

UNIDAD DIDÁCTICA 7

USO DE BASES DE DATOS NO RELACIONALES

**MÓDULO PROFESIONAL:
BASES DE DATOS**



CESUR
Tu Centro Oficial de FP

Índice

RESUMEN INTRODUCTORIO	2
INTRODUCCIÓN	2
CASO INTRODUCTORIO	2
1. BASES DE DATOS NOSQL	4
2. CARACTERÍSTICAS, VENTAJAS Y DESVENTAJAS DE LAS BBDD NOSQL.....	6
3. TIPOS DE BBDD NOSQL Y ELEMENTOS COMUNES	10
4. SISTEMAS GESTORES DE BASES DE DATOS NOSQL	16
4.1 MongoDB	16
4.1.1 MongoDB Atlas.....	18
4.1.2 MongoDB Compass	19
4.1.3 Studio 3T.....	19
4.2 Redis.....	21
4.2.1 Redis CLI	22
4.3 Apache Cassandra	23
4.3.1 DataStax DevCenter	24
4.4 Neo4j.....	25
4.4.1 Neo4j Browser.....	26
5. MONGODB.....	27
5.1 Creación de Base de Datos	28
5.2 Inserción de documentos	30
5.3 Modificación de documentos	33
5.4 Búsqueda de documentos	37
5.4.1 Operador LIKE.....	39
5.4.2 Operadores AND, OR y NOT	40
5.4.3 Operadores \$gt, \$gte, \$lt, \$lte y \$eq	42
5.4.4 Acceso a valores de un array.....	42
5.4.5 Agregaciones	46
5.5 Eliminación de documento y colecciones.....	56
RESUMEN FINAL	59

RESUMEN INTRODUCTORIO

En la presente unidad se estudiará el concepto de las bases de datos no relacionales o NoSQL.

Fundamentalmente, la unidad pretende proporcionar al alumno una visión global de las características y el funcionamiento de este tipo de bases de datos, así como las diferentes tecnologías o tipos de ficheros utilizados para almacenar la información.

Para terminar, se pondrá el foco en los principales gestores de bases de datos, uno de cada tipo.

INTRODUCCIÓN

El crecimiento exponencial de datos en la era digital ha impulsado la necesidad de sistemas de almacenamiento de información más flexibles y escalables, lo que ha llevado al surgimiento de las bases de datos NoSQL. Estas bases de datos, a diferencia de las tradicionales bases de datos relacionales, ofrecen un enfoque más dinámico y adaptable para gestionar grandes volúmenes de datos no estructurados o semi-estructurados. La diversidad de modelos de datos y esquemas flexibles de las bases de datos NoSQL permite a las organizaciones manejar eficientemente datos heterogéneos, como documentos, gráficos etc.

El conocimiento y manejo de los diferentes aspectos y conceptos de las bases de datos NoSQL resulta esencial en el ámbito tecnológico actual. Estas bases de datos ofrecen soluciones especializadas para casos de uso específicos, como la gestión de datos en tiempo real, la escalabilidad horizontal y la flexibilidad en el desarrollo de aplicaciones. Comprender los fundamentos de las bases de datos NoSQL permite a los profesionales de TI y desarrolladores seleccionar la tecnología más adecuada para sus necesidades, optimizando el rendimiento, la escalabilidad y la confiabilidad de los sistemas de almacenamiento de datos en entornos cada vez más complejos y demandantes.

CASO INTRODUCTORIO

La empresa para la que trabajas está experimentando un grandísimo y rápido crecimiento. La empresa tiene millones de usuarios y un amplio catálogo de productos que se actualiza constantemente. Además, los usuarios generan una gran cantidad de interacciones diarias, como compras, comentarios, valoraciones y seguimiento de pedidos.

Debido a esto, vuestra base de datos actual está empezando a experimentar una gran lentitud y está dejando de ser eficiente. Es por ello que tus jefes te hacen responsable de encontrar una solución al problema, por lo que te dispones a informarte sobre un nuevo tipo de bases de datos existentes en el mercado conocidas como bases de datos no relacionales o NoSQL, así como de sus principales características, ventajas y desventajas con el objetivo de plantear la mejor solución posible para tu empresa.

Al finalizar la unidad, conocerás el concepto de base de datos no relacional o NoSQL, identificarás sus diferentes tipos de bases de datos y sus principales características. Además, conocerás los principales gestores de bases de datos y los diferentes tipos de herramientas para trabajar con ellas. Por último, serás capaz de realizar operaciones básicas con MongoDB.

1. BASES DE DATOS NOSQL

Paula, una de tus superiores, es la que te ha solicitado que te hagas cargo de la nueva tarea. Para ello, decides informarte sobre los tipos de bases de datos NoSQL por si una base de datos de este tipo pudiera resultar útil para tu empresa.

En el panorama actual, donde los datos son un recurso esencial y las aplicaciones generan cantidades masivas de información, la necesidad de sistemas de almacenamiento de datos eficientes y escalables se ha vuelto primordial. Es en este contexto que las bases de datos NoSQL han emergido como una solución disruptiva y versátil para enfrentar los desafíos de almacenamiento y gestión de datos en el siglo XXI.

