MANUAL MONGODB

Javier Figueroa Vicente

ÍNDICE

1. Introducción a MongoDB

1.1 Definición de bases de datos NoSQL

Es un diseño de base de datos que permite almacenar y consultar datos fuera de las estructuras tradicionales que se encuentran en las bases de datos relacionales. Aunque puede almacenar los datos que se encuentran dentro de los sistemas de gestión de bases de datos relacionales. Las bases de datos NoSQL alojan datos dentro de una estructura de datos, como un documento JSON.
Proporcionan velocidad y escalabilidad, lo que las convierte en una opción popular por su rendimiento y facilidad de uso.

1.2 Tipos de base de datos NO-SQL

1.2.1 Almacén de pares clave-valor

Se considera la forma más simple de bases de datos NoSQL. Este modelo de datos sin esquema se organiza en un diccionario de pares clave-valor, donde cada elemento tiene una clave y un valor. La clave podría ser un ID y el valor, por ejemplo, unos datos de compra. Se suele usar para almacenar caché y guardar datos de sesión. Cuando se necesita extraer varios registros a la vez no es la mejor opción..

1.2.2 Almacén de documentos

Las bases de datos de documentos almacenan datos como documentos. Pueden ser útiles en la gestión de datos semiestructurados y, por lo general, los datos se almacenan en formatos JSON, XML o BSON.
Los usos más comunes de estas bases de datos son los sistemas de gestión de contenidos y los perfiles de usuario.
MongoDB es un ejemplo de base de datos de almacenamiento de documentos.

1.2.3 Almacén distribuido en columnas

Almacenan información en columnas, lo que permite a los usuarios acceder solo a las columnas específicas que necesitan sin asignar memoria adicional a datos irrelevantes.
Apache HBase es un ejemplo de este tipo de base de datos.

1.2.4 Almacén de grafos

Guarda datos de una rama de conocimiento o grafo. Los elementos de datos se almacenan como nodos, aristas y propiedades. Se utilizan para almacenar y gestionar una red de conexiones entre elementos dentro del grafo.

1.2.5 Almacén en memoria

Con esta base de datos, los datos residen en la memoria principal en lugar de en disco, lo que agiliza el acceso a los datos.

1.3 Ventajas y desventajas frente a las bases de datos relacionales

Ventajas:

- **Rentabilidad:** El coste de licencias, gestores de base de datos y hardware es mucho más barato que el de un RDBMS . Optimizando así los recursos.

- **Flexibilidad:** Que escale de manera horizontal y tenga un modelo de datos flexible significa que las bases de datos NoSQL pueden llegar a volúmenes mucho más grandes de datos que el resto. Lo que hace que tenga un desarrollo más rápido e iteraciones de código más frecuentes.

- **Réplica:** Esta funcionalidad copia y almacena los datos en varios servidores. Esta proporciona fiabilidad de datos y garantiza el acceso durante tiempos de inactividad y protección frente a la pérdida de datos si los servidores se desconectan.

- **Velocidad:** agiliza el almacenamiento y procesamiento de datos para todo tipo de usuarios. La velocidad también hace que NoSQL sea mejor opción para las webs modernas.

Desventajas:

- **Falta de estandarización:** Existen muchas soluciones con diferentes formas de operación y consulta. Tanta variedad puede complicar la elección y el proceso de aprendizaje.

- **Madurez relativa:** Muchas bases NoSQL no tienen la misma madurez que las soluciones SQL

- **Consistencia vs Disponibilidad:** Muchas bases de datos NoSQL siguen el teorema CAP, lo que significa que es difícil garantizar simultáneamente la consistencia, disponibilidad y tolerancia a particiones. Esto puede llevar a compromisos entre estos factores dependiendo de la base de datos elegida.

- **Menor énfasis en la integridad de datos**: La flexibilidad de las bases de datos NoSQL puede ser una ventaja, pero también puede ser un desafío cuando se trata de mantener la integridad de los datos, especialmente en sistemas con esquemas variables o inexistentes.

