4. Bases de Datos NoSQL

Acceso a Datos

Alejandro Roig Aguilar
alejandro.roig@iesalvarofalomir.org

IES Álvaro Falomir
Curso 2023-2024

Objetivo de la unidad y criterios de evaluación

RA5. Desarrolla aplicaciones que gestionan la información almacenada en bases de datos documentales nativas evaluando y utilizando clases específicas.

a) Se han valorado las ventajas e inconvenientes de utilizar bases de datos documentales nativas.

b) Se ha establecido la conexión con la base de datos.

c) Se han desarrollado aplicaciones que efectúan consultas sobre el contenido de la base de datos.

d) Se han añadido y eliminado colecciones de la base de datos.

e) Se han desarrollado aplicaciones para añadir, modificar y eliminar documentos de la base de datos.

Introducción

El término NoSQL (Not Only SQL) se utilizó por primera vez en 1998 por Carlo Strozzi, al referirse al uso de una BD de código abierto que omitía el uso de SQL pero seguía implementando el modelo relacional. No fue hasta el año 2000 cuando surgió Neo4j, la primera BD NoSQL.

Se relacionan con la llegada de la Web 2.0, donde cualquier usuario puede subir contenidos en RRSS, provocando un gran incremento de datos. En general, se recomiendan en aplicaciones que requieren entornos de alto rendimiento o alta concurrencia de acceso.

Las BD NoSQL surgieron para soportar esquemas de datos dinámicos, funcionar en la nube, ser resilientes y escalables horizontalmente.

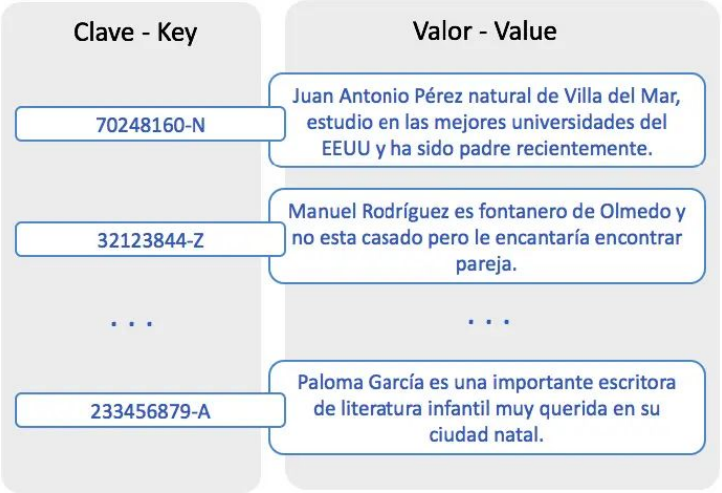
Para ello, se sacrifican las garantías ACID por la filosofía BASE (Basic Availability, Soft State, Eventual Consistency), priorizando la disponibilidad, la tolerancia a fallos y la escalabilidad en lugar de la consistencia estricta.

Diferencias con las bases de datos SQL

- No utilizan SQL como lenguaje de consultas. Por ejemplo, MongoDB utiliza MQL y Cassandra utiliza CQL.
- No utilizan estructuras de almacenamiento fijas para guardar la información: hacen uso de otros modelos, como sistemas de clave-valor, columnares, documentales o grafos.
- No suelen permitir operaciones JOIN ya que la sobrecarga en el sistema resulta muy costosa. La solución es la desnormalización de los datos o realizar JOIN por software en la propia aplicación.
- Arquitectura distribuida, compartiendo la información entre varias máquinas mediante tablas Hash distribuidas.

Tipos de BBDD NoSQL

Clave – Valor: Son las más sencillas en cuanto a funcionalidad. Cada elemento de la BD se identifica y almacena como un nombre de atributo o clave única junto a su valor. Esta información se almacena como un objeto binario (BLOB). Un ejemplo de este tipo es Redis.



