



# NoSQL



## Grado en Ingeniería de Sistemas de Información

alvaro.sanchezpicot@ceu.es

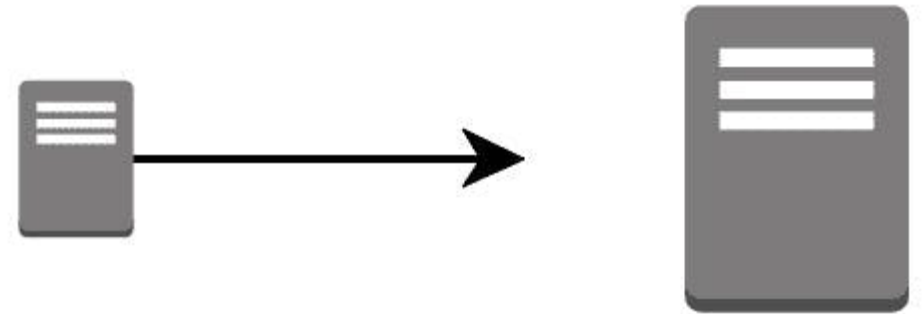
v20240418



# INTRODUCCIÓN

# Introducción

- Las bases de datos relacionales aparecieron en los años 70
- La información se organiza en tablas y relaciones
- Scale up (vertical):
  - Mejorar el equipo
  - Puede ser muy caro



# Introducción

- Las bases de datos no relacionales aparecieron en los años 60
- El término NoSQL
  - Not Only SQL
  - Apareció por primera vez en 1998
  - Gano interés en los 2000 con el auge de Internet
  - 2004: Google Big Table y Amazon DynamoDB
- Soluciones específicas para las necesidades de grandes empresas

# Introducción

- 63% de las empresas gestionan datos superiores a los 50 PB
  - La mitad de la información es desestructurada
- Scale out (horizontal):
  - Añadir más máquinas
  - Problemas de licencias



# Introducción

- Resolver el "impedance mismatch":
  - Conversión de clases y objetos a una tabla
  - Encapsulación
  - Herencia
  - Tipos de datos
  - [Más información](#)

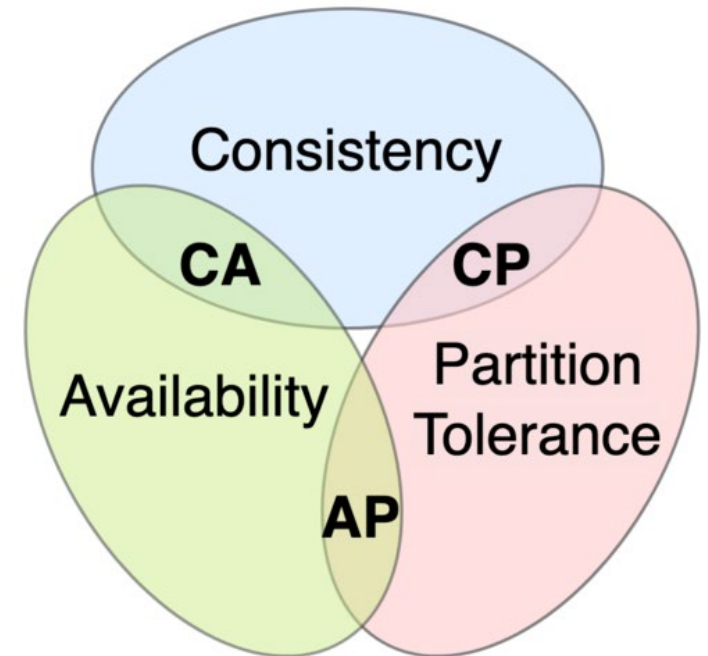
# ACID

Garantizar la validez en caso de fallo:

- Atomicity: la transacción se ejecuta completa o no se ejecuta
- Consistency: las transacciones mantienen la validez
- Isolation: no importa el orden en el que se ejecuten las transacciones
- Durability: cuando la transacción se ha completado se guarda en memoria no volátil

# CAP

- El teorema de Brewer
- Hay que elegir dos de entre:
  - **Consistency**: Las lecturas reciben el valor más reciente o un error
  - **Availability**: Las peticiones reciben una respuesta que no sea un error
  - **Partition tolerance**: El sistema funciona aunque se pierdan paquetes o haya retardos





# BASE

- También formulado por Brewer
- Versión ACID de NoSQL
- Propiedades
  - Basic Availability: el sistema está disponible en caso de fallo
  - Soft state: el estado puede cambiar sin interacciones
  - Eventual consistency: después de un tiempo sin inputs, el sistema será consistente

# SQL vs NOSQL

414 systems in ranking, April 2023

Rank			DBMS	Database Model	Score		
Apr 2023	Mar 2023	Apr 2022			Apr 2023	Mar 2023	Apr 2022
1.	1.	1.	Oracle +	Relational, Multi-model i	1228.28	-33.01	-26.54
2.	2.	2.	MySQL +	Relational, Multi-model i	1157.78	-25.00	-46.38
3.	3.	3.	Microsoft SQL Server +	Relational, Multi-model i	918.52	-3.49	-19.94
4.	4.	4.	PostgreSQL +	Relational, Multi-model i	608.41	-5.41	-6.05
5.	5.	5.	MongoDB +	Document, Multi-model i	441.90	-16.89	-41.48
6.	6.	6.	Redis +	Key-value, Multi-model i	173.55	+1.10	-4.05
7.	7.	↑ 8.	IBM Db2	Relational, Multi-model i	145.49	+2.57	-14.97
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model i	141.08	+2.01	-19.76
9.	9.	↑ 10.	SQLite +	Relational	134.54	+0.72	+1.75
10.	10.	↓ 9.	Microsoft Access	Relational	131.37	-0.69	-11.41
11.	↑ 12.	11.	Cassandra +	Wide column	111.81	-1.98	-10.19
12.	↓ 11.	↑ 14.	Snowflake +	Relational	111.12	-3.27	+21.68

[Source](#)



# NOSQL

# NoSQL

## Características NoSQL:

- No hay un modelo relacional
- No usan SQL
  - Alguno puede tener un lenguaje parecido (ej. Cassandra's CQL)
- Pensado para sistemas distribuidos (clusters)
- Schemaless: puedes añadir datos sin tener que definir primero la estructura
- Open source