- **Curva de aprendizaje:** Para aquellos familiarizados con el modelo relacional y SQL, el cambio a NoSQL puede requerir un período de aprendizaje y ajuste debido a las diferencias en el enfoque y la sintaxis de consulta.

1.4 Explicar qué es MongoDB y por qué es útil

1.4.1 ¿Qué es MongoDB?

MongoDB es un sistema de gestión de bases de datos NoSQL y de código abierto, que utiliza documentos flexibles en lugar de tablas y filas para procesar y almacenar varias formas de datos. Esto no solo simplifica la gestión de la base de datos para los desarrolladores, sino que también crea un entorno altamente escalable para aplicaciones y servicios multiplataforma.
Estos documentos, con formato JSON, pueden almacenar varios tipos de datos y distribuirse en varios sistemas.

1.4.2 ¿Por qué es útil?

MongoDB es útil porque es muy flexible en cuanto a la arquitectura de esquema que tiene, funciona con almacenamiento y datos no estructurados. Además permite fragmentar los datos de un gran conjunto de datos y los distribuye en varios servidores.
También obtiene un mayor rendimiento que el resto de bases de datos NoSQL porque almacena los datos en la RAM para acceder rápidamente a ellos. por eso también implica que necesita menos potencia para buscar y recuperar los datos de la base de datos.

2. Creación de la estructura de datos de MongoDB

2.1 Cómo es la Estructura de Datos

La estructura de datos de MongoDB consta de 3 partes:

- **BBDD:** base de datos o repositorio donde se almacenan los archivos.
- **Colecciones:** grupos de datos. Se almacenan dentro de la base de datos y están formadas por documentos.
- **Documentos:** Cada uno de los grupos de información que forman una colección.

2.2 Cómo hacer la creación de Colecciones

Para entrar dentro del contenedor de Docker tendremos que escribir en el cmd: **docker exec -it MongoDB bash.**
Seguido de **mongosh.**

```
C:\Users\javie>docker exec -it MongoDB bash
root@cca2cb9069e9:/# mongosh
Current Mongosh Log ID: 662d51e11f09ac7c522202d7
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2
.2.5
Using MongoDB:          7.0.8
Using Mongosh:          2.2.5

For mongosh info see: https://docs.mongodb.com/mongodb-shell/


To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.co
m/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

------
   The server generated these startup warnings when booting
   2024-04-27T19:15:49.142+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine. S
ee http://dochub.mongodb.org/core/prodnotes-filesystem
   2024-04-27T19:15:50.687+00:00: Access control is not enabled for the database. Read and write access to data and conf
iguration is unrestricted
   2024-04-27T19:15:50.688+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'nev
er' in this binary version
   2024-04-27T19:15:50.688+00:00: vm.max_map_count is too low
------
```

Dentro de MongoDB podemos tener tantas colecciones como queramos. En una analogía con SQL sería el equivalente a las tablas.
No tiene una estructura estándar como una tabla, quiere decir que no tiene las mismas columnas pero N filas.

- **Como ver una colección:** Para ver una colección de una base de datos tenemos que escribir: “show collections”.

```
test> show collections
usuarios
```

- **Para crear una colección** tenemos que escribir el comando: db.createCollection("nombre_coleccion").

```
test> db.createCollection("usuarios")
{ ok: 1 }
test>
```

- **Para borrar una colección** tenemos que escribir el comando: db.nombre_colección.drop().

```
test> db.usuarios.drop()
true
```

- **Para insertar datos en una colección** deberemos usar el comando: db.nombre_coleccion.insert().

```
test> db.usuarios.insertOne({"nombre":"pepe"})
{
  acknowledged: true,
  insertedId: ObjectId('662d53df1f09ac7c522202d9')
}
test>
```

2.3 Cómo hacer la creación de documentos

Cada documento es una estructura JSON que puede tener diferentes campos de diferentes tipos de datos.

