# Introducción a bases de datos NoSQL 2

//Segunda parte: Conceptos

IT BOARDING

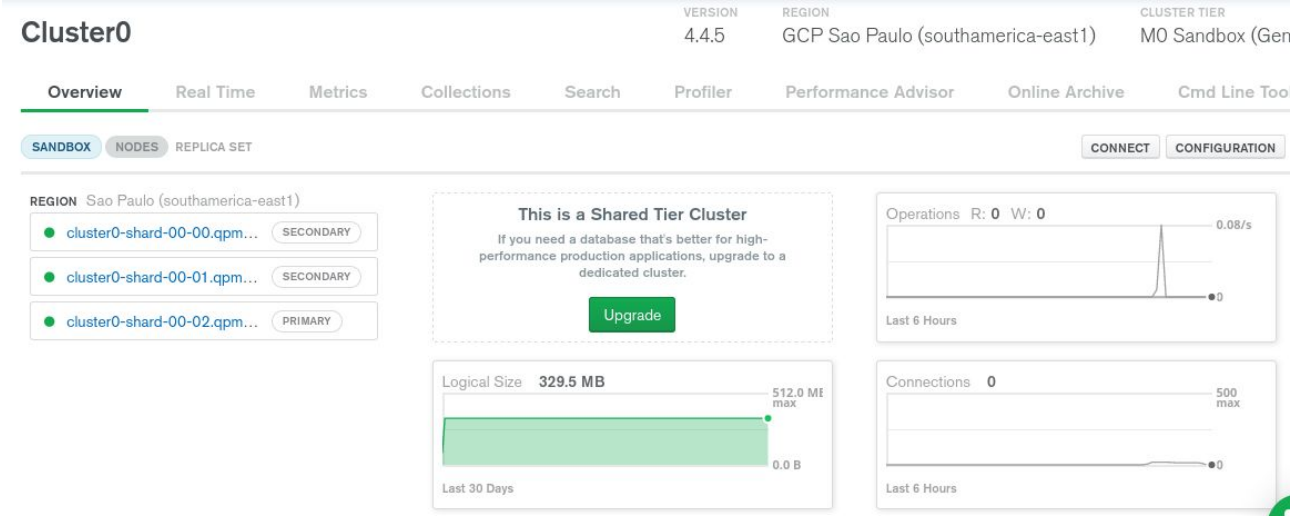
BOOTCAMP

# Replicación

Ver más: https://docs.mongodb.com/manual/replication

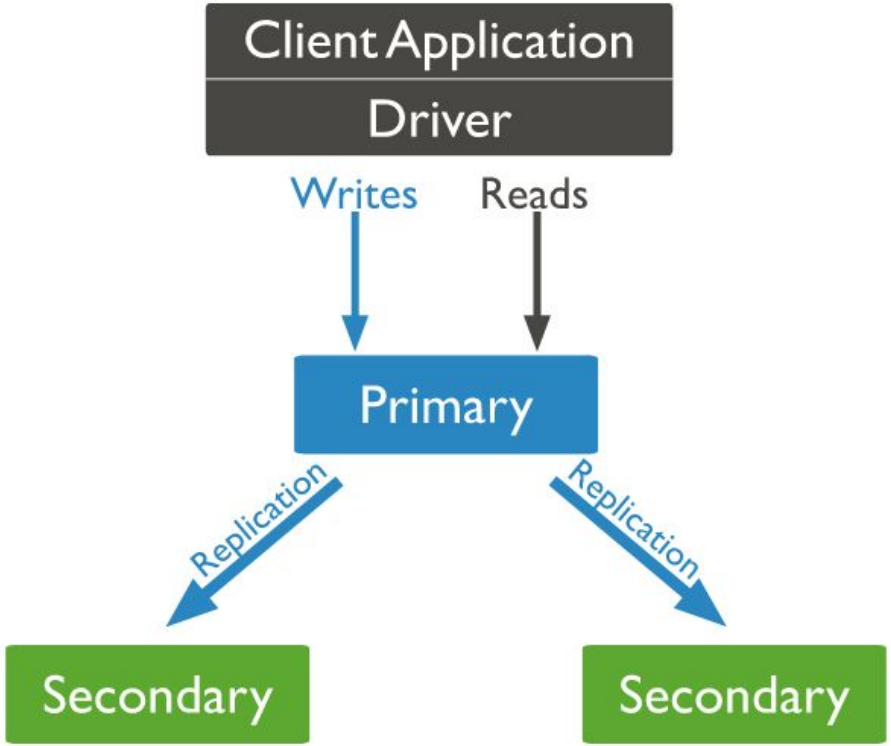
Un *replica set* —conjunto de réplicas— en MongoDB es un grupo de procesos que mantienen el mismo *dataset*. Cada proceso corre en una máquina distinta (nodo) dentro de un grupo de máquinas (*cluster*).

Las réplicas proveen **redundancia y tolerancia a fallas**, son la base de todos los *deploys* en producción.



# Nodo primario

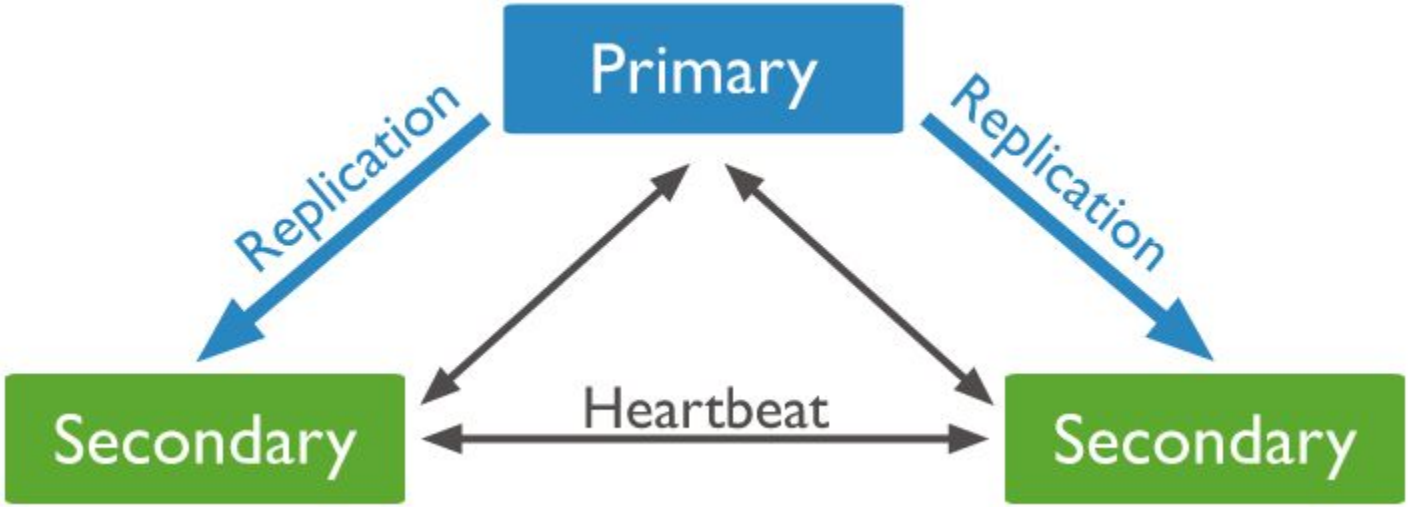
El **nodo primario** recibe todas las operaciones de escritura y lectura; solo puede haber uno.