El término "NoSQL" engloba un conjunto diverso de tecnologías y enfoques de almacenamiento que se alejan del paradigma tradicional de las bases de datos relacionales. Este término no significa que estas bases de datos estén en contra del SQL (Structured Query Language), sino que representan "Not Only SQL" o "No Solo SQL". Es decir, si bien algunas bases de datos NoSQL no utilizan SQL como lenguaje de consulta, otras pueden ofrecer soporte para él, además de proporcionar interfaces y modelos de datos más flexibles.

La premisa fundamental detrás de las bases de datos NoSQL es brindar una alternativa a las bases de datos SQL tradicionales, que históricamente se han basado en el modelo de tablas relacionales. Si bien este enfoque ha sido ampliamente exitoso durante décadas, también ha mostrado limitaciones en términos de escalabilidad, rendimiento y adaptabilidad a ciertos tipos de datos no estructurados o semiestructurados.

Las bases de datos NoSQL, al romper con el esquema relacional, adoptan diversos modelos de datos, lo que les permite abordar una amplia variedad de casos de uso y tipos de datos. Esta flexibilidad ha llevado a un crecimiento significativo de las bases de datos NoSQL en diferentes industrias y aplicaciones, incluyendo redes sociales, comercio electrónico, análisis de grandes volúmenes de datos, Internet de las cosas (IoT) y sistemas de recomendación, entre otros.

Además de proporcionar una mayor flexibilidad de esquema, las bases de datos NoSQL se destacan por su capacidad para escalar horizontalmente de manera más sencilla que las bases de datos relacionales. Al distribuir los datos en múltiples nodos y servidores, estas bases de datos pueden manejar grandes volúmenes de datos y altas cargas de trabajo, adaptándose fácilmente a las crecientes demandas de almacenamiento y procesamiento de datos en la era digital.

Asimismo, las bases de datos NoSQL han encontrado una fuerte afinidad con las arquitecturas en la nube y tecnologías modernas de desarrollo de aplicaciones. Su naturaleza distribuida se adapta bien a los enfoques de microservicios, contenedores y sistemas altamente escalables, comunes en la actualidad. Esta combinación de características ha posicionado a las bases de datos NoSQL como una opción atractiva para empresas y desarrolladores que buscan soluciones de almacenamiento de datos versátiles y altamente eficientes.

En conclusión, las bases de datos NoSQL representan un emocionante y dinámico campo dentro de la gestión de datos, ofreciendo una alternativa valiosa a las bases de datos relacionales tradicionales. Su capacidad para proporcionar escalabilidad horizontal, flexibilidad de esquema y un rendimiento óptimo en escenarios específicos ha llevado a una adopción cada vez mayor en diversos dominios de aplicación. Si bien no son adecuadas para todos los casos de uso, las bases de datos NoSQL son una herramienta poderosa y esencial para abordar los desafíos del almacenamiento y la gestión de datos en el entorno tecnológico actual en constante evolución.



PARA SABER MÁS

Amplía tu información sobre las bases de datos NoSQL:



ENLACE DE INTERÉS

Profundiza en las características más relevantes de las bases de datos NoSQL:



2. CARACTERÍSTICAS, VENTAJAS Y DESVENTAJAS DE LAS BBDD NOSQL

Te has informado mínimamente sobre las bases de datos NoSQL y piensas que podrían ser interesantes. Por este motivo, decides informarte más a fondo sobre las características de las mismas, así como de sus principales ventajas y desventajas para poder informar mejor a Paula.

En el ámbito de la gestión de datos, las bases de datos NoSQL han ganado cada vez más popularidad en las últimas décadas debido a su capacidad para abordar desafíos asociados con el crecimiento exponencial de datos y la necesidad de sistemas de almacenamiento más flexibles y eficientes. A diferencia de las tradicionales bases de datos relacionales, conocidas como SQL, las bases de datos NoSQL adoptan un enfoque no relacional y proporcionan diversas características que las hacen especialmente adecuadas para aplicaciones modernas y entornos con altas demandas de escalabilidad y rendimiento.

En este apartado, exploraremos las características distintivas de las bases de datos NoSQL. También examinaremos cómo estas características permiten a las bases de datos NoSQL adaptarse a los requisitos cambiantes y ofrecer soluciones efectivas para una variedad de aplicaciones, desde redes sociales y análisis de Big data hasta Internet de las cosas (IoT) y mucho más.

Las principales características de este tipo de bases de datos son:

- **Escalabilidad horizontal:** uno de los principales atractivos de las bases de datos NoSQL es su capacidad para escalar horizontalmente. Esto significa que podemos agregar nuevos nodos o servidores a la infraestructura para manejar aumentos en la carga de trabajo y el volumen de datos. En lugar de depender de un servidor único para toda la base de datos, se distribuye la carga entre múltiples nodos, lo que mejora el rendimiento y la disponibilidad.
- **Alto rendimiento y baja latencia:** las bases de datos NoSQL están optimizadas para rendir bien en entornos distribuidos y grandes conjuntos de datos. Gracias a su arquitectura distribuida y esquemas de datos más flexibles, pueden proporcionar tiempos de respuesta más rápidos y un mejor rendimiento en comparación con algunas bases de datos relacionales, especialmente en escenarios de lectura y escritura intensivas.