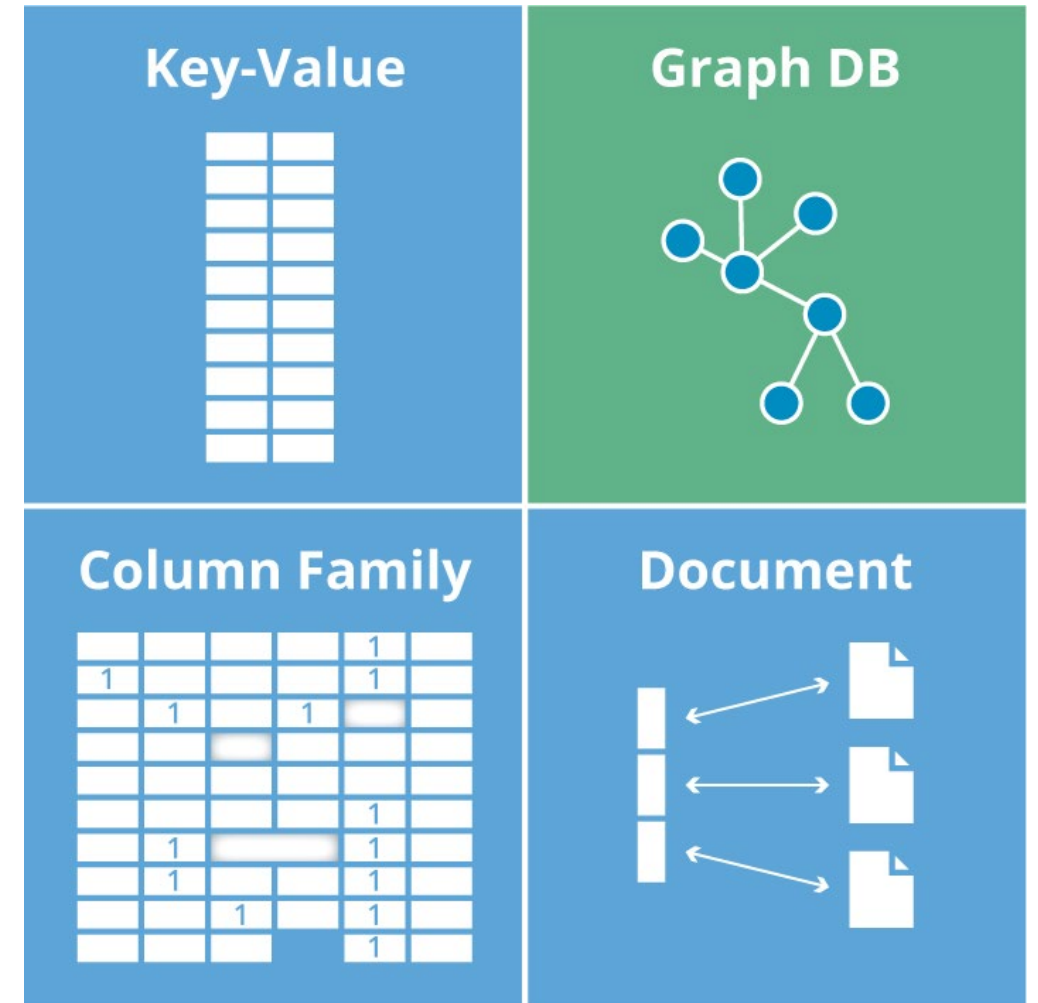
# NoSQL

- The pace of development with NoSQL databases can be much faster than with a SQL database
- The structure of many different forms of data is more easily handled and evolved with a NoSQL database
- The amount of data in many applications cannot be served affordably by a SQL database
- The scale of traffic and need for zero downtime cannot be handled by SQL
- New application paradigms can be more easily supported

# NoSQL

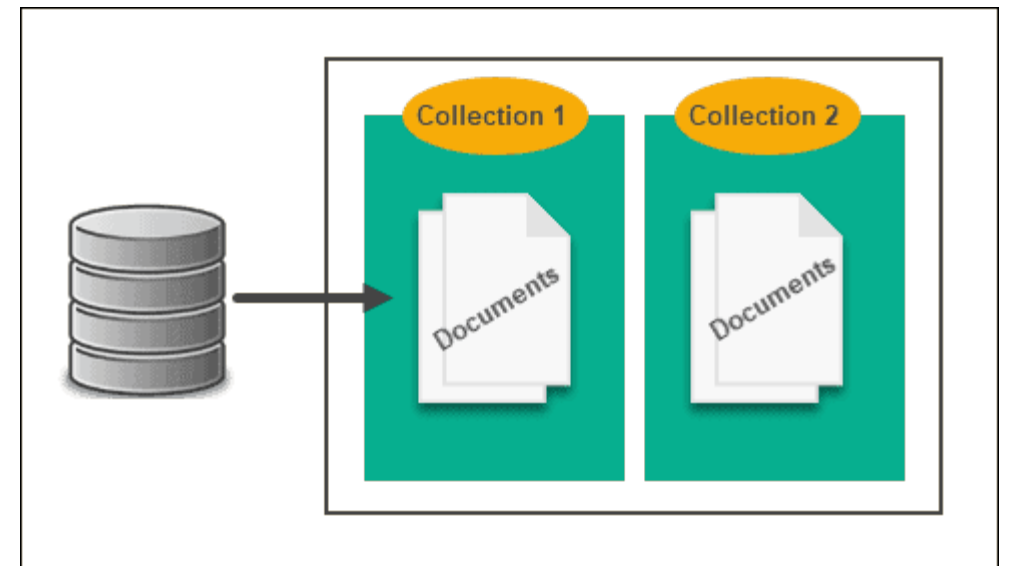
Tipos de bases de datos NoSQL:

- Document
- Key-value
- Wide-column
- Graph



# NoSQL – Document Databases

- Almacena datos semiestructurados (JSON, BSON o XML)
- Se pueden anidar los documentos
- Elementos específicos se pueden indexar para acelerar el acceso
- Casos de uso:
  - Ecommerce platforms
  - Trading platforms
- Implementaciones:
  - MongoDB
  - AWS DynamoDB



# NoSQL – Key-Value Stores

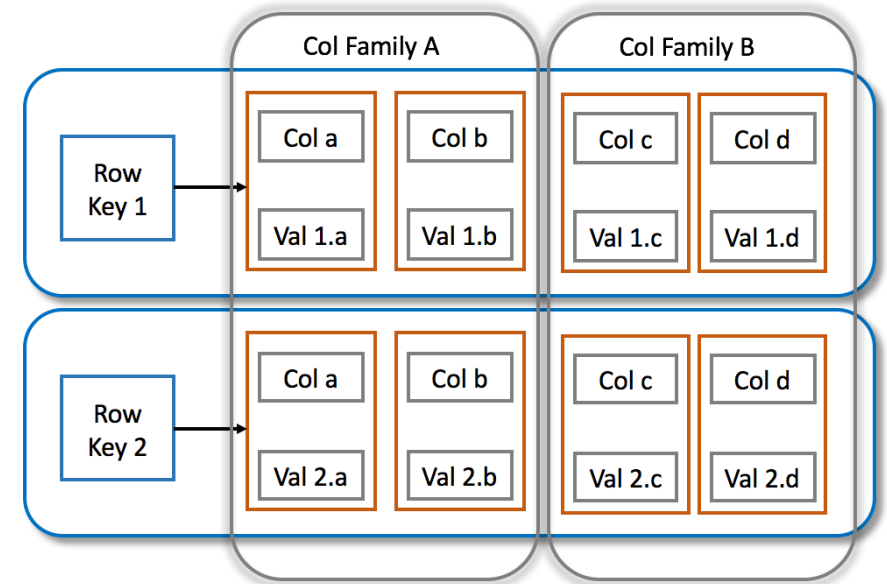
- Pareja clave-valor (hash table o diccionario)
- Equivalente a una tabla SQL con dos columnas
- Casos de uso:
  - Shopping carts
  - User preferences
  - User profiles
- Implementaciones:
  - AWS DynamoDB
  - Redis