# Nodos secundarios

Los **nodos secundarios** replican las operaciones del primario en sus datasets, de tal manera de reflejarlo.

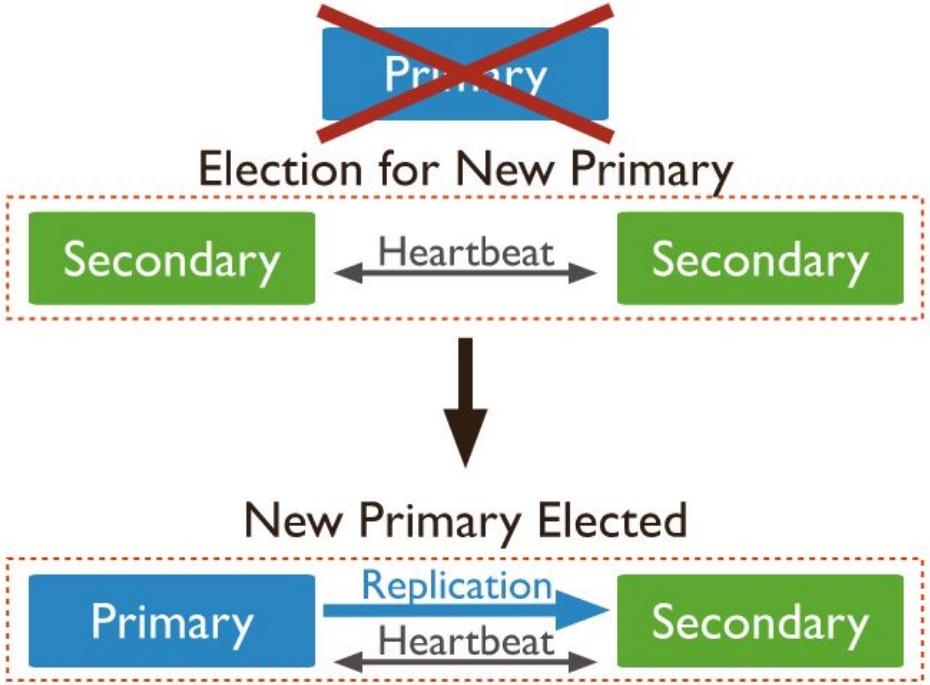
Si el nodo primario no está disponible, un secundario iniciará ocupar su lugar.



# Tolerancia a fallas

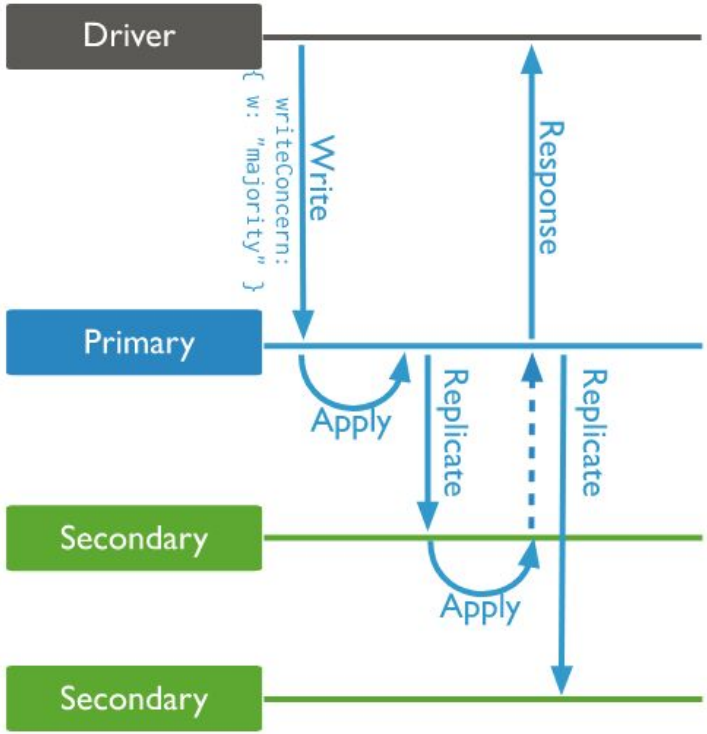
Cuando un primario no se comunica con los otros nodos dentro de un cierto periodo (10 segundos por defecto), un secundario llama a elección para nominarse como el nuevo primario.

El *replica set* **no puede procesar operaciones de escritura** durante la elección.



# Write concern

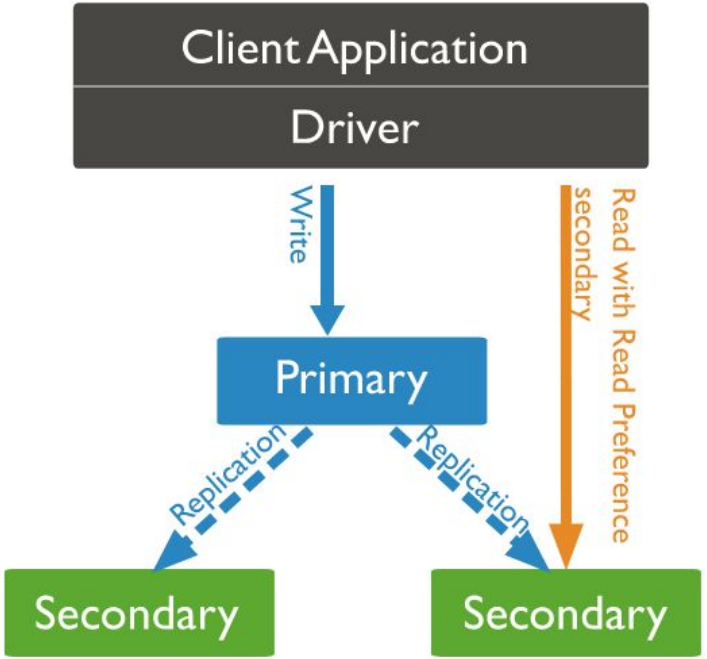
Las operaciones de escritura requieren un **reconocimiento de persistencia**. Este reconocimiento es configurable; cuando su valor es "mayoría", la escritura se confirma cuando las operaciones se hayan propagado en la mayoría de los nodos.