- **Colección:** db.usuarios.insertOne({}) ó db.usuarios.insertMany({})

```
test> db.usuarios.insertOne({"nombre":"pepe"})
{
  acknowledged: true,
  insertedId: ObjectId('662d53df1f09ac7c522202d9')
}
test>
```

- **Tipo de datos char:** “nombre”:”pepe”

```
({"nombre":"pepe"})
```

- **Tipo de datos decimal:** “precio”: 40.2

```
({"precio":40.2})
```

- **Tipo de datos boolean:** “active”: false

```
({"active":false})
```

- **Tipo de datos date:** “created_at”: new Date (“27/04/2024”)

```
({"creado": new Date ("27/04/2024")})
```

- **Tipo de datos list:** “varios_datos”:[1,”hola”]

```
({"varios_datos":[1,"hola"]})
```

- **Tipo de datos dictionary:** “facturer”:{
  “name”:”dell”,
  “version”:”xps”,
  “location”:{
  “city”:”usa”,
  “adress”:”sdsfs”}
  }

```
({"facturer":{"name":"dell","version":"xps","location":{"city":"usa","adress":"sdsfs"}}})
```

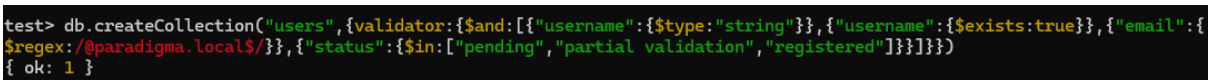
2.4 Cómo hacer la validación de la Estructura de Datos

Una validación en MongoDB sirve para que no se puedan introducir datos en formato no válido en ninguna colección.

Para crear una validación deberemos introducir primero la estructura que queremos que tenga la colección, seguido del comando **validator: [$and:[]]**, para que no se inserte ningún documento que no cumpla la estructura.

Ejemplo:

```
db.createCollection("users",
{
validator: {$and:
 [
  {"username":{$type: "string"} },
  {"username":{$exists: true} },
  {"email": {$regex: /@paradigma.local$/} },
  {"status":{$in: ["pending","partial validation","registered" ]
} }
 ]
 }
})
```



En el caso de que el dato que intentamos insertar no sea correcto podemos hacer que salga un mensaje de error.

```
db.users.insert(
 {"username":1,
 email:"evalero@paradigma.local"
 })
WriteResult({
    "nInserted" : 0,
    "writeError" : {
        "code" : 121,
        "errmsg" : "Document failed validation"
    }
})
```

3. CRUD

CRUD son las siglas de **Create**, **Read**, **Update** y **Delete.** Qué son las acciones que nos permite realizar MongoDB para modificar las colecciones y los documentos de la base de datos.

3.1 Cómo es la inserción de Documentos de un valor

Para insertar un documento de un valor en una colección tendremos que escribir el comando: db.createCollection(“nombre_coleccion”, {“nombre”: {$type:”string”}}).

3.2Cómo es la inserción de Documentos de muchos valores

Para insertar un documento o varios en MongoDB, utilizaremos el comando insert seguido de la cantidad de datos que queramos insertar, si queremos insertar solo uno será “One” y si son varios “Many”. Quedando los siguientes comandos:

Insertar solo un documentos:

db.nombre_coleccion.insertOne([{“nombre”:”Javi”}])

```
test> db.usuarios.insertOne([{"nombre":"Javi"}])
{
  acknowledged: true,
  insertedId: ObjectId('662d66d61f09ac7c522202df')
}
```

Insertar varios documentos:

db.nombre_coleccion.insertMany([{“nombre”:”Javi”},{“nombre”:”Miquel”},{“nombre”:”Alex”}])

```
test> db.usuarios.insertMany([{"nombre":"Javi"},{"nombre":"Miquel"},{"nombre":"Alex"}])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('662d67171f09ac7c522202e0'),
    '1': ObjectId('662d67171f09ac7c522202e1'),
    '2': ObjectId('662d67171f09ac7c522202e2')
  }
}
```

3.3 Cómo se hace una consulta básica

Para hacer una consulta básica tenemos que escribir el comando **.find()**, para que nos devuelva todos los documentos.