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623



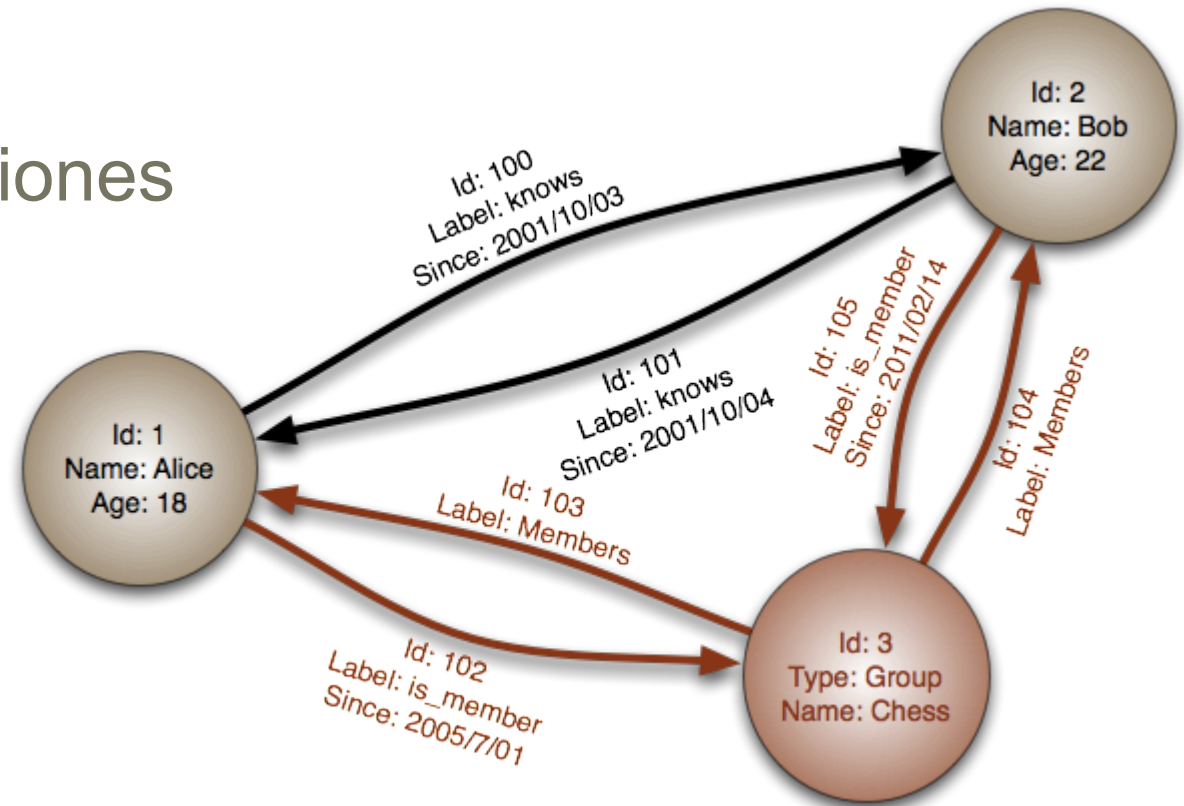
# NoSQL – Wide-column store

- La información se organiza en columnas
- El nombre y el formato de las columnas puede variar entre filas
- Cada columna se almacena de forma separada en el disco
- Casos de uso:
  - Analytics
- Implementaciones:
  - Cassandra
  - Google Bigtable
  - ScyllaDB



# NoSQL – Graph Databases

- Se centra en la relación entre los elementos
- Cada elemento es un nodo
- Las relaciones son las conexiones
- Casos de uso:
  - Fraud detection
  - Social networks
- Implementaciones:
  - Amazon Neptune
  - Neo4j



# NoSQL

Polyglot persistence:

- Usar diferentes tecnologías en función de las necesidades
- No usar una misma base de datos para todo
- Basado en Polyglot programming



**MONGODB**



CEU

*Universidad  
San Pablo*

# MongoDB

- Base de datos NoSQL
- Document DB
- Gratis
- Modelo de negocio:
  - Entrenamiento
  - Soporte
  - DB as a service (Atlas)
- Casi open source ([usa la SSPL](#))
- [Link](#)

# MongoDB – Características

- Ad-hoc queries (MQL)
- Índices
- Replicación
- Load balancing
- Pipelines de agregación
- Transacciones

# MongoDB – Conceptos

- **Document:** a way to organize and store data as a set of field-value pairs
- **Field:** a unique identifier for a datapoint
- **Value:** data related to a given identifier
- **Collection:**
  - An organized store of documents in MongoDB
  - Usually with common fields between documents
  - There can be many collections per database
  - There can be many documents per collection

# MongoDB – Conceptos

- **Replica Set:**
  - A few connected machines that store the same data
  - They ensure that if something happens to one of the machines the data will remain intact
  - Comes from the word replicate - to copy something
- **Instance:** a single machine locally or in the cloud, running a certain software, in our case it is the MongoDB database
- **Cluster:** group of servers that store your data



# MongoDB – Instalación



- [MongoDB Community Server](#)
  - [Instalación en MacOS](#)
  - [MongoDB Compass](#) (incluido en la instalación básica)
- [MongoDB Shell](#)
- [MongoDB Database Tools](#)
  - [Añadir las al PATH](#)

# MongoDB – Configuración

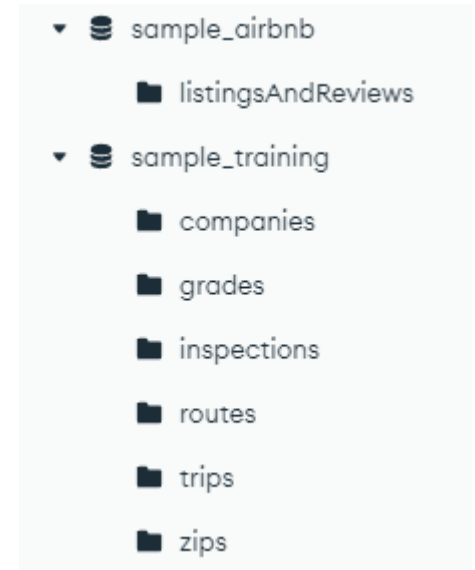
- Clonar [este repositorio](#)

Dataset Name	Description	Collections
<a href="#">Sample AirBnB Listings Dataset</a>	Contains details on AirBnB listings.	listingsAndReviews
<a href="#">Sample Analytics Dataset</a>	Contains training data for a mock financial services application.	accounts, customers, transactions
<a href="#">Sample Geospatial Dataset</a>	Contains shipwreck data.	shipwrecks
<a href="#">Sample Mflix Dataset</a>	Contains movie data.	comments, movies, theaters, users
<a href="#">Sample Supply Store Dataset</a>	Contains data from a mock office supply store.	sales
<a href="#">Sample Training Dataset</a>	Contains MongoDB training services dataset.	companies, grades, inspection, posts, routes, stories, trips, tweets, zips
<a href="#">Sample Weather Dataset</a>	Contains detailed weather reports.	data



# MongoDB – Configuración

- Abrir MongoDB Compass e importar las siguientes colecciones:
  - sample\_airbnb:
    - listingsAndReviews
  - sample\_training:
    - companies
    - grades
    - inspections
    - routes
    - trips
    - zips



# MongoDB – Mongo Shell

`cls`: Limpiar la consola

`show dbs`: Mostrar las bases de datos

`db`: Devuelve el nombre de la base de datos activa

`use <nombre_db>`: Cambiar a la base de datos <nombre\_db>

# MongoDB – Collections

`show collections`

- Muestra las colecciones de la DB activa

`db.createCollection(<name>, <options>)`

- Crea una nueva colección
- Al insertar un elemento, se crea automáticamente si no existe
- [options](#)