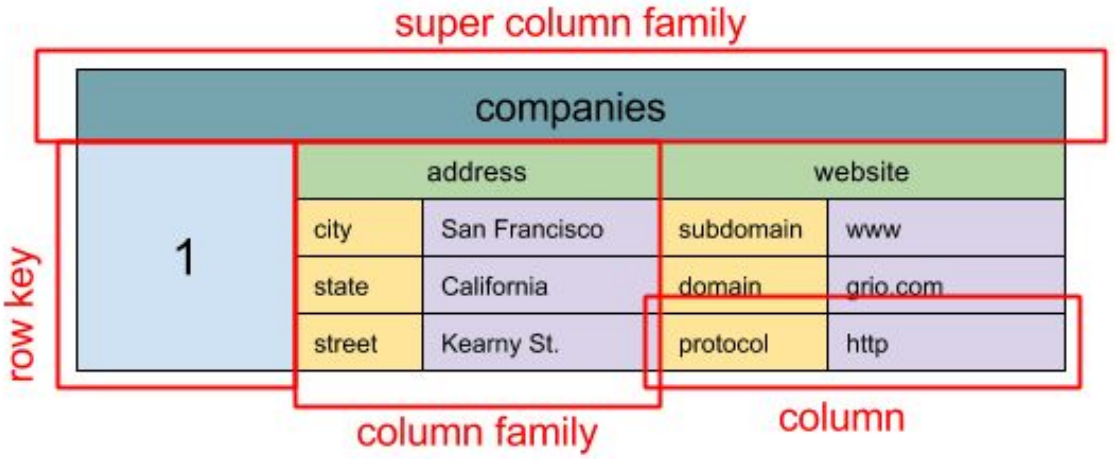
Tipos de BBDD NoSQL

Documentales: En estas BD se empareja una clave con una estructura de datos compleja denominada documento, basado en JSON. Este modelo es el más versátil y los ejemplos más conocidos son MongoDB, Google Cloud Firestore o CouchDB.

```
{
    Nombre:"Alberto",
    Dirección:"Castaños 17",
    Hijos:[
        {Nombre:"Andrés", Edad:12},
        {Nombre:"Susana", Edad:7},
        {Nombre:"Verónica", Edad:4}
    ]
}
```

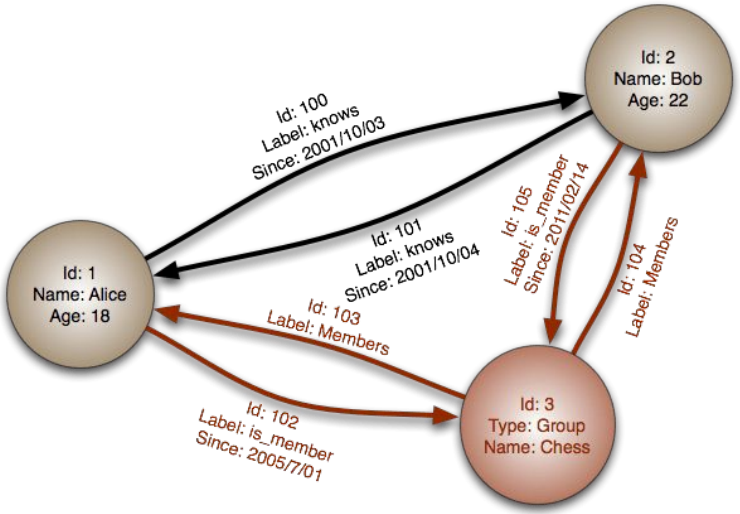
Tipos de BBDD NoSQL

Columnares: Están organizadas en columnas en lugar de filas. Columnas relacionadas se pueden agrupar para formar familias de columnas que contienen múltiples filas. Algunos ejemplos de este tipo son BigTable, Cassandra, HBase o AWS DynamoDB.