- **Datos no estructurados o semiestructurados:** mientras que las bases de datos SQL requieren un esquema fijo y tablas con columnas predefinidas, las bases de datos NoSQL permiten trabajar con datos no estructurados (imágenes, PDF, vídeos etc.) o semiestructurados (datos sin estructura fija, como ficheros XML o JSON). Esto es especialmente útil cuando los datos pueden variar en estructura o cuando no se conoce previamente el esquema exacto que se utilizará.
- **Alta disponibilidad y tolerancia a fallos:** la naturaleza distribuida de las bases de datos NoSQL permite que los datos estén replicados en varios nodos o centros de datos. Esto asegura que, incluso si falla un nodo o servidor, los datos seguirán estando disponibles en otros lugares. La tolerancia a fallos es crucial para aplicaciones que deben estar en funcionamiento de forma continua y no pueden permitirse largas interrupciones.
- **Flexibilidad en el crecimiento de datos:** con las bases de datos NoSQL, puedes agregar nuevos campos a los documentos o registros sin afectar los datos existentes. Esto facilita el crecimiento y la adaptación de la base de datos a medida que evoluciona la aplicación o se agregan nuevos requisitos.
- **Escalabilidad lineal:** la capacidad de escalar horizontalmente permite una escalabilidad lineal, lo que significa que agregar más recursos (nodos o servidores) proporciona un aumento proporcional en el rendimiento. Esto contrasta con algunas bases de datos relacionales que pueden enfrentar limitaciones en su capacidad de escalar a medida que crecen.
- **Menos dependencia de estructuras rígidas:** en las bases de datos NoSQL, los datos relacionados no necesitan estar en tablas separadas con relaciones complejas. Esta flexibilidad en la estructura de datos puede mejorar la eficiencia y simplicidad de algunas operaciones.
- **Mayor facilidad para integrarse con arquitecturas modernas:** las bases de datos NoSQL se ajustan bien a las arquitecturas modernas de aplicaciones, como las basadas en microservicios. Estos entornos favorecen la escalabilidad y modularidad, lo que encaja perfectamente con las capacidades de las bases de datos NoSQL.

En resumen, las bases de datos NoSQL son ideales para escenarios donde se necesita una alta escalabilidad, rendimiento y flexibilidad en la estructura de los datos. Su arquitectura distribuida y la capacidad de trabajar con datos no estructurados las hacen especialmente adecuadas para aplicaciones modernas y entornos con grandes volúmenes de información. Sin embargo, es importante tener en cuenta que no son una solución universal y la elección de una base de datos adecuada dependerá de las necesidades específicas de cada proyecto.

En este apartado, examinaremos las ventajas y desventajas de las bases de datos NoSQL. Exploraremos cómo su escalabilidad horizontal, flexibilidad en el esquema de datos y alto rendimiento ofrecen soluciones eficientes para aplicaciones modernas con grandes volúmenes de datos. Sin embargo, también analizaremos las desventajas asociadas, como la falta de estándares unificados y el modelo de consistencia eventual, para tomar decisiones informadas al seleccionar el tipo de base de datos más adecuado para cada proyecto.

Entre las principales ventajas cabe destacar:

- **Escalabilidad horizontal y rendimiento:** las bases de datos NoSQL están diseñadas para escalar horizontalmente, lo que les permite manejar grandes volúmenes de datos y una alta concurrencia. Esto se logra distribuyendo la carga en múltiples nodos o servidores, lo que mejora el rendimiento y garantiza una mayor capacidad de respuesta para aplicaciones con alto tráfico.
- **Flexibilidad en el esquema de datos:** a diferencia de las bases de datos SQL, las bases de datos NoSQL no requieren un esquema fijo y pueden manejar datos no estructurados o semiestructurados. Esto permite una mayor agilidad en el desarrollo y facilita la adaptación a cambios en los requisitos de datos sin requerir una reestructuración completa de la base de datos.
- **Manejo de grandes volúmenes de datos:** las bases de datos NoSQL son ideales para aplicaciones que generan y manejan grandes cantidades de datos, como redes sociales. La escalabilidad horizontal y la capacidad de distribuir la carga en múltiples nodos les permite gestionar eficientemente grandes conjuntos de información.
- **Alta disponibilidad y tolerancia a fallos:** gracias a su arquitectura distribuida y replicación de datos, ofrecen una alta disponibilidad y tolerancia a fallos. Si un nodo falla, los datos pueden ser accedidos y manipulados desde otros nodos, lo que garantiza que las aplicaciones continúen funcionando incluso en situaciones de fallo.

- **Rendimiento optimizado para operaciones específicas:** cada tipo de base de datos NoSQL está diseñado para tareas específicas, lo que les permite optimizar el rendimiento para operaciones particulares. Por ejemplo, las bases de datos de documentos son ideales para consultar y manejar datos semiestructurados, mientras que las bases de datos de clave-valor son excelentes para operaciones de lectura/escritura intensivas.

Como se puede apreciar, algunas de sus ventajas están estrechamente relacionadas con sus principales características.