`db.<collection>.drop()`

- Elimina la colección

# MongoDB – Read

## Read:

`db.<collection_name>.findOne()`

- Devuelve un elemento de la colección

`db.<collection_name>.find()`

- Devuelve todos los elementos de la colección
- Si hay más de 20, devuelve 20 y un puntero para recorrerlos  
`it`
- Avanzar a los siguientes 20 elementos



# MongoDB – Read

Read (cont.):

```
db.<collection_name>.find().pretty()
```

- Devuelve los elementos para que sea más fácil leerlos

```
db.<collection_name>.find().count()
```

- Devuelve el número de elementos

```
db.<collection_name>.find(query)
```

- Se puede añadir una query para filtrar la búsqueda
- Ej:

```
db.zips.find({"state":"AL"}).count()
```

# MongoDB – Insert

## Insert:

`db.<collection_name>.insertOne(<document>)`

- Inserta el documento en la colección

`db.<collection_name>.insertMany([<document 1>,<document 2>,...])`

- Inserta múltiples documentos



# MongoDB – Insert

- Al insertar un elemento, si no existe la colección o la base de datos, se crean en ese momento
- Cada documento de una colección tiene que tener un `_id` único
- Si no lo indicamos nosotros al insertar, se genera un [ObjectId](#)
- Si intentamos insertar uno que ya existe, salta una excepción:
  - E11000 duplicate key error collection
  - Si estábamos insertando múltiples, no se insertan los que falten
    - Podemos añadir la opción `{ "ordered": false }` para que sí se inserten

# MongoDB – Update

## Update:

```
db.<collection_name>.updateOne(filter, update, options)
```

```
db.<collection_name>.updateMany(filter, update, options)
```

- Actualiza el valor de un campo
- `filter`: selection criteria
- `update`: modificación a aplicar

```
db.<collection_name>.replaceOne(filter, replacement, options)
```

- Reemplaza un documento por otro
- `replacement`: nuevo documento

# MongoDB – Update

Operadores para update:

- `$set`: Para cambiar un valor
- `$unset`: Para eliminar un campo
- `$inc`: Para incrementar un número
- `$push`: Para añadir un elemento a un array
- [Más operadores](#)

Ej: `db.zips.updateOne({"zip": "12534"}, {"$set": {"pop": 17630}})`

# MongoDB – Delete

## Delete:

```
db.<collection_name>.deleteOne(filter, options)
```

```
db.<collection_name>.deleteMany(filter, options)
```

- Elimina los documentos que cumplan el filtro
- `filter`: criterio de selección

# MongoDB

## Comparadores:

- \$eq: = (default)
- \$ne: !=
- \$gt: >
- \$lt: <
- \$gte: >=
- \$lte: <=

# MongoDB

## Operadores lógicos:

- \$and (default)
- \$not
- \$nor
- \$or

# MongoDB

{<field>: {<operator>: <value>}}

Ejemplo:

- Documentos cuya duración del viaje sea menor de 70 segundos y el tipo de usuario no sea subscriptor

```
db.trips.find({"tripduration": { "$lte" : 70 },  
              "usertype": { "$ne": "Subscriber" } }).pretty()
```

# MongoDB

{<field>: {<operator>: <value>}}

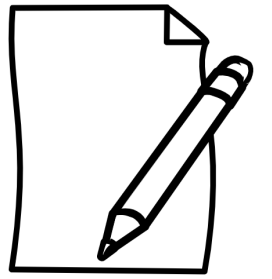
Ejemplo:

- Documentos donde los aviones CR2 o A81 aterrizaron o despegaron del aeropuerto KZN

```
db.routes.find({ "$and": [  
  { "$or" :[ { "dst_airport": "KZN" }, { "src_airport": "KZN" } ] },  
  { "$or" :[ { "airplane": "CR2" }, { "airplane": "A81" } ] }  
] }).pretty()
```

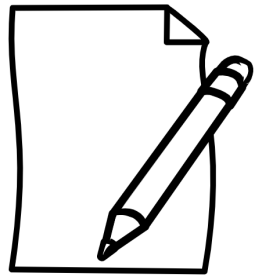


# MongoDB – EJ1



1. En `sample_training.zip` ¿Cuántas colecciones tienen menos de 1000 personas en el campo `pop`? (sol. 8065)
2. En `sample_training.trips` ¿Cuál es la diferencia entre la gente que nació en 1998 y la que nació después de 1998? (sol. 6)
3. En `sample_training.routes` ¿Cuántas rutas tienen al menos una parada? (sol. 11)

# MongoDB – EJ1



4. En `sample_training.inspections` ¿Cuántos negocios tienen un resultado de inspección "Out of Business" y pertenecen al sector "Home Improvement Contractor - 100"? (sol. 4)
5. En `sample_training.inspections` ¿Cuántos documentos hay con fecha de inspección "Feb 20 2015" o "Feb 21 2015" y cuyo sector no sea "Cigarette Retail Dealer - 127"? (sol. 204)

# MongoDB

## Expressive \$expr

- Allows us to use variables and conditional statements
- {"\$expr": { <expression> }}
- "\$fieldname" → referencia el valor de ese field
- Ej. 1: Número de documentos de sample\_training.trips donde el viaje empieza y termina en la misma estación:

```
db.trips.find({ "$expr": { "$eq": [ "$end station id",  
"$start station id"] } }).count()
```



# MongoDB

Ej. 2: Find all documents where the trip lasted longer than 1200 seconds, and started and ended at the same station:

```
db.trips.find({ "$expr":  
  { "$and": [  
    { "$gt": [ "$tripduration", 1200 ] },  
    { "$eq": [ "$end station id", "$start station id" ] }  
  ] } }).count()
```

# MongoDB

- Array Operators
- \$push: añadir un elemento a un array (crearlo si no existe)  
`db.students.insertOne({_id: 1, scores: [44, 78, 38, 80]})`  
`db.students.updateOne({ _id: 1 }, {$push: {scores: 89}})`

# MongoDB

- En sample\_training.listingsAndReviews:  
`{"amenities": "Shampoo"}` →
  - Permite buscar en el array  
`{"amenities": ["Shampoo"]}` →
  - Busca exactamente ese array
  - Recordatorio, en los arrays importa el orden

# MongoDB

"\$all": [...]

- Para buscar arrays que contengan al menos esos elementos
- Ej.:

```
db.listingsAndReviews.find({ "amenities": {  
  "$all": [ "Internet", "Wifi", "Kitchen", "Heating",  
    "Family/kid friendly", "Washer", "Dryer", "Essentials"]  
}})
```

# MongoDB

`"$size": number`

- Para especificar un tamaño exacto del array
- Ej.:

```
db.listingsAndReviews.find({"amenities":{"$size":55}}).count()
```



# MongoDB – Projection

```
db.<collection>.find({<query>}, {<projection>})
```

- Permite obtener sólo los campos que queremos
- {<field1>: 1, <field2>:1,...} →
  - Se incluyen esos campos (además de \_id), se excluyen el resto
- {<field1>: 0, <field2>: 0,...} →
  - Se excluyen esos campos, se incluyen todos los demás
- No se pueden mezclar 0s y 1s excepto para excluir \_id

# MongoDB

- Ej.:

```
db.listingsAndReviews.find({ "amenities": "Wifi" },  
    { "price": 1, "address": 1})
```

```
db.listingsAndReviews.find({ "amenities": "Wifi" },  
    { "price": 1, "address": 1, "_id": 0 })
```



```
db.listingsAndReviews.find({ "amenities": "Wifi" },  
    { "price": 1, "address": 1, "maximum_nights":0 })
```

# MongoDB

## "\$elemMatch"

- Usado con arrays, los proyecta sólo si tienen un elemento que cumpla el criterio
- Ej.:

```
db.grades.find({"class_id": 431},  
  {"scores": {"$elemMatch": {"score": {"$gt": 85}}}})
```

# MongoDB

"<field1>.<field2>.<field3>..."