Tipos de BBDD NoSQL

Grafos: Los datos se almacenan en una estructura de grafo con nodos, aristas y datos, pudiendo aplicar la teoría de grafos. Se utilizan para almacenar información sobre redes de datos, como las conexiones sociales. El ejemplo más conocido es Neo4j.



MongoDB

MongoDB es una BD orientada a documentos que utiliza documentos JSON para crear el esquema de la base de datos.

Es totalmente flexible ya que no obliga a diseñar un esquema antes de comenzar a registrar información. De esa forma, es fácil adaptarnos a los cambios que nuestros objetos puedan sufrir.

Comandos MongoDB

Comando	Consola MongoDB	Java
Crear/Conectar BD	use basedatos	mongoClient.getDatabase(basedatos);
Borrar BD	db.dropDatabase()	basedatos.drop();
Crear Colección	db.createCollection(coleccion, ops)	basedatos.createCollection(coleccion);
Borrar Colección	db.coleccion.drop()	basedatos.getCollection(coleccion).drop();
Seleccionar Colección	db.getCollection(coleccion)	basedatos.getCollection(coleccion);

Comandos MongoDB

Comando	Consola MongoDB	Java
Insertar Documento	db.coleccion.insert(doc) db.coleccion.insertOne(doc) db.coleccion.insertMany(docs)	coleccion.insertOne(doc) coleccion.insertMany(List<doc>)

Obtener Documentos	db.coleccion.find() // Todos
	db.coleccion.find(filtro)
	db.coleccion.findOne(filtro)
		coleccion.find()
		coleccion.find(filtro)
		coleccion.find(filtro).first()

Actualizar Documento	db.coleccion.update(filtro, set) db.coleccion.update(filtro, set, multi) db.coleccion.save(id, doc) db.coleccion.findOneAndUpdate(filtro, set) db.coleccion.updateOne(filtro, set) db.coleccion.updateMany(filtro, set)	coleccion.updateOne(filtro, set) coleccion.updateMany(filtro, set) coleccion.replaceOne(filtro, doc)
Borrar Documento	db.coleccion.remove(filtro) db.coleccion.remove(filtro, 1)	coleccion.deleteOne(filtro) coleccion.deleteMany(filtro)

Tipos de datos en MongoDB

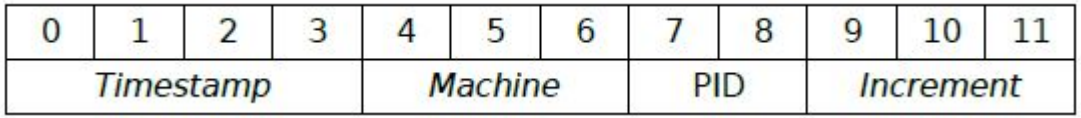
Podemos diferenciar los siguientes tipos de datos en MongoDB:

- null: representa tanto un valor nulo como un campo que no existe
- boolean: permite valores true y false
- number: por defecto, MongoDB utiliza floats. Para integers o longs se tiene que indicar de forma explícita
- string: cadena de caracteres válidos en UTF-8
- date: fechas en milisegundos desde el 1/1/1970
- array: listas de valores heterogéneos
- documentos embebidos: los documentos pueden contener otros documentos embebidos como valor de un campo

Tipo ObjectId

El tipo ObjectId es el tipo por defecto para el campo _id, y está diseñado para ser sencillo de generar valores únicos de forma global.

Para ello, utiliza 12 bytes de la siguiente manera:



- Timestamp: contienen información de fecha.
- Machine: hash determinado por el nombre de la máquina.
- PID: ID de proceso generado en el mismo segundo que el timestamp.
- Increment: autoincremental para mantener la unicidad dentro del mismo segundo.