## Preferencia de lectura

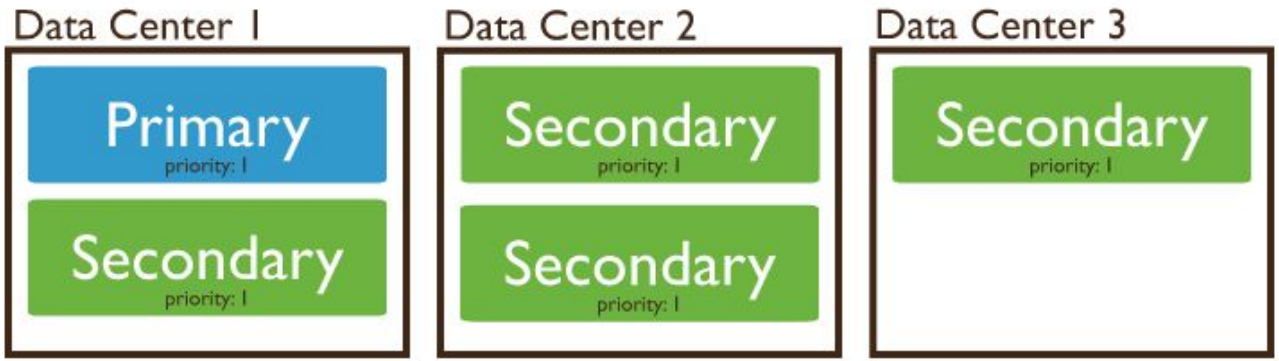
Opcionalmente, es posible habilitar la lectura desde nodos secundarios para aumentar la capacidad de lectura.

MongoDB se vuelve **eventualmente consistente**, permitiendo leer datos potencialmente desactualizados.

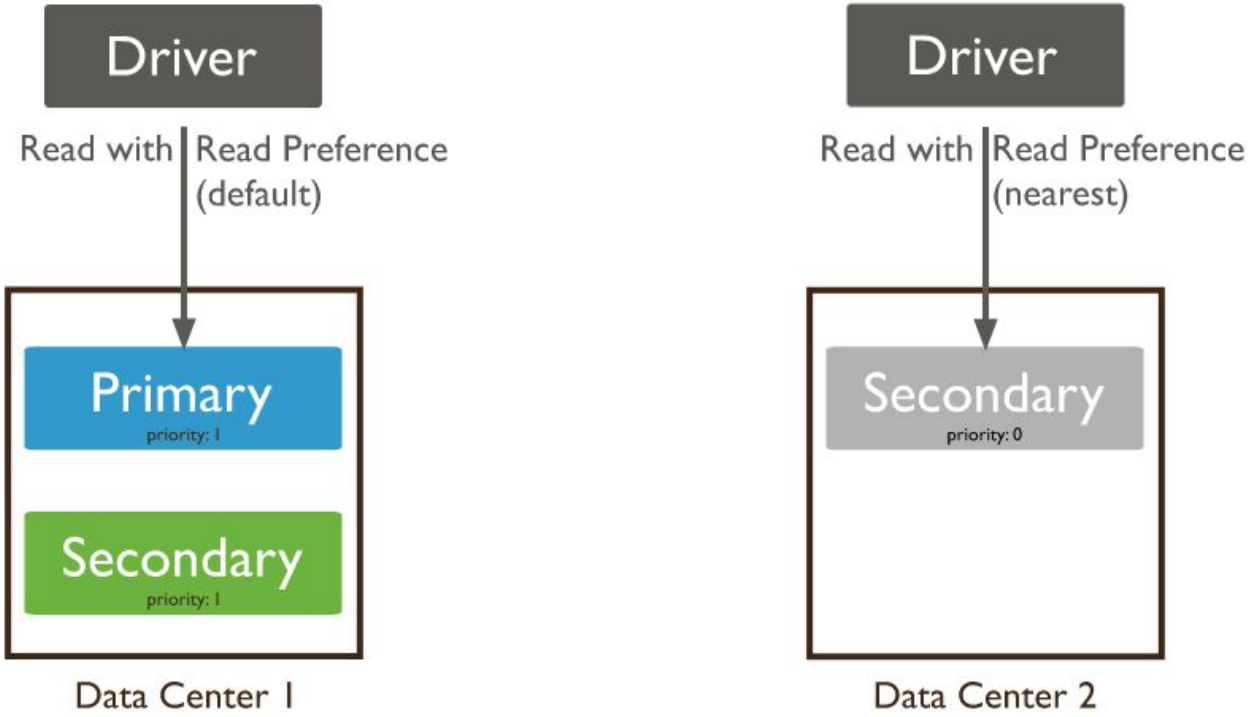


## Distribución geográfica

Distribuir nodos en distintos centros de datos mejora la redundancia y la tolerancia a fallas.



Los clientes pueden configurarse para preferir leer desde nodos secundarios para mejorar la latencia.



# Fragmentación

Ver más: https://docs.mongodb.com/manual/sharding

*Sharding* —fragmentación— en MongoDB es un método para distribuir los datos a lo largo de múltiples máquinas.