- [Dot notation](#)
- Usada para recorrer subdocumentos
- Se pueden usar números para las posiciones de los arrays
- Tiene que estar entre comillas
- Ej.:

```
db.companies.find(  
  { "relationships.0.person.last_name": "Zuckerberg"},  
  { "name": 1 })
```

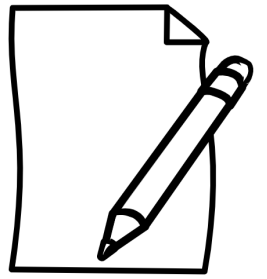
# MongoDB

"\$regex"

- Para buscar patrones en Strings usando expresiones regulares
- [Más info](#)
- Ej.:

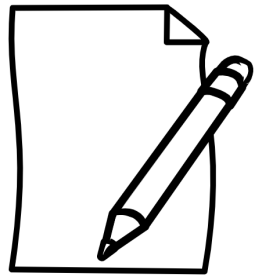
```
db.companies.find({  
    "relationships.0.person.first_name": "Mark",  
    "relationships.0.title": {"$regex": "CEO" }  
}, { "name": 1 }  
)
```

# MongoDB – EJ2



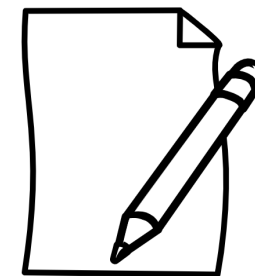
1. En `sample_training.companies`, ¿cuántas empresas tienen más empleados que el año en el que se fundaron? (sol. 324)
2. En `sample_training.companies`, ¿en cuántas empresas coinciden su `permalink` con su `twitter_username`? (sol. 1299)
3. En `sample_airbnb.listingsAndReviews`, ¿cuál es el nombre del alojamiento en el que pueden estar más de 6 personas alojadas y tiene exactamente 50 reviews? (sol. Sunset Beach Lodge Retreat)

## MongoDB – EJ2



4. En `sample_airbnb.listingsAndReviews`, ¿cuántos documentos tienen el "property\_type" "House" e incluyen "Changing table" como una de las "amenities"? (sol. 11)
5. En `sample_training.companies`, ¿Cuántas empresas tienen oficinas en Seattle? (sol. 117)
6. En `sample_training.companies`, haga una query que devuelva únicamente el nombre de las empresas que tengan exactamente 8 "funding\_rounds"

# MongoDB – EJ2



7. En `sample_training.trips`, ¿cuántos viajes empiezan en estaciones que están al oeste de la longitud -74? (sol. 1928)

Nota 1: Hacia el oeste la longitud decrece

Nota 2: el formato es `<field_name>: [ <longitud>, <latitud> ]`

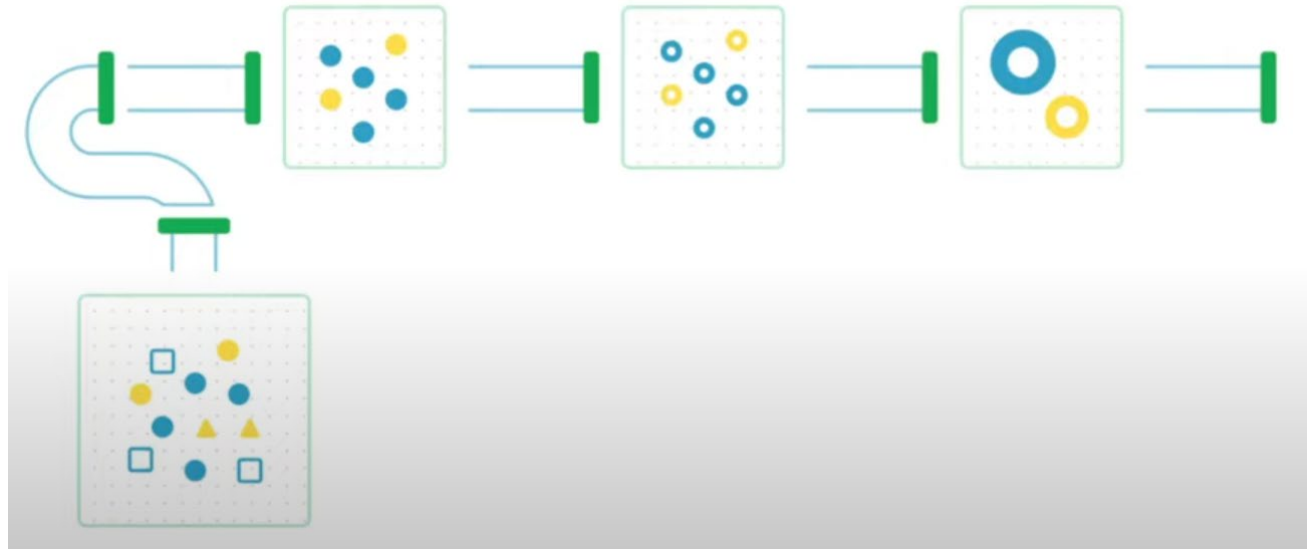
8. En `sample_training.inspections`, ¿cuántas inspecciones se llevaron a cabo en la ciudad de "NEW YORK"? (sol. 18279)

9. En `sample_airbnb.listingsAndReviews`, haga una query que devuelva el nombre y la dirección de los alojamientos que tengan "Internet" como primer elemento de "amenities"



# MongoDB – Aggregation

- Pipeline para realizar operaciones sobre los datos
- Compuesto de varias etapas (stages)
- [Operadores](#)



# MongoDB – Aggregation

```
db.<collection>.aggregate( [ { <stage> }, ... ] )
```

- Una o más etapas aplicadas en orden
- Stages:
  - \$match
  - \$project
  - \$group

# MongoDB – Aggregation

`{ $match: { <query> } }`

- Devuelve todos los documentos que cumplan la query
- Las queries son equivalentes a las de lectura (los find)
- [Más información](#)
- Ej.:

```
db.listingsAndReviews.aggregate(  
  [{ "$match": { "amenities": "Wifi" } }]  
)
```

# MongoDB – Aggregation

```
{ $project: { <specification(s)> } }
```

- Devuelve todos los documentos que cumplan la query
- Las queries son equivalentes a las de lectura (los find)
- [Más información](#)
- Ej.:

```
db.listingsAndReviews.aggregate([  
  { "$match": { "amenities": "Wifi" } },  
  { "$project": { "price": 1, "address": 1 } } ] )
```

# MongoDB – Aggregation

```
{ $group: {  
  _id: <expression>, // Group By Expression  
  <field1>: { <accumulator1> : <expression1> }, ...  
}}
```