Ejemplo:

db.usuarios.find( )

```
test> db.usuarios.find()
[
  {
    '0': { nombre: 'Javi' },
    _id: ObjectId('662d66d61f09ac7c522202df')
  },
  { _id: ObjectId('662d67171f09ac7c522202e0'), nombre: 'Javi' },
  { _id: ObjectId('662d67171f09ac7c522202e1'), nombre: 'Miquel' },
  { _id: ObjectId('662d67171f09ac7c522202e2'), nombre: 'Alex' }
]
```

3.4 Cómo hacer una búsqueda por criterios simples

Para hacer una consulta básica en MongoDB utilizaremos el comando **“find”** seguido del dato que queremos buscar.

Por ejemplo:

db.nombre_coleccion.find({“nombre”: ”Javi”})

```
test> db.usuarios.find({"nombre":"Javi"})
[ { _id: ObjectId('662d67171f09ac7c522202e0'), nombre: 'Javi' } ]
```

3.5 Cómo ordenar resultados

Para ordenar los resultados en MongoDB tenemos el comando **sort .** Para poder ordenar en este caso, en orden alfabético le pondremos el valor por el que queremos que aparezcan seguido del número 1.

Ejemplo: db.usuarios.find({"nombre":"Javi"})

```
test> db.usuarios.find().sort({nombre: 1})
[
  {
    '0': { nombre: 'Javi' },
    _id: ObjectId('662d66d61f09ac7c522202df')
  },
  { _id: ObjectId('662d67171f09ac7c522202e2'), nombre: 'Alex' },
  { _id: ObjectId('662d67171f09ac7c522202e0'), nombre: 'Javi' },
  { _id: ObjectId('662d67171f09ac7c522202e1'), nombre: 'Miquel' }
]
test>
```

Y para ordenarlo en orden inverso, en vez de un 1 le colocaremos un -1.

Ejemplo:

```
test> db.usuarios.find().sort({nombre: -1})
[
  { _id: ObjectId('662d67171f09ac7c522202e1'), nombre: 'Miquel' },
  { _id: ObjectId('662d67171f09ac7c522202e0'), nombre: 'Javi' },
  { _id: ObjectId('662d67171f09ac7c522202e2'), nombre: 'Alex' },
  {
    '0': { nombre: 'Javi' },
    _id: ObjectId('662d66d61f09ac7c522202df')
  }
]
```

3.6 Cómo hacer consultas Avanzadas

Para hacer una búsqueda por criterios simples, además del find, podemos utilizar varios recursos:

- **Cursores:** al igual que en SQL un cursor es una conexión con el servidor que permite iterar sobre los resultados de una consulta.
  Para crear un cursor hay que utilizar una variable:
  Ejemplo: var miCursor=db.usuarios.find();

  ```
  test> var miCursor=db.usuarios.find();
  ```

  Para buscar todos los datos de una colección tendremos que hacer, por ejemplo, un while para recorrer la colección.

  Ejemplo: while(miCursor.hasNext()){printjson(miCursor.next());};

```
test> while(miCursor.hasNext()){printjson(miCursor.next());};
{
  '0': {
    nombre: 'Javi'
  },
  _id: ObjectId('662d66d61f09ac7c522202df')
}
{
  _id: ObjectId('662d67171f09ac7c522202e0'),
  nombre: 'Javi'
}
{
  _id: ObjectId('662d67171f09ac7c522202e1'),
  nombre: 'Miquel'
}
{
  _id: ObjectId('662d67171f09ac7c522202e2'),
  nombre: 'Alex'
}
```

- **Skip:** Con skip le decimos al cursor que salte el número de registros que nosotros le indicamos, en este caso 2**.**

  Ejemplo: db.usuarios.find().skip(2);

```
test> db.usuarios.find().skip(2);
[
  { _id: ObjectId('662d67171f09ac7c522202e1'), nombre: 'Miquel' },
  { _id: ObjectId('662d67171f09ac7c522202e2'), nombre: 'Alex' }
]
```

- **Limit:** Con limit indicamos que el cursor nos devuelva sólo el número de registros que nosotros le indicamos.