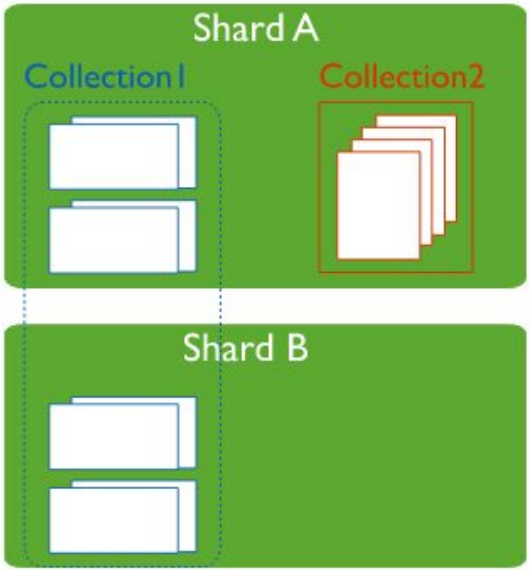
Es útil cuando el *dataset* es muy grande y/o la cantidad de operaciones es muy alta. Un conjunto de datos de trabajo que supera la RAM del sistema estresa la lecto-escritura de los discos. Una alta tasa de consultas exhausta al CPU.

Dividir la carga entre múltiples servidores y añadir servidores según necesidad para aumentar la capacidad es lo que se conoce como **escalibilidad horizontal**.

Incrementar las capacidades de procesamiento, memoria y almacenamiento de un único servidor tiene el nombre de **escalibilidad vertical**; en la práctica esta escalabilidad es limitada.

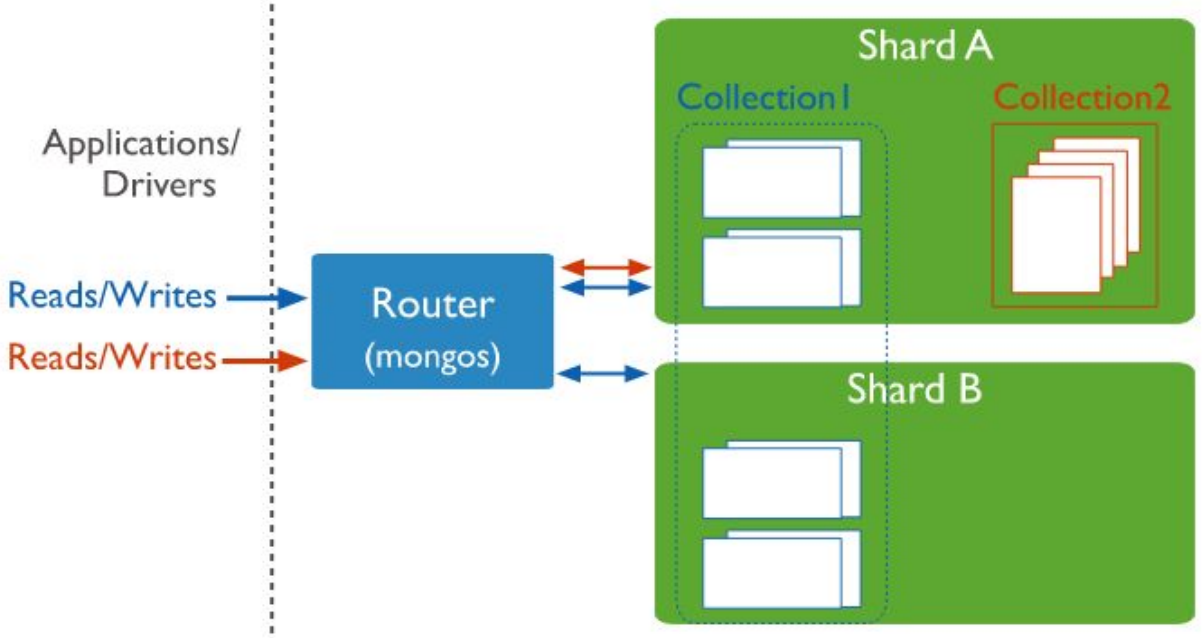
Las colecciones fragmentadas son particionadas y distribuidas en los *shards* —nodos— del *cluster*.

Las colecciones no particionadas son guardadas en un *shard* primario.



# Conexión

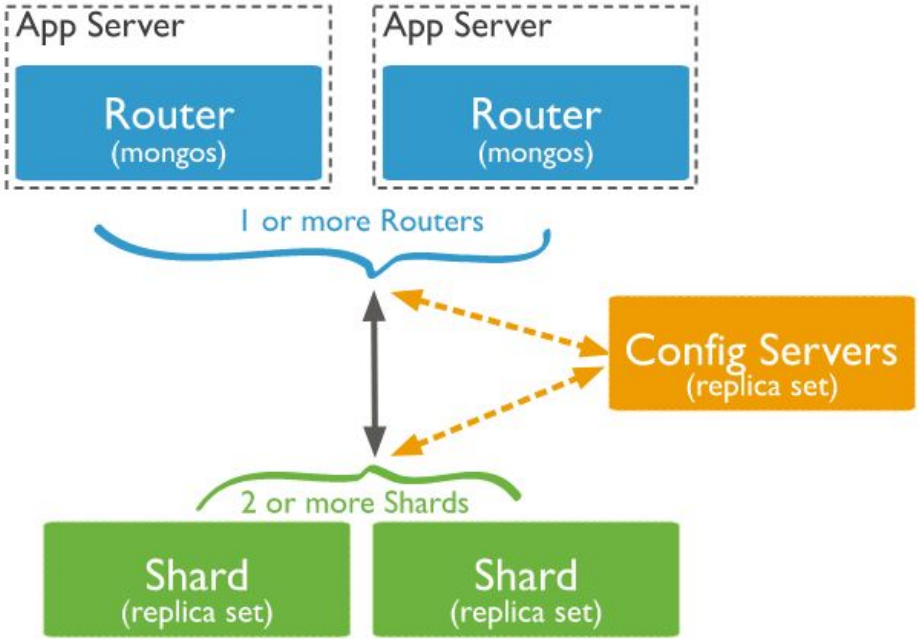
Los clientes jamás deberían conectarse a un fragmento (nodo que corre un proceso mongod) directamente. Deben hacerlo a través de un nodo enrutador (corre un proceso mongos).



# Cluster fragmentado

Consiste en los siguientes componentes:

- Fragmentos. Cada uno de estos nodos puede deployarse como un *replica set*.
- Enrutadores.
- Servidores de configuración.



# Shard keys

La *shard key* es un **índice simple o compuesto** que determina la distribución de los documentos de una colección entre los fragmentos del cluster.