- Agrupa todos los elementos que sean iguales
- <field>:
  - Operación a realizar
  - Ej.: \$count, \$max, \$min, \$sum...
- [Más información](#)



# MongoDB – Aggregation

\$group

- Ej.:

```
db.listingsAndReviews.aggregate([  
  {"$project": {"address": 1, "_id": 0 }},  
  {"$group": {"_id": "$address.country" } }])
```

```
db.listingsAndReviews.aggregate([  
  {"$project": {"address": 1, "_id": 0 }},  
  {"$group": {"_id": "$address.country", "count": {"$sum": 1  
} } } ] )
```

# MongoDB – Cursor methods

- Métodos que se aplican a un cursor (por ejemplo lo que devuelve un find())
- [Listado](#)
- Ej.:
  - count()
  - limit()
  - pretty()
  - skip()
  - sort()

# MongoDB – Cursor methods

limit(<number>)

- Número de elementos que se devuelven
- 0: no hay límite
- Ej.:

```
db.zipps.find().limit(10)
```



# MongoDB – Cursor methods

skip(<offset>)

- Número de elementos al principio que ignoro
- Útil junto con `limit()` para hacer paginado
- Ej.:

```
db.zips.find().skip(10)
```

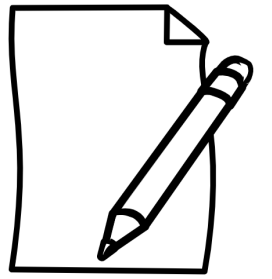
# MongoDB – Cursor methods

sort(<field1>: <value1>, <field2>: <value2>, ...)

- Ordenar los elementos
- Si no hay un campo único, el resultado puede ser inconsistente
- Máximo 32 fields
- value:
  - 1 para orden ascendente
  - -1 para orden descendente
- Ej.:

```
db.zips.find().sort({ "pop": -1 }).limit(5)
```

# MongoDB – EJ3



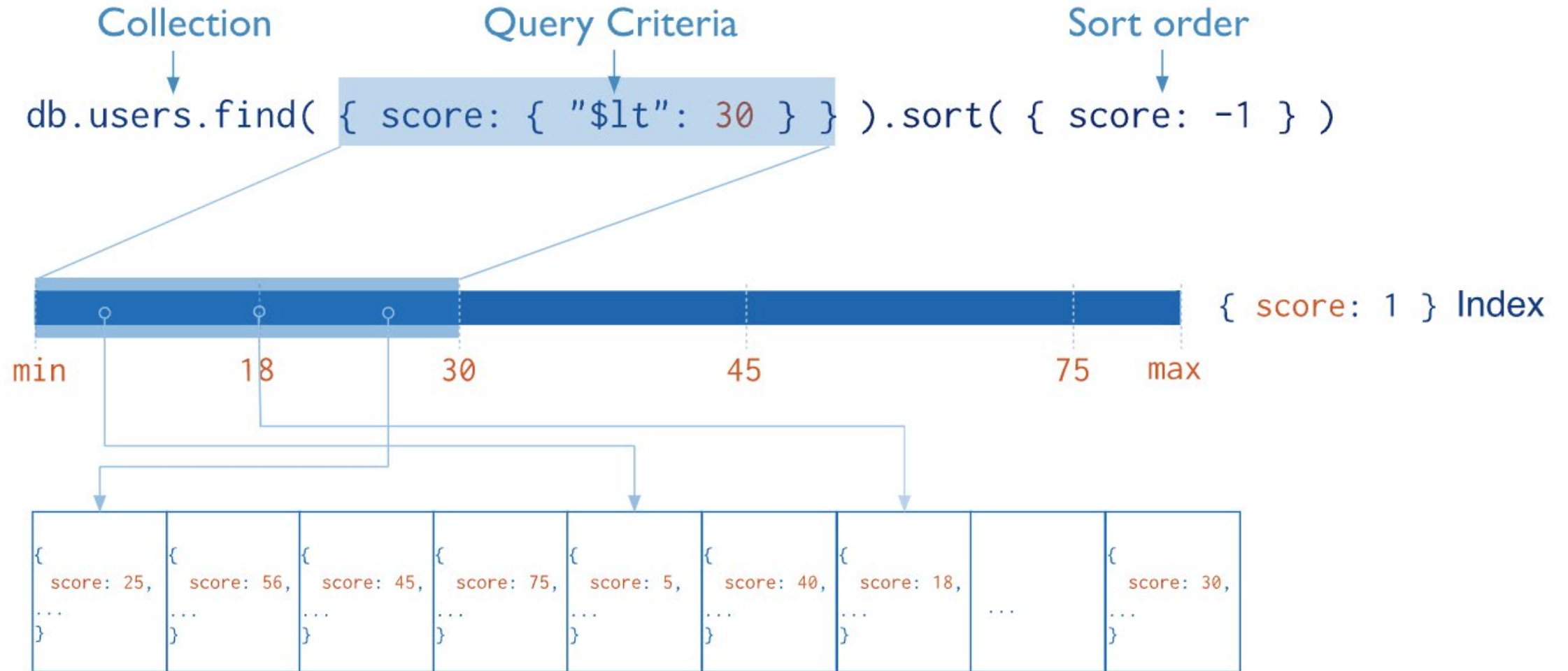
1. En `sample_airbnb.listingsAndReviews`, ¿qué "room types" existen?
2. En `sample_training.companies`, haga una query que devuelva el nombre y el año en el que se fundaron las 5 compañías más antiguas.
3. En `sample_training.trips`, ¿en qué año nació el ciclista más joven? (sol. 1999)

# MongoDB – Índices

`db.<collection>.createIndex()`

- Crea un índice para facilitar las búsquedas
- Hay creado un índice por defecto para `_id`
- [Más información](#)

# MongoDB – Índices



# MongoDB – Upsert

```
db.<collection>.updateOne({<query>},{<update>},{ "upsert":true})
```

- Híbrido entre update e insert
  - Si hay una coincidencia se actualiza
  - Si no hay una coincidencia se inserta
- Usarlo sólo cuando se necesita
- [Más información](#)

# MongoDB – Upsert

```
db.iot.updateOne({  
  "sensor": r.sensor,  
  "date": r.date,  
  "valcount": { "$lt": 48 }  
}, {  
  "$push": { "readings": { "v": r.value, "t": r.time } },  
  "$inc": { "valcount": 1, "total": r.value } },  
{ "upsert": true })
```

# MongoDB – Upsert

```
db.iot.updateOne({  
  "sensor": "SENS9123987",  
  "date": "2022-05-10",  
  "valcount": { "$lt": 3 }  
}, {  
  "$push": { "readings": { "v": 89, "t": ISODate() } },  
  "$inc": { "valcount": 1, "total": 89 } },  
{ "upsert": true })
```



# MongoDB

- Y muchos más comandos:
  - \$exists: para obtener los documentos que contienen (o no) un campo.
  - \$unwind

# MongoDB – Modelado de datos

- A way to organize fields in a document to support your application performance and querying capabilities.
- Data is stored in the way that it is used
- Data used together should be stored together

# MongoDB – Validación

- MongoDB permite usar JSON Schema
- Se puede configurar al crear una colección  

```
db.createCollection("nombre_colección",  
  {validator: { $jsonSchema: {...}}})
```
- También se puede configurar a posteriori
- Si no es válido el elemento:
  - Devuelve un error (opción por defecto)
  - Se puede configurar para que devuelva un warning
- [Más información](#)