Entre las desventajas que presentan se pueden mencionar las siguientes:

- **Falta de estándares:** a diferencia de las bases de datos SQL, que siguen estándares bien establecidos como SQL, las bases de datos NoSQL carecen de un estándar unificado. Cada tipo de base de datos NoSQL puede tener su propio conjunto de comandos y lenguaje de consulta, lo que puede dificultar la transición y el aprendizaje para los desarrolladores.
- **Consistencia eventual:** algunas bases de datos NoSQL utilizan el modelo de consistencia eventual, lo que significa que, después de una actualización, no todos los nodos pueden reflejar inmediatamente el cambio. Si bien esto puede mejorar el rendimiento, también puede llevar a situaciones donde los datos puedan estar temporalmente inconsistentes entre nodos.
- **Menos soporte de herramientas y comunidades más pequeñas:** aunque la popularidad de las bases de datos NoSQL ha crecido en los últimos años, todavía pueden tener menos soporte de herramientas y comunidades más pequeñas en comparación con las bases de datos SQL, que han estado en uso durante décadas.
- **Menos adecuadas para consultas complejas:** algunos tipos de bases de datos NoSQL, como las de clave-valor, pueden no ser ideales para consultas complejas que involucran múltiples tablas y relaciones. Para casos donde las consultas complejas son una parte fundamental de la aplicación, las bases de datos SQL pueden ser más adecuadas.

En conclusión, las bases de datos NoSQL ofrecen muchas ventajas en términos de escalabilidad, flexibilidad y rendimiento. Sin embargo, también presentan ciertas desventajas que deben ser consideradas al seleccionar el tipo de base de datos más adecuado para cada proyecto.



PARA SABER MÁS

Amplía tu información sobre las características de las bases de datos NoSQL:



ENLACE DE INTERÉS

También puedes consultar las características mediante un ejemplo:



3. TIPOS DE BBDD NOSQL Y ELEMENTOS COMUNES

En base a la información que has ido recopilando, empiezas a plantearte que, quizá en un futuro no muy lejano, podría ser bastante interesante empezar a trabajar con este tipo de bases de datos. Para estar más seguro, te propones buscar información sobre los diferentes tipos de bases de datos NoSQL.

Existen diferentes tipos de bases de datos NoSQL, cada uno de ellos diseñado para abordar diferentes necesidades y casos de uso específicos. A continuación, se describen brevemente los cuatro principales tipos de bases de datos NoSQL:

- **Bases de datos de documentos:** este tipo de bases de datos almacenan la información como un documento, usando principalmente una estructura simple como XML o JSON, o incluso BSON (Binary JSON), donde se usa una clave única por cada registro. Estas bases de datos son altamente flexibles y permiten que cada documento tenga diferentes estructuras de datos, lo que las hace ideales para almacenar datos semiestructurados o cambiantes en aplicaciones web y móviles. Las principales ventajas de este tipo de bases de datos son las siguientes:
 - Esquema flexible: cada documento puede tener un esquema diferente.
 - Alto rendimiento en consultas por clave: el acceso a documentos mediante una clave única es rápido.
 - Índices secundarios: permite indexar campos dentro de los documentos para mejorar el rendimiento de las consultas.
 - Escalabilidad horizontal: se puede agregar fácilmente más capacidad de almacenamiento distribuido para manejar grandes volúmenes de datos.

Algunos ejemplos de este tipo de bases de datos son **MongoDB** y **CouchDB**.

```
[
  {
    "id": 1,
    "nombre": "Marcos",
    "apellidos": "Suárez Jiménez",
    "empresa": "Limpiezas Suárez",
    "comisiones": {
      "enero": 2000,
      "febrero": 300
    }
  },
  {
    "id": 2,
    "nombre": "Luisa",
    "apellidos": "Pérez Oreal",
    "empresa": "Cárnicas Pérez",
    "comisiones": {
      "enero": 300,
      "febrero": 400
    }
  }
]
```

Ejemplo de fichero JSON.



ENLACE DE INTERÉS

Conoce más sobre las bases de documentales:



- **Bases de datos de clave-valor:** en estas bases de datos, los datos se almacenan en pares clave-valor, donde una clave única se asocia a un valor, que puede ser cualquier tipo de dato, desde un simple valor hasta una estructura más compleja como un documento JSON. Son ideales para aplicaciones con alta concurrencia y requisitos de lectura y escritura rápidas. Estas bases de datos son eficientes para almacenar y recuperar información a gran escala. Algunas de sus principales características son:
 - Eficaz para datos simples: son excelentes para cachés, sesiones y sistemas de recomendación.
 - Escalabilidad lineal: a medida que se agregan más nodos, el rendimiento aumenta linealmente.
 - Operaciones atómicas en clave-valor: Proporciona transacciones rápidas en un solo elemento.
 - Baja latencia: tiempos de respuesta rápidos debido a la simplicidad de la estructura de datos.

Algunos ejemplos destacados son **Redis** y **Riak**.



ENLACE DE INTERÉS

Profundiza sobre las bases de datos clave-valor:



- **Bases de datos de columnas:** a diferencia de las bases de datos SQL tradicionales, donde los datos se almacenan en filas, las bases de datos de columnas almacenan los datos en columnas en lugar de filas. Esta estructura optimizada permite una compresión más eficiente de los datos y un acceso rápido a subconjuntos de información. Son ideales para aplicaciones de análisis de datos y Big Data. Sus principales características son:
 - Esquema flexible por columna: Cada columna puede tener su propio tipo de datos.
 - Eficiencia en lecturas y agregaciones: Óptimas para consultas que involucran subconjuntos de columnas.
 - Compresión de datos: Pueden comprimir mejor los datos que las bases de datos tradicionales.
 - Alta escalabilidad: Pueden crecer horizontalmente para manejar grandes volúmenes de datos.