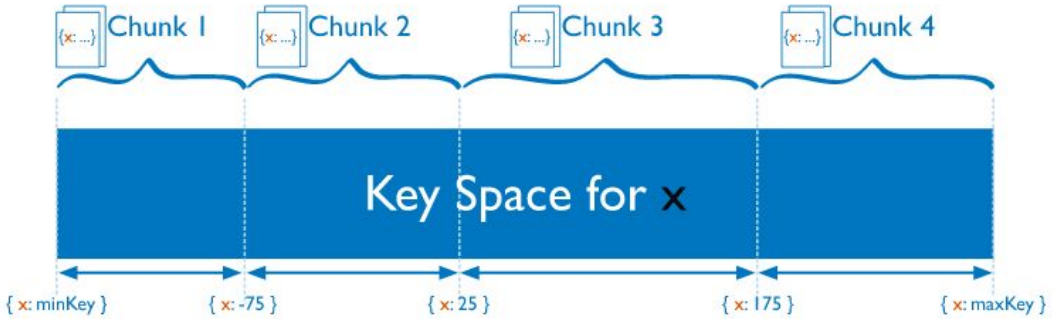
MongoDB divide los valores de la llave. Cada rango es asociado a un pedazo (*chunk*).

MongoDB intenta distribuirlos de manera equitativa entre los *shards*.

La llave debería tener los siguientes atributos:

- Alta cardinalidad.
- Baja frecuencia de valores.
- No ser monótona.

Para cuando no es posible cumplirlos, existe otra estrategia llamada hashed sharding.

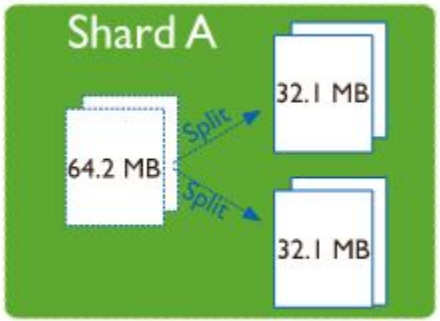


# Chunks

Los *chunks* consisten es subconjuntos de datos. Poseen un tamaño máximo configurable (64 MB por defecto).

MongoDB despedaza el *chunk* en caso de crezca más allá del tamaño máximo. En el caso opuesto puede unirlos.

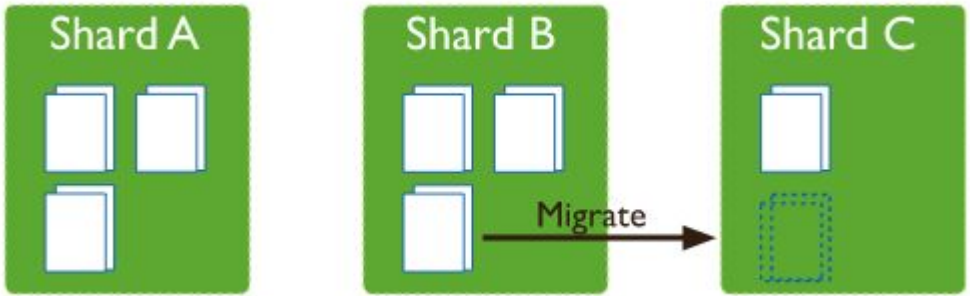
Ambas operaciones llevan a una **distribución despareja** de la colección a los largo de los fragmentos.



# Balanceador

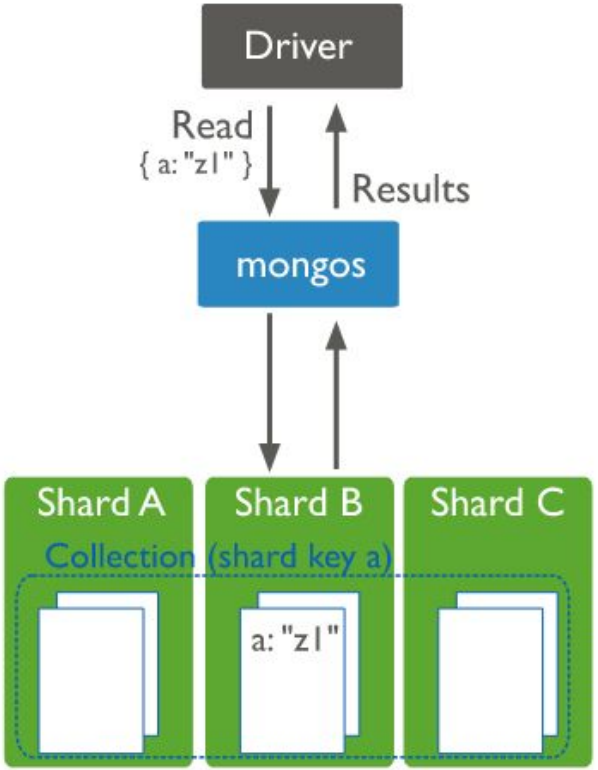
El balanceador es un proceso de fondo que administra las migraciones de *chunks*.

Si la diferencia en el número de pedazos entre el *shard* más ocupado y el *shard* más vacío supera determinado umbral, el balanceador realiza migraciones para distribuir los datos de manera equitativa.

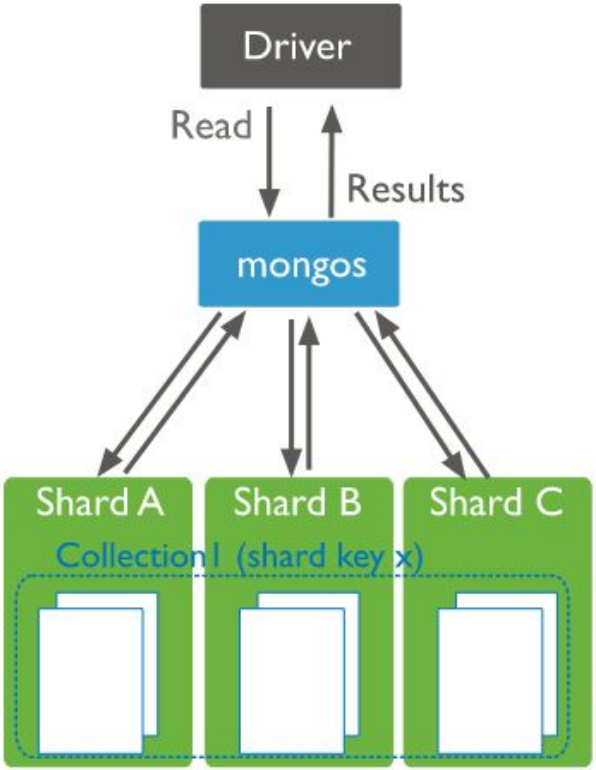


## Operaciones dirigidas

En un entorno fragmentado, las consultas más rápidas son aquellas que el enrutador puede dirigir a un único *shard* utilizando la *shard key*.