# MongoDB – Validación

```
db.createCollection("contacts", { validator: {  
  $jsonSchema: {  
    type: "object",  
    required: [ "phone", "name" ],  
    properties: {  
      phone: {type: "string"},  
      name: {type: "string"}  
    }  
  }  
})
```

```
db.contacts.insertOne({phone:"91123123", name:"Juan"})  
db.contacts.insertOne({phone:91123123, name:"Sofía"})
```

# MongoDB – Validación

## BSON Types:

```
db.createCollection("contacts", validator: {
  $jsonSchema: {
    bsonType: "object",
    required: [ "phone", "name" ],
    properties: {
      phone: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      name: {
        bsonType: "string",
        description: "must be a string and is required"
      }
    }
  }
})
```

# MongoDB – Validación

BSON Types:

ObjectId

```
ObjectId("643e64d37bca71282283f4c4").getTimestamp()
```

# Útiles MongoDB

- Métodos

<https://www.mongodb.com/docs/manual/reference/method/>

- Operaciones CRUD

<https://www.mongodb.com/docs/mongodb-shell/crud/>



# NODE.JS



# Node.js

- Varias opciones para trabajar
- Driver oficial: [MongoDB](#)
- Object Modeling Tool: [Mongoose](#)



# Node.js – Conexión

- Instalar el driver

```
npm install mongodb
```

- Connection String

- Formato:

```
mongodb://[username:password@]host1[:port1][,...hostN[:portN]][/[defaultauthdb][?options]]
```

- Ej.: `mongodb://127.0.0.1:27017`

- [Info](#)

# Node.js – Conexión

```
const { MongoClient, ObjectId } = require("mongodb");
const uri = "mongodb://127.0.0.1";
const client = new MongoClient(uri);
const connectToDatabase = async () => {
  try { await client.connect();
    } catch (e){ console.error(e); }
}
const listDatabases = async (client) => {
  databasesList = await client.db().admin().listDatabases();
  console.log("Databases:");
  databasesList.databases.forEach((db) => console.log(` - ${db.name}`));
}
const main = async () => {
  try {
    await connectToDatabase();
    await listDatabases(client);
  } catch (e){ console.error(e);
  } finally { client.close();
  }
}
main();
```



# Node.js – Insert

`insertOne(document)`

- Para insertar un documento
- [Info](#)

`insertMany(documents_array)`

- Para insertar múltiples documentos
- [Info](#)

# Node.js – insertOne()

```
const dbname = "bank";
const collection_name = "accounts";
const accountsCollection = client.db(dbname).collection(collection_name);
const sampleAccount = {
  account_holder: "Linus Torvalds",
  account_id: "MDB829001337",
  account_type: "checking",
  balance: 50352434,
}
const main = async () => {
  try {
    await connectToDatabase()
    let result = await accountsCollection.insertOne(sampleAccount);
    console.log(`Inserted document: ${result.insertedId}`);
  } catch (err) { console.error(`Error inserting document: ${err}`); }
  finally { await client.close(); }
}
main();
```



# Node.js – insertMany()

```
const dbname = "bank";
const collection_name = "accounts";
const accountsCollection = client.db(dbname).collection(collection_name);
const sampleAccounts = [
  {
    account_id: "MDB011235813",
    account_holder: "Ada Lovelace",
    account_type: "checking",
    balance: 60218,
  }, {
    account_id: "MDB829000001",
    account_holder: "Muhammad ibn Musa al-Khwarizmi",
    account_type: "savings",
    balance: 267914296,
  }
];
const main = async () => {
  try {
    await connectToDatabase();
    let result = await accountsCollection.insertMany(sampleAccounts);
    console.log(`Inserted ${result.insertedCount} documents`);
    console.log(result);
  } catch (err) { console.error(`Error inserting documents: ${err}`); }
  finally { await client.close(); }
}
main();
```



# Node.js – Find

`findOne(query, options)`

- Para buscar un documento
- [Info](#)

`find(query, options)`

- Para buscar múltiples documentos
- [Info](#)

# Node.js – Find

options:

`{sort: sort_info}`

- Para ordenar la búsqueda
- También como método: `.sort(sort_info)`

`{projection: projection_info}`

- Para aplicar proyección
- También como método: `.project(projection_info)`



# Node.js – Find

options:

`{limit: number}`

- Para limitar los resultados de la búsqueda
- También como método: `.limit(number)`

`{skip: number}`

- Para saltarnos los primeros resultados
- También como método: `.skip(number)`

# Node.js – findOne()

```
const dbname = "sample_mflix";
const collection_name = "movies";
const accountsCollection = client.db(dbname).collection(collection_name);
const documentToFind = { _id: new ObjectId("573a13adf29313caabd2b765") };

const main = async () => {
  try {
    await connectToDatabase();
    let result = await accountsCollection.findOne(documentToFind);
    console.log("Found the following document:\n", result);
  } catch (err) {
    console.error(`Error finding document: ${err}`);
  } finally {
    await client.close();
  }
}

main();
```

# Node.js – find()

```
const dbname = "sample_mflix";
const collection_name = "movies";
const collection = client.db(dbname).collection(collection_name);
const documentsToFind = { year: 2015 };

const main = async () => {
  try {
    await connectToDatabase()
    let result = collection.find(documentsToFind).limit(5);
    let docCount = collection.countDocuments(documentsToFind);
    await result.forEach((doc) => console.log(doc));
    console.log(`Found ${await docCount} documents`);
  } catch (err) {
    console.error(`Error finding documents: ${err}`);
  } finally {
    await client.close();
  }
}
main();
```

# Node.js – Update

`updateOne(filter, update, options)`

- Para actualizar un documento
- [Info](#)

`updateMany(filter, update, options)`

- Para actualizar múltiples documentos
- [Info](#)

`replaceOne(filter, replacement, options)`

- Para reemplazar un documento
- [Info](#)



# Node.js – updateOne()

```
const dbname = "sample_mflix";
const collection_name = "movies";
const collection = client.db(dbname).collection(collection_name)
const documentToUpdate = { _id: new ObjectId("573a13adf29313caabd2b765") };
const update = { $inc: { metacritic: 1 } }
const main = async () => {
  try {
    await connectToDatabase()
    let result = await collection.updateOne(documentToUpdate, update);
    result.modifiedCount === 1
      ? console.log("Updated one document")
      : console.log("No documents updated");
  } catch (err) {
    console.error(`Error updating document: ${err}`);
  } finally {
    await client.close()
  }
}
main();
```



# Node.js – updateMany()

```
const dbname = "sample_mflix";
const collection_name = "movies";
const collection = client.db(dbname).collection(collection_name);
const documentsToUpdate = { year: 2015 };
const update = { $inc: { metacritic: 1 } };
const main = async () => {
  try {
    await connectToDatabase()
    let result = await collection.updateMany(documentsToUpdate, update);
    result.modifiedCount > 0
      ? console.log(`Updated ${result.modifiedCount} documents`)
      : console.log("No documents updated");
  } catch (err) {
    console.error(`Error updating documents: ${err}`);
  } finally {
    await client.close();
  }
}
main();
```



# Node.js – Delete

`deleteOne(query)`

- Para borrar un documento
- [Info](#)

`deleteMany(query)`

- Para borrar múltiples documentos
- [Info](#)