Algunos ejemplos de este tipo de base de datos son **Apache Cassandra** y **HBase**.



ENLACE DE INTERÉS

Consulta más información sobre las bases de datos de columnas:



- **Bases de datos de grafos:** estas bases de datos están diseñadas para trabajar con datos altamente interconectados, como redes sociales y sistemas de recomendación, rutas de navegación, entre otros. Almacenan los datos en forma de nodos (entidades) y relaciones (aristas) entre ellos, lo que permite consultas y análisis avanzados de las relaciones entre los datos. Sus principales ventajas son:
 - Modelado de relaciones complejas: los datos se organizan en estructuras de grafo, lo que permite representar relaciones entre nodos de manera eficiente.
 - Consultas avanzadas: pueden realizar consultas complejas para encontrar patrones y relaciones entre entidades.
 - Rendimiento en grafos densos: eficientes en grafos con muchas conexiones y relaciones.
 - Alto rendimiento para consultas de proximidad: buen rendimiento en consultas que involucran vecinos cercanos en el grafo.

Algunos ejemplos de estas bases de datos son **Neo4j** y **Amazon Neptune**.



ENLACE DE INTERÉS

Profundiza sobre las bases de datos de grafos:



Cada tipo de base de datos NoSQL tiene sus ventajas y desventajas, y la elección adecuada dependerá de los requisitos específicos de la aplicación y del modelo de datos que mejor se adapte a las necesidades del proyecto.



PARA SABER MÁS

También puedes conocer más sobre las bases de datos clave-valor aquí:



Los elementos de las bases de datos no relacionales varían según el tipo específico de base de datos que se esté utilizando. Sin embargo, hay ciertos elementos comunes que se encuentran en muchos de estos sistemas. Aquí están los principales:

- **Consultas y búsquedas:** todas las bases de datos NoSQL admiten operaciones para consultar y buscar datos. Dependiendo del tipo de base de datos, las operaciones de consulta pueden variar, pero en general, permiten recuperar información específica de acuerdo con los criterios especificados.

- **Indexación:** la mayoría de las bases de datos NoSQL admiten la creación de índices para mejorar el rendimiento de las consultas. Los índices pueden ser clave-valor, índices secundarios o índices de búsqueda de texto completo, dependiendo del tipo de base de datos.
- **Escalabilidad y replicación:** un elemento clave de muchas bases de datos NoSQL es su capacidad para escalar horizontalmente, lo que permite agregar más nodos para manejar aumentos en la carga de trabajo y el volumen de datos. Además, muchas bases de datos NoSQL admiten la replicación de datos para garantizar alta disponibilidad y tolerancia a fallos.

Es importante tener en cuenta que cada tipo de base de datos NoSQL tiene sus propias características y elementos específicos que los hacen adecuados para diferentes casos de uso. Al seleccionar una base de datos NoSQL, es fundamental comprender cómo se estructuran y gestionan los datos, y cómo se pueden aprovechar sus características para satisfacer las necesidades de una aplicación o proyecto en particular.

4. SISTEMAS GESTORES DE BASES DE DATOS NOSQL

A pesar del esfuerzo que supuso la anterior migración, viendo las ventajas y la tendencia del mercado, te planteas seriamente migrar tus datos a una base de datos NoSQL. Por ello, empiezas a documentarte sobre los principales gestores de bases de datos no relacionales.

En este apartado exploraremos algunos de los principales sistemas gestores de bases de datos NoSQL, como MongoDB, Cassandra, Redis o Neo4j entre otros. Cada uno de ellos ofrece características únicas y soluciones específicas para distintos tipos de aplicaciones y casos de uso.

4.1 MongoDB

MongoDB es una base de datos NoSQL orientada a documentos, diseñada para almacenar, recuperar y gestionar grandes volúmenes de datos en formato BSON (Binary JSON). Fue desarrollada por la empresa MongoDB Inc. y se lanzó por primera vez en 2009. Desde entonces, ha ganado una gran popularidad debido a su flexibilidad, rendimiento y escalabilidad, lo que lo convierte en una opción atractiva para una amplia variedad de aplicaciones y casos de uso.



Logo MongoDB.

Fuente: https://commons.wikimedia.org/wiki/File:MongoDB_Logo.png

Características clave de MongoDB:

- **Modelo de datos basado en documentos:** en lugar de utilizar tablas y filas como en las bases de datos SQL tradicionales, MongoDB almacena datos en documentos BSON. Cada documento es una estructura de datos flexible que puede contener campos con diferentes tipos de datos.
- **Alta escalabilidad y rendimiento:** MongoDB es una base de datos horizontalmente escalable, lo que significa que puede distribuirse en varios servidores para manejar grandes cargas de trabajo y grandes conjuntos de datos.
- **Indexación y consultas poderosas:** MongoDB permite crear índices en cualquier campo dentro de un documento, lo que acelera la velocidad de las consultas y mejora el rendimiento.
- **Replicación y alta disponibilidad:** MongoDB admite la replicación maestro-esclavo, lo que garantiza que los datos estén disponibles incluso si uno de los nodos del clúster falla.
- **Comunidad y soporte:** cuenta con una comunidad activa de usuarios y una amplia gama de recursos de soporte, incluida una extensa documentación, foros de discusión y la empresa MongoDB Inc., que ofrece servicios de asistencia técnica y herramientas adicionales.