  Ejemplo: db.usuarios.find().limit(2);

- **Filtros:** En las consultas find, el filtro sería como utilizar un WHERE en las consultas SQL. Para utilizar el filtro tendríamos que pasar un documento como parámetro de la consulta.

  Ejemplo: db.usuarios.find({“nombre”:”Alex”});

```
test> db.usuarios.find({"nombre":"Alex"});
[ { _id: ObjectId('662d67171f09ac7c522202e2'), nombre: 'Alex' } ]
```

- **Proyecciones:** se utiliza para devolver un conjunto determinado de campos de un documento.

  Ejemplo: db.usuarios.find({},{“nombre”:1});

```
test> db.usuarios.find({},{"nombre":1});
[
  { _id: ObjectId('662d66d61f09ac7c522202df') },
  { _id: ObjectId('662d67171f09ac7c522202e0'), nombre: 'Javi' },
  { _id: ObjectId('662d67171f09ac7c522202e1'), nombre: 'Miquel' },
  { _id: ObjectId('662d67171f09ac7c522202e2'), nombre: 'Alex' }
]
```

En caso de que no queramos mostrar el id, ya que estamos aplicando un filtro vacío, tendríamos que añadir _id:0

Ejemplo: db.usuarios.find({},{“nombre”:1},{“_id”:0});

- **FindOne:** se usa para devolver solo 1 documento en concreto. Por lo que como solo devuelve 1, no devuelve el cursor

  Ejemplo: db.usuarios.findOne({“nombre”:”Javi”});

```
test> db.usuarios.findOne({"nombre":"Javi"});
{ _id: ObjectId('662d67171f09ac7c522202e0'), nombre: 'Javi' }
```

  - **findAndModify:** utilizando findAndModify buscaremos un documento para modificar alguno de sus datos y luego devolverlo. El comando tiene diferentes parametros Query: Los criterios de búsqueda. El filtro que dice que documentos deben buscarse. Igual que en find y en findOne. Aunque la consulta devuelva varios documentos, solo se modificará uno de ellos:
    1. **Sort:** podemos especificar un orden para tener control sobre el documento que se modificará en caso de que se devuelvan varios resultados.
    2. **Remove:** campo booleano. Este parámetro no es necesario si se añade el parámetro Update (ver siguiente punto). Si el valor es true, se borrará el documento encontrado.
    3. **Update:** este parámetro no es necesario si se añade el parámetro Remove. En este campo deberemos introducir un JSON con la sentencia de actualización.
    4. **New:** campo booleano. Si es true se devuelve el documento ya modificado, si es false se devuelve el original.
    5. **Fields:** proyección con los datos que queremos devolver.
    6. **Upsert:** si es true y la consulta no encuentra documentos, se creará un documento nuevo. En otro caso se actualizará el documento.

3.7 Cómo usar Operadores de consulta

- **Operaciones de comparación**
  Es muy típico filtrar por un valor determinado de un documento para encontrar los documentos que deseamos, para esto utilizaremos diferentes operadores de comparación:

  1. **Igual que ($eq):** { <field>: { $eq: <value> } }

2. **Mayor que ($gt:):** { field: { $gt: value } }
3. **Mayor o igual a ($gte:):** { field: { $gte: value } }
4. **Menor que ($lt:):**{ field: { $lt: value } }
5. **Menor o igual a ($lte:):** { field: { $lte: value } }
6. **Diferente a ($ne:):** { field: { $ne: value } }
7. **Que sea igual a ($in:):** { field: { $in: [<value1>, <value2>, ... <valueN>] } }
8. **Que no incluye ($nin)**: { field: { $nin: [<value1>, <value2> ... <valueN>] } }

- **Operaciones lógicas**

  El símil con las bases de datos relacionales sería el operador **OR** y el operador **AND** que va después del WHERE.

  En MongoDB se utiliza **$or** para la cláusula OR y **$and** para la cláusula AND.

  1. **OR** se utiliza para que nos muestre los datos que cumplan, como mínimo 1 de las condiciones que le digamos.