# Node.js – deleteOne()

```
const dbname = "bank";
const collection_name = "accounts";
const accountsCollection = client.db(dbname).collection(collection_name);
const documentToDelete = { _id: new ObjectId("6447c4e556b01c9d4d84028e") };
const main = async () => {
  try {
    await connectToDatabase();
    let result = await accountsCollection.deleteOne(documentToDelete);
    result.deletedCount === 1
      ? console.log("Deleted one document")
      : console.log("No documents deleted");
  } catch (err) {
    console.error(`Error deleting documents: ${err}`);
  } finally {
    await client.close();
  }
}
main();
```



# Node.js – deleteMany()

```
const dbname = "bank";
const collection_name = "accounts";
const accountsCollection = client.db(dbname).collection(collection_name);
const documentsToDelete = { balance: { $gt: 90000 } };
const main = async () => {
  try {
    await connectToDatabase()
    let result = await accountsCollection.deleteMany(documentsToDelete);
    result.deletedCount > 0
      ? console.log(`Deleted ${result.deletedCount} documents`)
      : console.log("No documents deleted");
  } catch (err) {
    console.error(`Error deleting documents: ${err}`);
  } finally {
    await client.close();
  }
}
main();
```

# Node.js – Agregaciones

- Pipeline para realizar operaciones sobre los datos
- Aplicar múltiples etapas
- [Más información](#)

```
const pipeline = [  
  { ... },  
  { ... },  
  ...];
```

```
client.db("dbname").collection("collectionname").aggregate(pipeline);
```

# Node.js – Agregaciones

```
const { MongoClient, ObjectId } = require("mongodb");
const uri = "mongodb://127.0.0.1:27017";
const client = new MongoClient(uri);
const connectToDatabase = async () => {
  try { await client.connect();
    } catch (e){ console.error(e);
  }
}
const dbname = "bank";
const collection_name = "accounts";
const accounts = client.db(dbname).collection(collection_name);
const pipeline = [
  { $match: { balance: { $lt: 1000 } } },
  { $group: {
    _id: "$account_type",
    total_balance: { $sum: "$balance" },
    avg_balance: { $avg: "$balance" }
  } } ];
const main = async () => {
  try {
    await connectToDatabase();
    let result = await accounts.aggregate(pipeline);
    await result.forEach((doc) => console.log(doc));
  } catch (err) {
    console.error(`Error finding documents: ${err}`);
  } finally {
    await client.close();
  }
}; main();
```



# Node.js – Índices

- Acelera las búsquedas
- Por defecto creados para `_id`
- Conviene usarlos con aquellas queries en las que busquemos elementos en orden
- [Más información](#)

```
const index = { ... }
```

```
client.db("dbname").collection("collectionname").createIndex(index);
```

# Node.js – Índices

```
const { MongoClient, ObjectId } = require("mongodb");
const uri = "mongodb://127.0.0.1:27017";
const client = new MongoClient(uri);
const connectToDatabase = async () => {
  try { await client.connect();
    } catch (e){ console.error(e);
  }
}
const dbname = "sample_mflix";
const collection_name = "movies";
const movies = client.db(dbname).collection(collection_name);
const main = async () => {
  try {
    await connectToDatabase();
    const result = await movies.createIndex({ title: 1 });
    console.log(`Index created: ${result}`);
  } catch (err) { console.error(`Error finding documents: ${err}`);
  } finally { await client.close();
  }
}
main();
```



# Node.js – Índices

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://127.0.0.1:27017";
const client = new MongoClient(uri);
const connectToDatabase = async () => {
  try { await client.connect();
    } catch (e) { console.error(e);
  }
};
const dbname = "sample_mflix";
const collection_name = "movies";
const movies = client.db(dbname).collection(collection_name);
const main = async () => {
  try {
    await connectToDatabase();
    const query = { title: /^Batman.* / };
    const sort = { year: 1 };
    const projection = { _id: 0, title: 1, year: 1 };
    let result = movies.find(query).sort(sort).project(projection);
    await result.forEach((doc) => console.log(doc));
  } catch (err) { console.error(`Error finding documents: ${err}`);
  } finally { await client.close();
  }
};
main();
```



# Node.js – Transacciones

- Permite ejecutar una serie de operaciones en conjunto
- Se aplican todas o ninguna (atomicidad)
- Cumple ACID
- Necesitamos una sesión común a todas las operaciones
- [Más información](#)

```
const session = client.startSession();
const transactionResults = await session.withTransaction(async () => {
  //Las operaciones
});
```



# Node.js – Transacciones

Pasos para realizar la transacción:

1. Restar la cantidad de la cuenta origen
2. Añadir la cantidad a la cuenta destino
3. Añadir la información a la colección transfers
4. Añadir la información de la transferencia a la cuenta origen
5. Añadir la información de la transferencia a la cuenta destino



# Node.js – Transacciones

```
const { MongoClient } = require("mongodb");
const uri = "mongodb://127.0.0.1:27017";
const client = new MongoClient(uri);
const dbname = "bank";
const accounts = client.db(dbname).collection("accounts");
const transfers = client.db(dbname).collection("transfers");
let account_id_sender = "MDB574189300";
let account_id_receiver = "MDB343652528";
let transfer_id = "TR21872187";
let transaction_amount = 100;
const session = client.startSession();
const main = async () => {
  try {
    const transactionResults = await session.withTransaction(async () => {
      //Continua en la siguiente página
    });
    if (transactionResults) { console.log("Transaction completed successfully.");
    } else { console.log("Transaction failed."); }
  } catch (err) { console.error(`Transaction aborted: ${err}`)
    process.exit(1)
  } finally { await session.endSession()
    await client.close()
  }
}
main();
```



# Node.js – Transacciones

```
const transactionResults = await session.withTransaction(async () => { //Continuación
  const senderUpdate = await accounts.updateOne(
    { account_id: account_id_sender },
    { $inc: { balance: -transaction_amount } },
    { session }
  );
  const receiverUpdate = await accounts.updateOne(
    { account_id: account_id_receiver },
    { $inc: { balance: transaction_amount } },
    { session }
  );
  const transfer = {
    transfer_id: transfer_id,
    amount: transaction_amount,
    from_account: account_id_sender,
    to_account: account_id_receiver,
  };
  const insertTransferResults = await transfers.insertOne(transfer, { session });
  const updateSenderTransferResults = await accounts.updateOne(
    { account_id: account_id_sender },
    { $push: { transfers_complete: transfer.transfer_id } },
    { session }
  );
  const updateReceiverTransferResults = await accounts.updateOne(
    { account_id: account_id_receiver },
    { $push: { transfers_complete: transfer.transfer_id } },
    { session }
  );
});
```



# Node.js – Más

- [Collations](#): reglas de ordenación de Strings
- [GridFS](#): para almacenar archivos mayores de 16MB
- [Cifrado](#)
- [Compound operations](#):
  - `findOneAndDelete()`
  - `findOneAndUpdate()`
  - `findOneAndReplace()`
- [Y más...](#)

# Referencias

- Guide to database systems

<https://alg.manifoldapp.org/read/guide-to-database-systems>

- Estadísticas sobre bases de datos

<https://db-engines.com/>

- Documentación MongoDB

<https://www.mongodb.com/docs/>

- MongoDB University

<https://university.mongodb.com/>

- [MongoDB Node.js Quick reference](#)