En general, MongoDB se ha convertido en una elección popular para una amplia gama de aplicaciones, desde pequeños startups hasta grandes empresas, debido a su flexibilidad, rendimiento y escalabilidad.



PARA SABER MÁS

Profundiza sobre MongoDB conociendo más aspectos interesantes:



4.1.1 MongoDB Atlas

MongoDB Atlas es un servicio de base de datos gestionado y completamente administrado ofrecido por MongoDB Inc. Es una plataforma en la nube que permite a los usuarios implementar, gestionar y escalar clústeres de bases de datos MongoDB de forma sencilla y sin preocuparse por la infraestructura subyacente. Algunas de sus principales características son:

- **Gestión y administración simplificadas:** MongoDB Atlas se encarga de todas las tareas de administración de bases de datos, como aprovisionamiento de hardware, configuración, instalación de parches y copias de seguridad.
- **Alta disponibilidad y replicación:** MongoDB Atlas ofrece replicación automática de los datos para garantizar la alta disponibilidad.
- **Seguridad avanzada:** MongoDB Atlas proporciona características de seguridad robustas, incluyendo conexiones cifradas con TLS/SSL y opciones para autenticación avanzada. También es posible configurar reglas de acceso basadas en direcciones IP.
- **Integración con servicios en la nube:** MongoDB Atlas se integra fácilmente con otros servicios de nube, como AWS, Azure y Google Cloud.
- **Monitoreo y análisis:** la plataforma ofrece herramientas de monitoreo y análisis para supervisar el rendimiento de la base de datos y obtener información valiosa sobre el comportamiento de las consultas y operaciones.

4.1.2 MongoDB Compass

Es el equivalente a MongoDB Atlas para versión de escritorio. Es una interfaz gráfica de usuario (GUI) desarrollada por MongoDB Inc. que permite a los usuarios interactuar con bases de datos MongoDB de una manera intuitiva y visual. Es una herramienta poderosa y fácil de usar que facilita la administración y el análisis de datos almacenados en clústeres de MongoDB. Sus principales características son:

- **Exploración y visualización de datos:** proporciona una forma gráfica para explorar las colecciones de datos almacenadas en la base de datos. Permite navegar a través de documentos, examinar campos y ver la estructura de los datos almacenados de manera clara y sencilla.
- **Construcción de consultas:** permite construir y ejecutar consultas sobre los datos almacenados en MongoDB utilizando una interfaz de arrastrar y soltar.
- **Índices y optimización:** ofrece información sobre los índices creados en las colecciones y sugiere índices adicionales que podrían mejorar el rendimiento de las consultas. También proporciona herramientas para analizar el rendimiento de las consultas y sugerir formas de optimizarlas.
- **Importación y exportación de datos:** permite importar y exportar datos entre la base de datos y archivos JSON o CSV, lo que facilita la migración de datos y la integración con otras herramientas.
- **Seguridad y auditoría:** proporciona herramientas para configurar la autenticación y las reglas de acceso a la base de datos, lo que garantiza un acceso seguro a los mismos. También ofrece funcionalidades de auditoría para rastrear y revisar las operaciones realizadas en la base de datos.

4.1.3 Studio 3T

Studio 3T es una herramienta de interfaz gráfica de usuario (GUI) para MongoDB que facilita la administración y el desarrollo de bases de datos MongoDB. Es una suite de software que ofrece varias características para trabajar con MongoDB de manera más eficiente y cómoda. Algunas de sus principales características son:

- **Exploración y Visualización de Datos:** permite explorar y visualizar datos almacenados en bases de datos MongoDB a través de tablas, vistas de árbol, gráficos y otras representaciones visuales.

- **Editor de Consultas Avanzado:** ofrece un editor de consultas con resaltado de sintaxis, autocompletado y herramientas para facilitar la escritura y ejecución de consultas MongoDB.
- **Generación de Código:** Studio 3T puede generar código en varios lenguajes de programación (como JavaScript, Python y Java) a partir de las consultas y operaciones que realizas en la herramienta.

Studio 3T ofrece tanto una versión gratuita con funcionalidades limitadas como una versión comercial más avanzada con todas las características disponibles. Es una herramienta útil para desarrolladores, administradores de bases de datos y cualquier persona que trabaje con MongoDB y busque una interfaz más amigable y productiva para sus tareas diarias.



PARA SABER MÁS

Comprueba y realiza el siguiente tutorial para conocer más aspectos sobre MongoDB:



ENLACE DE INTERÉS

Puedes descargar Studio 3T aquí:



4.2 Redis

Redis es una popular base de datos en memoria de código abierto, que se utiliza como almacén de estructura de datos en tiempo real, caché y motor de mensajería. Su nombre es una abreviatura de "Remote Dictionary Server" (Servidor de Diccionario Remoto).