Para aquellas consultas que no incluyen la *shard key*, el enrutador debe consultar a todos los fragmentos.



# Teorema CAP

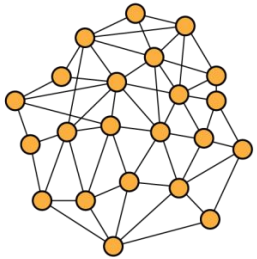
Dice que es **imposible para una base de datos distribuida** cumplir simultáneamente con más dos de las siguientes tres garantías:

- Consistencia
- Disponibilidad
- Tolerancia a la partición

Fuente: https://cryptographics.info/cryptographics/blockchain/cap-theorem

## C : Consistency

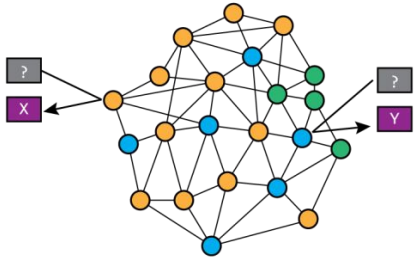
At any given time, all nodes in the network have exactly the same (most recent) value.



= Value: X @ 2018-05-03T08:52:40

## A : Availability

Every request to the network receives a response, though without any guarantee that returned data is the most recent.



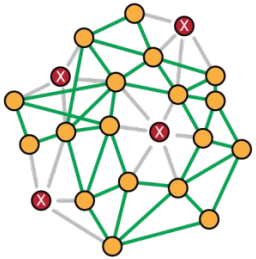

= Value: X @ 2018-05-03T08:52:40

= Value: Z @ 2018-05-03T08:32:58

= Value: Y @ 2018-05-03T07:12:12

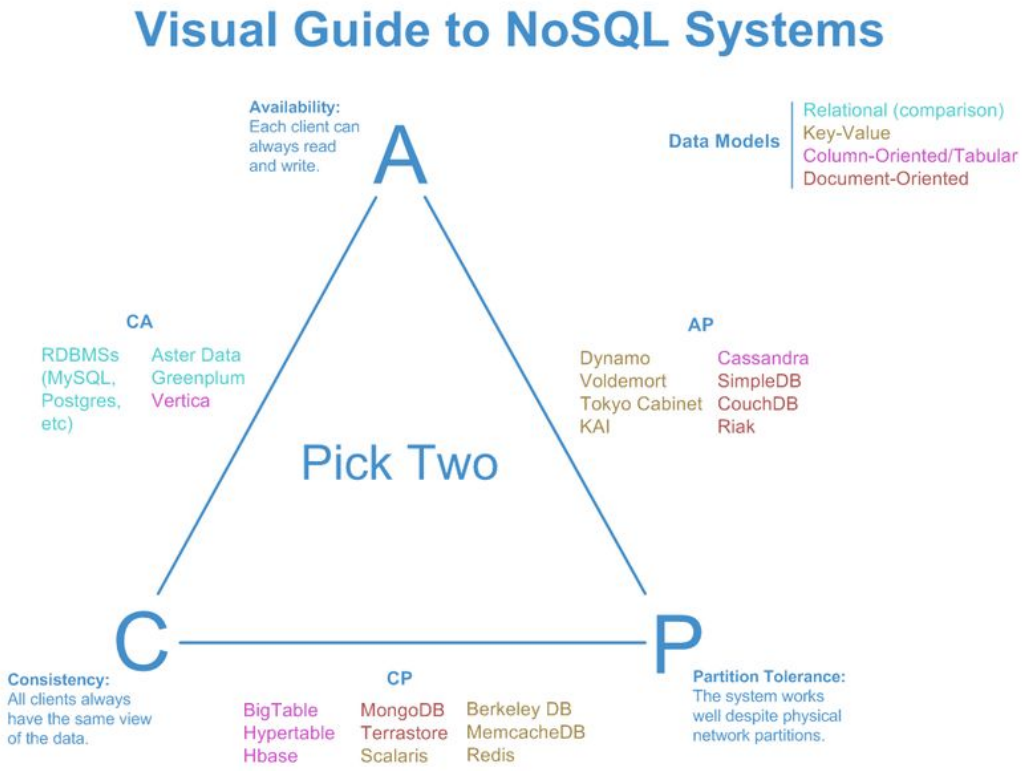
## P : Partition tolerance

The network continues to operate, even if an arbitrary number of nodes are failing.



Cuando sucede una **partición de red** (desconexión de nodos) es necesario decidir:

- Cancelar la operación, por ende perder disponibilidad pero asegurar la consistencia.
- Proceder con la operación, mantener la disponibilidad a riesgo de inconsistencias.



## MongoDB

- Por defecto, todas las operaciones de lectura y escritura van al nodo primario — **consistencia fuerte**.
- Maneja fallas en la red, guardando la misma data en nodos secundarios — **tolerancia a la partición**.
- Compromete la **disponibilidad** durante la votación de un nuevo nodo primario.

El *Domain Name System* (DNS) de internet es un buen ejemplo de sistema de alta disponibilidad y consistencia eventual.