Filtros en MongoDB

Comparadores

- eq (igual que), gt (mayor que), gte (mayor o igual que), lt (menor que), lte (menor o igual que), ne (distinto de), in (igual que un valor de un array), nin (distinto de todos los valores de un array)

Lógicos

- and (y), or (o), not (no y), nor (no o)

Para aplicar estos filtros, hay opciones basadas en el parseo de filtros a JSON... pero lo más cómodo es usar Filters del API de Java.

```
collection.find(and(gte("edad", 18), lt("edad", 23)));
```

Modelado de datos

Es posible modelar relaciones como las encontramos en las BBDD estructuradas.

Dos patrones de modelado:

- Embebido: consistirá en incrustar documentos uno dentro de otro, haciéndolo así parte del mismo registro, sería una relación directa
- Referencia: mediante este método se imitan las claves ajenas para relacionar los datos entre colecciones

Relaciones 1:1

Generalmente, se normaliza esta relación en una única tabla, aunque en casos especiales se separe en 2. En MongoDB, podemos utilizar el método embebido, como en el ejemplo:

```
Persona = {
        nombre: "Alejandro",
        apellido: "Roig",
        telefono: 999666333,
        direccion: {
                ciudad: "Castellon",
                calle: "Mayor",
                numero: 1,
                piso: 1,
                puerta: 1
        }
}
```

Relaciones 1:N

Se puede hacer tanto con referencias como embebido, que es lo preferible

```
Persona = {
        nombre: "Alejandro",
        apellido: "Roig",
        coches: [{
                matricula: "1111AAA",
                marca: "Seat",
        },{
                matricula: "2222BBB",
                marca: "Ford",
        }]
}
```

```
Persona = {
        nombre: "Alejandro",
        apellido: "Roig",
        coches: [1,2]
}
Coche1 = {
        _id: 1,
        matricula: "1111AAA",
        marca: "Seat",
}
Coche2 = {
        _id: 2,
        matricula: "2222BBB",
        marca: "Ford",
}
```

Relaciones N:N

Se utilizan referencias

```
Autor1 = {
        _id: 1,
        nombre: "Carlos",
        apellido: "González",
        libros: [{
                libro_id: 1000,
                orden: 1
        }]
}
Autor2 = {
        _id: 2,
        nombre: "Javier",
        apellido: "Albusac",
        libros: [{
                libro_id: 1000,
                orden: 2
        }, {
                libro_id: 1001,
                orden: 1
        }]
}
```

```
Libro1= {
        _id: 1000,
        titulo: "Desarrollo de Videojuegos",
        ibsn: "1517413389",
        autores: [1,2]
}
Libro2= {
        _id: 1001,
        titulo: "Programación Concurrente y Tiempo
Real",
        ibsn: "1518608264",
        autores: [2]
}
```

Validación de esquema

Anteriormente hemos destacado la flexibilidad de MongoDB al no obligar a diseñar un esquema para nuestras colecciones.

Sin embargo, en casos con requerimientos estructurales, podemos añadir reglas de validación de esquema. Veamos un ejemplo:

```
db.createCollection("profesores", {
        validator: { $jsonSchema: {
                bsonType: "object",
                required: ["dni", "nombre", "especialidad", "modulos"],
                properties: {
                        dni: {
                                bsonType: "string",
                                pattern: "^[0-9]{8}[A-Z]$"
                        },
                                                },
                        especialidad: {
                                bsonType: "string",
                                enum: ["PES", "PT"]
}}}});
```

Transacciones

Se estima que entre el 80%–90% de las aplicaciones que usan MongoDB no necesitan transacciones. Sin embargo, existen casos de uso en los cuales es absolutamente necesario.

MongoDB garantiza transacciones ACID a nivel de documento. Desde la v4, tenemos también transacciones ACID entre múltiples documentos.

```
try (ClientSession clientSession = client.startSession()) {
        clientSession.startTransaction();
        collection.insertOne(clientSession, docOne);
        collection.insertOne(clientSession, docTwo);
        clientSession.commitTransaction();
}
```