Una de las razones principales de su popularidad es su simplicidad y facilidad de uso. Su API es sencilla y consistente, lo que hace que sea fácil de aprender y utilizar para desarrolladores. Es ampliamente utilizado en aplicaciones web para mejorar el rendimiento y reducir la carga en las bases de datos tradicionales. Al actuar como una caché de datos en memoria, puede acelerar considerablemente las operaciones de lectura, disminuyendo la necesidad de acceder a una base de datos más lenta.

En aplicaciones en tiempo real, Redis es una opción popular para almacenar y procesar datos que necesitan una baja latencia. Su capacidad para manejar operaciones atómicas permite realizar transacciones de manera segura, lo que es esencial en escenarios donde la integridad de los datos es crítica.

Además de las características mencionadas, Redis cuenta con una activa comunidad de desarrolladores y usuarios que contribuyen con mejoras y módulos adicionales para extender aún más sus funcionalidades.

Es importante mencionar que, debido a que Redis es una base de datos en memoria, la cantidad de datos que puede almacenar está limitada por la memoria del sistema donde se ejecuta. Además, si el sistema experimenta un apagón o se reinicia, se perderán los datos no persistidos.

En resumen, Redis es una base de datos en memoria que destaca por su rendimiento, simplicidad y versatilidad, y se ha convertido en una herramienta esencial para desarrolladores y arquitectos de sistemas que buscan una solución rápida y fiable para almacenar y manipular datos en tiempo real.



Logo redis.

Fuente: <https://commons.wikimedia.org/wiki/File:Logo-redis.svg>



PARA SABER MÁS

Consulta más aspectos sobre Redis:



4.2.1 Redis CLI

Redis CLI (Command-Line Interface) es una herramienta que permite interactuar con una instancia de Redis a través de la línea de comandos. Es una forma rápida y conveniente de administrar, consultar y modificar datos en una base de datos Redis sin necesidad de una interfaz gráfica o una aplicación cliente más compleja. Algunas características importantes de Redis CLI incluyen:

- **Acceso directo:** Redis CLI proporciona una manera directa de comunicarse con el servidor Redis, simplemente ejecutando comandos desde la línea de comandos.
- **Interfaz de línea de comandos:** como su nombre indica, Redis CLI es una interfaz basada en texto. Para interactuar con él, debes escribir comandos en la línea de comandos y presionar "Enter" para enviarlos al servidor Redis.
- **Versatilidad:** Redis CLI es capaz de manejar una amplia variedad de operaciones, desde comandos básicos como SET, GET, DELETE, hasta operaciones más avanzadas que involucran estructuras de datos complejas, como listas, conjuntos, hashes y conjuntos ordenados.
- **Facilidad de scripting y automatización:** Redis CLI es útil para scripting y automatización de tareas, ya que puedes combinar múltiples comandos Redis en secuencias para realizar operaciones más complejas o realizar tareas repetitivas.
- **Visualización de resultados:** muestra las respuestas del servidor en formato de texto, lo que facilita la visualización de los datos recuperados o los resultados de las operaciones.

Es importante mencionar que Redis CLI debe estar instalado y disponible en el sistema donde se encuentra la instancia de Redis a la que se desea conectar. Se puede iniciar Redis CLI simplemente ejecutando el comando 'redis-cli' en la línea de comandos.

4.3 Apache Cassandra

Apache Cassandra es una base de datos NoSQL de código abierto que fue originalmente desarrollada por Facebook y luego donada a la Apache Software Foundation. Cassandra utiliza un modelo de datos distribuido basado en columnas y es conocida por su capacidad de escalar horizontalmente para manejar grandes volúmenes de datos.

Esta base de datos se ha diseñado para brindar alta disponibilidad y tolerancia a fallos mediante la replicación de datos en múltiples nodos del clúster. Cada nodo en el clúster tiene la misma función y almacena una parte del conjunto de datos, lo que permite que Cassandra se mantenga operativa incluso si algunos nodos fallan.

Cassandra ofrece una capa de abstracción llamada CQL (Cassandra Query Language) que permite a los usuarios interactuar con la base de datos utilizando un lenguaje similar a SQL, lo que facilita la transición para aquellos familiarizados con bases de datos relacionales.

Esta base de datos se adapta bien a aplicaciones web, análisis en tiempo real y sistemas que requieran una alta disponibilidad y una escalabilidad horizontal para manejar grandes volúmenes de datos. Su capacidad de integrarse con otras herramientas, como Apache Spark y Apache Hadoop, también la hace atractiva para aplicaciones de análisis de datos a gran escala.

En resumen, Apache Cassandra es una base de datos NoSQL distribuida y escalable que ofrece una alta disponibilidad, tolerancia a fallos y un modelo de datos basado en columnas. Su versatilidad y rendimiento la convierten en una opción popular para una variedad de aplicaciones que manejan grandes volúmenes de datos en entornos distribuidos y de alto rendimiento.



Logo redis.