# Características de las bases NoSQL

Naturaleza distribuida

- Escalabilidad horizontal (almacenamiento y cómputo)
- Tolerancia a la partición
- Disponibilidad

Modelos de datos flexibles

- Consultas más rápidas (datos que se acceden juntos se guardan juntos)
- Facilidad de desarrollo

## Posibles inconvenientes

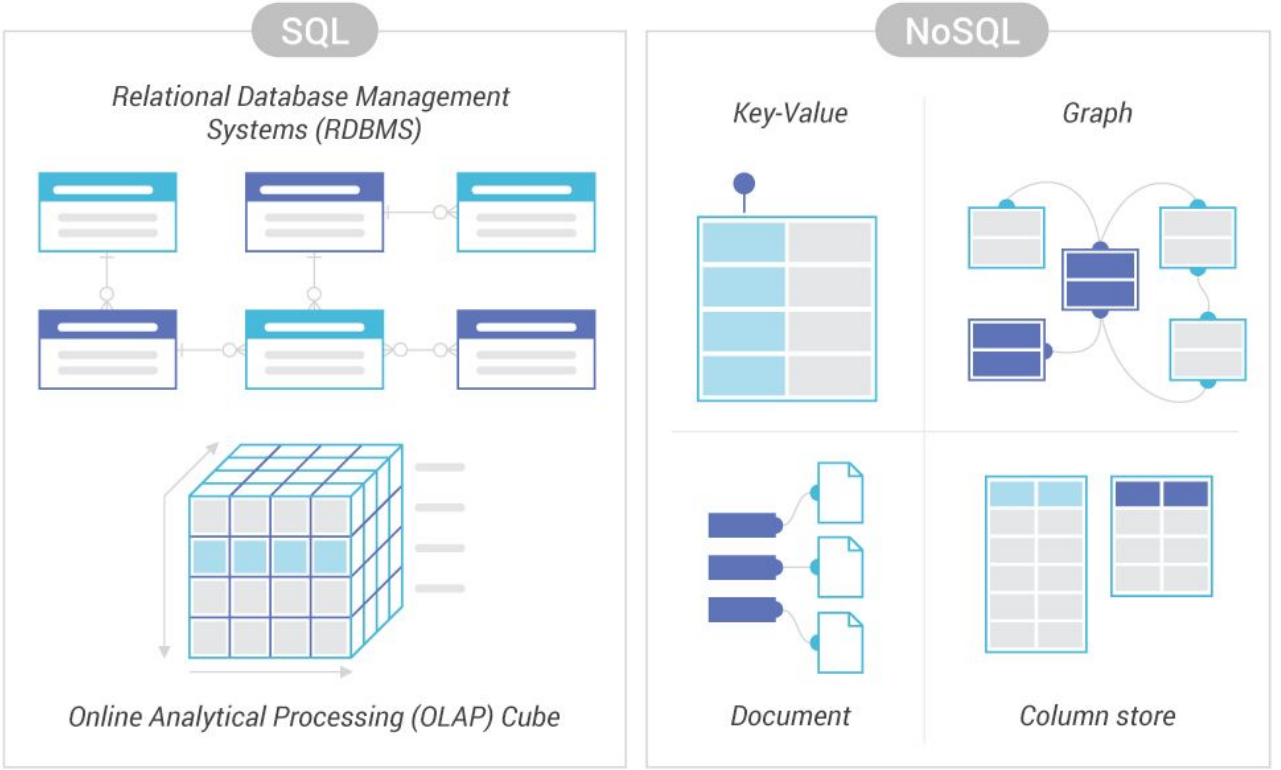
- Falta de transacciones entre múltiples registros.
- Duplicación de información.

# Tipos de bases NoSQL

Hay cuatro grandes tipos:

- documentos,
- llave-valor,
- columnares,
- grafos.

*Fuente*: https://itsmecevi.github.io/dwbi



## Documentos

Almacenan colecciones de documentos similares a objetos JSON (*JavaScript object notation*). Estos objetos suelen alinearse con los objetos que existen en el código de la aplicación.

En estructura, los documentos son similares pero no necesariamente iguales. Es posible trabajar con estructuras complejas (documentos anidados).

Estas bases permiten recuperar documentos y generar índices en base a su contenido; pueden procesar consultas complejas.

*Fuente*: https://www.mongodb.com/document-databases

# Relational

| ID | first_name | last_name | cell | city | year_of_birth | location_x | location_y |
|---|---|---|---|---|---|---|---|
| 1 | 'Mary' | 'Jones' | '516-555-2048' | 'Long Island' | 1986 | '-73.9876' | '40.7574' |

| ID | user_id | profession |
|---|---|---|
| 10 | 1 | 'Developer' |
| 11 | 1 | 'Engineer' |

| ID | user_id | name | version |
|---|---|---|---|
| 20 | 1 | 'MyApp' | 1.0.4 |
| 21 | 1 | 'DocFinder' | 2.5.7 |

| ID | user_id | make | year |
|---|---|---|---|
| 30 | 1 | 'Bentley' | 1973 |
| 31 | 1 | 'Rolls Royce' | 1965 |

# MongoDB

```
{
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
        type: "Point",
        coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
 { name: "MyApp",
   version: 1.0.4 },
 { name: "DocFinder",
   version: 2.5.7 }
 ],
cars: [
   { make: "Bentley",
     year: 1973 },
   { make: "Rolls Royce",
     year: 1965 }
 ]
}
```

## Casos de uso

Las bases de datos de documentos están pensadas para datos con estructura compleja y variable.

- Uso general
- Aplicaciones web

## Bases populares

- MongoDB
- CouchDB
- DynamoDB (AWS)
- Firestore (GCP)

## Llave-valor

Almacenan colecciones de pares llave-valor, como un diccionario.

La llave funciona como identificador único, los valores pueden ser de **cualquier tipo**: estas bases tratan a los datos que contienen como BLOBs (*binary large object*) — 1s y 0s sin estructura. Es responsabilidad de la aplicación entender cómo interpretarlos.

Debido a que los BLOBs son opacos, estas bases

- no realizan búsquedas en base al contenido de los valores,
- ni establecen otro índice más que para las llaves.

*Fuente*: https://redislabs.com/blog/get-sql-like-experience-redis

**Table**

Products

| ID | Name | Description | Price |
|---|---|---|---|
| 10001 | AAAA | Description of AAAA | 100 |
| 10002 | BBBB | Description of BBBB | 200 |
| 10003 | CCCC | Description of CCCC | 200 |
| . | . | . | . |
| . | . | . | . |
| 10999 | XXYZ | Description of XXYZ | 500 |

**Hash**

```
product:10999
  name  XXYZ
product:10003
product:10002
product:10001
  name  AAAA
  desc  Description of AAAA
  price 100
```

**Sorted Set**

| product_list | |
|---|---|
| 10001 | product:10001 |
| 10002 | product:10002 |
| 10003 | product:10003 |
| . | . |
| . | . |
| 10999 | product:10999 |

**Sorted Set**

| product_price | |
|---|---|
| 100 | product:10001 |
| 200 | product:10002 |
| 200 | product:10003 |
| . | . |
| . | . |
| 500 | product:10999 |

## Casos de uso

Las bases de datos llave-valor son la opción más eficiente cuando las consultas para recuperar los datos no son complejas.