     Si queremos usar **$or,** usaremos por ejemplo el siguiente código:

     { $or: [{ <expression1> }, { <expression2> }, { <expressionN> }] }

     Y en contraposición tenemos el **$nor:**

     { $nor: [{ <expression1> }, { <expression2> }, { <expressionN> }] }

  2. **AND** se utiliza para que se cumplan todas las condiciones que pongamos en el código.

     Si queremos usar **$and,** usaremos por ejemplo el siguiente código:

     { $and: [{ <expression1> }, { <expression2> } ,{ <expressionN> }] }

     Y en contraposición al **$and** tenemos el **$not:**

     { field: { $not: { <operator-expression> } } }

- **Operaciones según la existencia o el tipo de los elementos**
    1. **$exists:** En MongoDB podemos introducir un operador para que compruebe si en 1 documento existe el campo que nosotros introducimos. Esto se hace mediante el comando **$exists.**

       Ejemplo: db.people.find({company:{**$exists**:true}},{name:1,age:1,company:1})

    2. **$type:** también podemos filtrar por el tipo de dato que tiene el campo, esto se hace mediante el operador **$type.**

       Ejemplo: db.people.find({company:{**$type**:2}},{name:1,age:1,company:1})

- **Operaciones de array**
    1. **$all:** es el equivalente al $and pero en operadores de array, selecciona los documentos que incluyan todos los valores dados:

       { <field>: { $all: [<value1> , <value2> ...] } }

    2. **$elemMatch:** es el equivalente al $or en operadores de array, selecciona los documentos que cumplan al menos 1 de los criterios introducidos.

       { <field>: { $elemMatch: { <query1>, <query2>, ... } } }

    3. **$size:** este operador busca documentos que tengan el número de valores que introducimos:

       db.collection.find({ field: { $size: 2 } });

3.8 Cómo hacer consultas con expresiones regulares

Las expresiones regulares equivalen a la expresión **like** en SQL.
Para poder utilizarla podemos utilizar el operador **$regex** o bien **/expresión/.**

Ejemplo 1: db.test.find({ correo : { **$regex :** «@» } })

Ejemplo 2: db.test.find({ correo : **/@/** })

3.9 Cómo hacer la actualización de Documentos de un valor

para hacer una actualización de documentos de un valor utilizaremos **.update** detrás de la colección. Dentro de los paréntesis tendremos que introducir el documento seguido del valor nuevo que queremos asignar.
Ejemplo: db.products**.update**({*_id:2},{*_id:5})

Este ejemplo solo es eficaz si el documento tiene 1 solo valor.

3.10 Cómo hacer la actualización de Documentos de muchos valores

Para hacer la actualización de 1 documento que tiene varios valores no sirve el método anterior, sino que tenemos que introducir el operador $set con el valor que queremos actualizar cambiado.

Ejemplo:

```
db.products.update({
   _id: ObjectId("1")
},
{
   $set: {
      cantidad: 35,
      precio: 60
   }
})
```

MongoDB solo cambia 1 valor para evitar errores, si queremos cambiar varios tendremos que utilizar el operador **multi:true:**
Ejemplo: db.products.update({tipo:"HDD"},{$set:{cantidad:10}},{multi:true})

3.11 Cómo eliminar datos de Documentos de un valor

Para eliminar datos de un valor tenemos que introducir el comando .remove detrás de la colección. Para eliminar documentos de un valor introduciremos el valor específico dentro del remove.

Ejemplo: db.products.**remove**({_id:890})

O en su defecto utilizaremos **.deleteOne**

3.12 Cómo eliminar datos de Documentos de muchos valores

Si queremos eliminar todos los documentos que incluyan el valor o valores dados, tendríamos que sustituir el **.remove** por **.deleteMany().**

4. Comparativa con otras bases de datos NO-SQL

4.1 Comparación entre MongoDB y otras Bases de Datos NoSQL

Vamos a comparar entre MongoDB y diferentes bases de datos NoSQL. Compararemos Cassandra que usa columnas, Neo4j que usa datos de Grafos y Redis que usa Clave-Valor.

