa respuesta no esté actualizada si algún ítem había sido modificado en algún nodo que actualmente está caído, o bien está caído el enlace hacia él.
- La **tolerancia a particiones** consiste en que el sistema pueda responder una consulta aún cuando algunas conexiones entre algunos pares de nodos estén caídas.
- En un sistema que garantiza consistencia y disponibilidad no podríamos tolerar un particionado, porque no podemos garantizar que la respuesta que damos tenga el máximo nivel de consistencia.

# Teorema CAP

## Garantías posibles

- El Teorema CAP dice entonces que a lo sumo podremos ofrecer 2 de las 3 garantías:
  - **AP:** Si la red está particionada, podemos optar por seguir respondiendo consultas aún cuando algunos nodos no respondan. Garantizaremos disponibilidad, pero el nivel de consistencia no será el máximo.
  - **CP:** Con la red particionada, si queremos garantizar consistencia máxima no podremos garantizar disponibilidad. Es posible que no podamos responder una consulta en forma efectiva porque esperamos mensajes de confirmación desde nodos que no pueden comunicarse.
  - **CA:** Si queremos consistencia y disponibilidad, entonces no podremos tolerar que una cantidad indeterminada de enlaces se caiga.

# Teorema CAP

## Solución de compromiso

- En la realidad no es factible garantizar que una red no se particione, con lo cual nuestro sistema deberá necesariamente ser tolerante a particiones.
- El teorema nos obliga entonces a encontrar una solución de compromiso entre consistencia y disponibilidad.
- En el campo específico de la ejecución de transacciones, estas limitaciones llevaron a la definición de garantías más débiles que las ACID: las **propiedades BASE**.

# Teorema CAP

## BASE vs. ACID

- Las propiedades BASE representan un sistema distribuido con:
  - (BA) Disponibilidad básica (*basic availability*): El SGBD distribuido está siempre en funcionamiento, aunque eventualmente puede devolvernos un error, o un valor desactualizado.
  - (S) Estado débil (*soft state*): No es necesario que todas los nodos réplica guarden el mismo valor de un ítem en un determinado instante. No existe entonces un “*estado actual de la base de datos*”.
  - (E) Consistencia eventual (*eventual consistency*): Si dejaran de producirse actualizaciones, eventualmente todos los nodos réplica alcanzarían el mismo estado.

## 1 Introducción

## 2 Bases de datos distribuidas

## 3 Bases de datos NoSQL

- Bases de datos clave-valor
- Bases de datos orientadas a documentos
- Bases de datos wide column
- Bases de datos basadas en grafos

## 4 El modelo MapReduce

## 5 Teorema CAP

## 6 Bibliografía

# Bibliografía

## Bases de Datos Distribuidas

[ELM16] Fundamentals of Database Systems, 7th Edition.

R. Elmasri, S. Navathe, 2016.

Capítulo 23

[GM09] Database Systems, The Complete Book, 2nd Edition.

H. García-Molina, J. Ullman, J. Widom, 2009.

Capítulo 18, Capítulo 19

[SILB19] Database System Concepts, 7th Edition.

A. Silberschatz, H. Korth, S. Sudarshan, 2019.

Capítulo 24.2

[CONN15] Database Systems, a Practical Approach to Design, Implementation and Management, 6th Edition.

T. Connolly, C. Begg, 2015.

Capítulo 24, Capítulo 26

# Bibliografía

## NoSQL

[ELM16] Fundamentals of Database Systems, 7th Edition.

R. Elmasri, S. Navathe, 2016.

Capítulo 24

[TIW11] Professional NoSQL.

S. Tiwari, 2011.

[FOWL13] NoSQL Distilled.

P. Sadalage, M. Fowler, 2013.

[CELKO13] Complete Guide to NoSQL.

J. Celko, 2014.

[SULL15] NoSQL for Mere Mortals.

D. Sullivan, 2015.

# Bibliografía

## Links y artículos

[GRAE08] The Five-minute Rule 20 Years Later.

G. Graefe, ACM Queue, 2008.

<https://queue.acm.org/detail.cfm?id=1413264>.

[PETR17] On Disk IO, Part 3: LSM Trees.

A. Petrov, 2017.

[https://medium.com/databasss/  
on-disk-io-part-3-lsm-trees-8b2da218496f](https://medium.com/databasss/on-disk-io-part-3-lsm-trees-8b2da218496f).

[TURU17] Consistency, causal and eventual.

S. Turukin, 2017.

[http://sergeiturukin.com/2017/06/29/  
eventual-consistency.html](http://sergeiturukin.com/2017/06/29/eventual-consistency.html).

[BERK19] Latency Numbers Every Programmer Should Know.

Colin Scott, Berkeley University, 2019.



# Bibliografía

## Bibliografía específica

---

[MONGO19] MongoDB, The Definitive Guide, 3rd Edition.

S. Bradshaw, E. Brazil, K. Chodorow. O'Reilly, 2019.

[CASS20] Cassandra, The Definitive Guide, 3rd Edition.

J. Carpenter, E. Hewitt. O'Reilly, 2020.

[GRAPH15] Graph Databases, 2nd Edition.

I. Robinson, J. Webber, E. Eifrem. O'Reilly, 2015.