- Almacenamiento caché
- Colas de mensajes
- Manejo de sesiones

## Bases populares

- Redis
- Riak
- DynamoDB (AWS)
- Firestore (GCP)

## Columnares

En apariencia muy similares a las bases de datos relacionales; en vez de agrupar las columnas en tablas, **cada columna es almacenada por separado**, lo que permite leer solo las columnas que se necesitan en vez de toda la tabla.

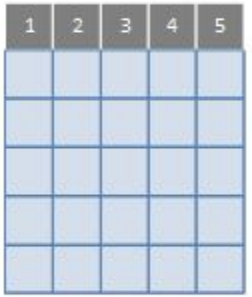
Las columnas guardan datos del **mismo tipo**. Las filas pueden tener **distintas columnas**, las tablas son esparzas.

Son consideradas bases llave-valor de dos dimensiones. Los ids de las filas y los nombres de las columnas son los únicos índices; los valores son opacos.

*Fuente*:
https://bi-insider.com/portfolio-item/techniques-of-data-warehouse-performance-optimization

# Columnar Data Storage

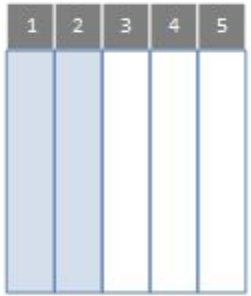
## Row-Based Data Storage




```
SELECT  Column1,
        Column2
FROM    Table
```

- Data is stored row-oriented on disk.
- Optimized for insert queries.
- All columns are read from disk in a select query – even if only a subset of columns are needed.
- Unselected columns are disregard after disk read.

## Column-Based Data Storage




```
SELECT  Column1,
        Column2
FROM    Table
```

- Data is stored column-oriented on disk.
- Optimized for read queries.
- Only selected columns are read from disk.
- Unselected columns are not read from disk.

## Casos de uso

Las bases de datos columnares son óptimas para lecturas rápidas de grandes volúmenes de datos y agregaciones de columnas.

- *Big Data*
- Series temporales. Ejemplos: IoT, telemetría, valores mercados.
- *Logs*. Ejemplos: monitoreo de systemas, datos de usuarios.

## Bases populares

- Cassandra
- HBase
- Redshift (AWS)
- Bigtable (GCP)

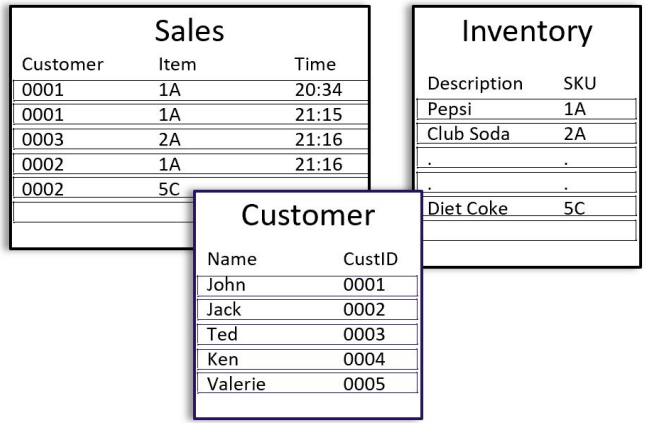
# Grafos

Tienen un parecido a las bases de datos de documentos, con hincapié en las relaciones entre documentos.

Los documentos suelen ser entidades (personas, lugares, cosas); las relaciones son semánticamente relevantes (pertenece a, compró un) y pueden tener atributos (distancia, tiempo, costo).

Ciertas operaciones son mucho más sencillas utilizando estas bases por la manera en la que enlazan y agrupan los datos.

*Fuente*: https://towardsdatascience.com/graph-databases-whats-the-big-deal-ec310b1bc0ed



**Sales**

| Customer | Item | Time |
|---|---|---|
| 0001 | 1A | 20:34 |
| 0001 | 1A | 21:15 |
| 0003 | 2A | 21:16 |
| 0002 | 1A | 21:16 |
| 0002 | 5C | |

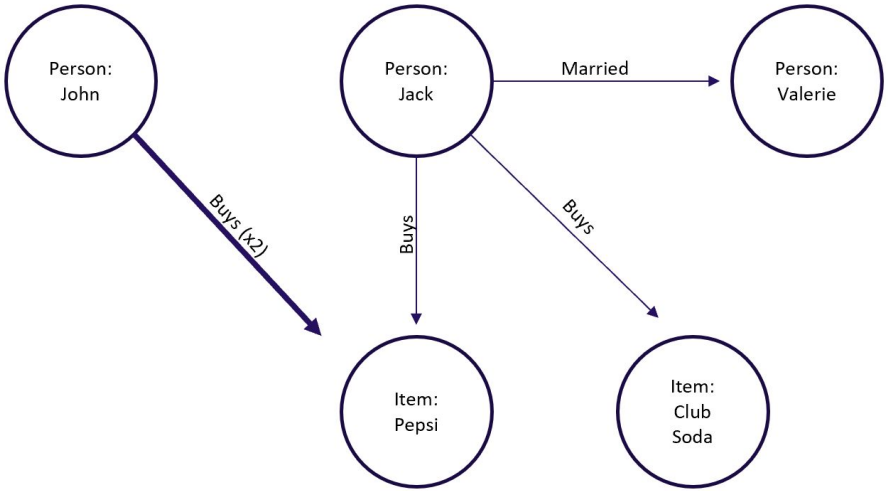
**Inventory**

| Description | SKU |
|---|---|
| Pepsi | 1A |
| Club Soda | 2A |
| . | . |
| . | . |
| Diet Coke | 5C |

**Customer**

| Name | CustID |
|---|---|
| John | 0001 |
| Jack | 0002 |
| Ted | 0003 |
| Ken | 0004 |
| Valerie | 0005 |

Traditional database store data to efficiently store facts, but relationships must be rebuilt with JOINs and other inexact techniques.




Graph databases store both facts and the relationships between the facts, making certain types of analysis more intuitive.

## Casos de uso

Las bases de datos de grafos sobresalen a la hora de "atravesar" una red de documentos en busca de patrones.

- Redes sociales
- Detección de fraude
- Sistemas de recomendación
- Bases de conocimiento, como Wikidata

## Bases populares

- Neo4j
- ArangoDB

# Gracias.

IT BOARDING

BOOTCAMP