1. **MongoDB (Documentos):**
    - **Descripción:** Utiliza un modelo de documentos flexibles basados en JSON, lo que permite esquemas dinámicos y anidados.
    - **Ventajas:** Flexibilidad para representar datos complejos, fácil integración con lenguajes de programación modernos, soporte para consultas complejas.
    - **Casos de Uso:** Aplicaciones web y móviles, sistemas de gestión de contenido, análisis de datos, sistemas de registro y seguimiento.

2. **Couchbase (Documentos):**
    - **Descripción:** Similar a MongoDB, utiliza documentos JSON pero con esquemas definidos, y ofrece soporte para transacciones ACID.
    - **Ventajas:** Consistencia y alta disponibilidad, escalabilidad horizontal y vertical, rendimiento predecible para aplicaciones en tiempo real.
    - **Casos de Uso:** Aplicaciones empresariales, Internet de las cosas (IoT), análisis en tiempo real, personalización de contenido.
3. **Cassandra (Columnas):**
    - **Descripción:** Utiliza un modelo de almacenamiento basado en columnas, óptimo para conjuntos de datos con un gran número de columnas y consultas selectivas.
    - **Ventajas:** Escalabilidad masiva, rendimiento rápido para operaciones de lectura y escritura distribuidas, tolerancia a fallos.
    - **Casos de Uso:** Sistemas de gestión de datos de alto rendimiento, catálogos de productos, sistemas de mensajería, análisis de registros.
4. **Redis (Clave-Valor):**
    - **Descripción:** Almacena datos en pares de clave-valor simples, con estructuras de datos adicionales como listas, conjuntos y hashes.
    - **Ventajas**: Rendimiento extremadamente rápido debido a su almacenamiento en memoria, alta disponibilidad y redundancia.
    - **Casos de Uso:** Caché de sesiones, almacenamiento en caché de datos, colas de mensajes, sistemas de conteo de visitas.

4.2 Tabla comparativa

	MongoDB	**Cassandra**	**Neo4j**	**Redis**
Escalabilidad	Escala horizontalmente mediante la fragmentación de datos en clústeres distribuidos.	Altamente escalable y distribuido, con capacidades de distribución automática.	Escalable, pero menos eficiente en escenarios de escalado horizontal masivo.	Escala horizontalmente particionando datos entre múltiples nodos.
Modelo de Datos	Utiliza un modelo de documentos JSON flexible, permitiendo esquemas dinámicos y anidados.	Utiliza un modelo de almacenamiento basado en columnas, óptimo para conjuntos de datos con un gran número de columnas.	Utiliza un modelo de datos de grafos que enfatiza las relaciones entre los datos.	Almacena datos en pares de clave-valor simples, ofreciendo una estructura básica y eficiente.
Consistencia y Disponibilidad	Proporciona opciones de consistencia flexible y alta disponibilidad, con configuraciones para ajustarse a los requisitos de la aplicación.	Proporciona alta disponibilidad y permite configurar diferentes niveles de consistencia.	Proporciona alta disponibilidad y consistencia ACID para operaciones de lectura/escritura.	Proporciona consistencia eventual y alta disponibilidad, con opciones para transacciones atómicas en un único comando.

Rendimiento	Ofrece un buen rendimiento para aplicaciones con necesidades de almacenamiento y consulta de documentos complejos.	Destaca en rendimiento para operaciones de lectura y escritura masivas y distribuidas.	Proporciona un rendimiento excepcional para consultas que involucran relaciones complejas.	Ofrece un rendimiento extremadamente rápido para operaciones de lectura y escritura debido a su almacenamiento en memoria.
Casos de Uso	Aplicaciones web y móviles, análisis de datos, gestión de contenido, sistemas de registro y seguimiento.	Sistemas de gestión de datos de alto rendimiento, catálogos de productos, sistemas de mensajería y análisis de registros.	Redes sociales, recomendaciones, análisis de redes, sistemas de recomendación de contenido.	Caché de sesiones, almacenamiento en caché de datos, colas de mensajes, sistemas de conteo de visitas.