Fuente: <https://commons.wikimedia.org/wiki/File:Apache-cassandra-icon.png>



PARA SABER MÁS

Conoce más sobre Apache Cassandra mediante una interesante introducción:



4.3.1 DataStax DevCenter

DataStax DevCenter es una herramienta de interfaz gráfica (GUI) desarrollada por DataStax, una empresa especializada en soluciones de datos basadas en Apache Cassandra. DevCenter está diseñado para facilitar la administración, desarrollo y depuración de aplicaciones que utilizan Cassandra como base de datos. Algunas de sus características más importantes son:

- **Interfaz gráfica intuitiva:** ofrece una interfaz gráfica de usuario que permite a los desarrolladores y administradores interactuar con Cassandra de manera intuitiva.
- **Ejecución de consultas CQL:** DevCenter proporciona un editor de consultas CQL (Cassandra Query Language) que permite a los usuarios escribir y ejecutar consultas directamente en la herramienta. Esto facilita la exploración y manipulación de datos almacenados en Cassandra.
- **Depuración y visualización de resultados:** cuando se ejecutan consultas en DevCenter, la herramienta muestra los resultados de manera clara y estructurada, lo que facilita la visualización de los datos recuperados de la base de datos.
- **Importación y exportación de datos:** DevCenter permite importar y exportar datos en formatos como CSV, JSON y XML, lo que facilita la carga y descarga masiva de datos desde y hacia Cassandra.
- **Configuración de conexiones:** permite a los usuarios configurar múltiples conexiones a diferentes clústeres de Cassandra, lo que es útil para trabajar con entornos de desarrollo y producción.

DataStax DevCenter es una herramienta valiosa para desarrolladores, administradores y cualquier persona que trabaje con Cassandra, facilitando el proceso de desarrollo, depuración y gestión de aplicaciones basadas en esta base de datos.

4.4 Neo4j

Neo4j es una base de datos de grafos de código abierto, que está diseñada específicamente para almacenar y consultar datos en forma de grafos. Éstos, son una estructura de datos que consta de nodos (que representan entidades) y relaciones (que representan las conexiones o interacciones entre los nodos). Esta base de datos se basa en el lenguaje de consulta de grafos llamado Cypher, que permite realizar consultas y análisis complejos de manera eficiente.

Entre las características de Neo4j, cabe destacar las siguientes:

- **Escalabilidad y rendimiento:** Neo4j ofrece una alta escalabilidad y rendimiento para manejar grandes volúmenes de datos y consultas complejas. Es especialmente eficiente para consultas que involucran búsquedas de patrones en grafos.
- **Lenguaje de consulta Cypher:** es un lenguaje de consulta específico para grafos, que permite a los usuarios expresar consultas de manera intuitiva y sencilla. Su sintaxis se asemeja al lenguaje natural y es fácil de aprender, lo que facilita la interacción con la base de datos.
- **Transacciones ACID:** garantiza la integridad de los datos mediante transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), lo que asegura que las operaciones se realicen de manera segura y consistente.
- **Comunidad y soporte:** cuenta con una comunidad activa de usuarios y desarrolladores, así como con soporte profesional ofrecido por la empresa Neo4j, que proporciona asistencia para proyectos.

Neo4j es ampliamente utilizado en aplicaciones que involucran datos conectados, como redes sociales, recomendaciones de productos, análisis de redes, sistemas de recomendación y muchos otros casos de uso donde las relaciones entre datos son esenciales.



PARA SABER MÁS

Si quieres saber más sobre Neo4j puedes visitar este enlace:



4.4.1 Neo4j Browser

Neo4j Browser es una interfaz de usuario web que viene incluida con Neo4j. Es una herramienta poderosa y amigable que permite interactuar con la base de datos Neo4j, explorar y visualizar los datos en forma de grafos, y ejecutar consultas en lenguaje Cypher. Algunas características clave de Neo4j Browser son:

- **Interfaz gráfica interactiva:** Neo4j Browser ofrece una interfaz gráfica interactiva y fácil de usar que permite a los usuarios explorar la base de datos y sus relaciones mediante una visualización de grafos.
- **Consultas en lenguaje Cypher.**
- **Visualización de resultados:** cuando se ejecutan consultas, Neo4j Browser muestra los resultados en una tabla o visualización gráfica, lo que facilita la comprensión y el análisis de los datos recuperados.
- **Historial de comandos:** el browser mantiene un historial de los comandos ejecutados, lo que permite a los usuarios revisar y repetir consultas anteriores fácilmente.
- **Integración con Neo4j Desktop:** Neo4j Browser se integra con Neo4j Desktop, una herramienta de administración y desarrollo para proyectos de Neo4j. Esto facilita la configuración y administración de la base de datos.

Neo4j Browser es una herramienta valiosa para desarrolladores, administradores y analistas que trabajan con Neo4j. Proporciona una forma intuitiva de interactuar con la base de datos de grafos y permite una exploración y visualización efectiva de los datos conectados. Su capacidad para ejecutar consultas Cypher en tiempo real lo convierte en una herramienta esencial para el desarrollo y análisis de aplicaciones basadas en grafos.

5. MONGODB

Paula y tú os habéis decidido por implantar MongoDB en vuestra empresa, por lo que te encarga que vayas creando las primeras colecciones y documentos para ello. Además, te pide que empieces a pensar cómo se deben hacer las consultas o su equivalente en esta base de datos.

En este apartado vamos a ver algunos ejemplos sencillos de cómo podemos trabajar con MongoDB, que es el base de datos NoSQL más utilizada actualmente.

Antes de empezar, debemos saber que MongoDB trabaja con colecciones (equivalente a las tablas del modelo relacional) y estas colecciones contienen, a su vez, documentos (equivalente al registro en el modelo relacional).

Una vez aclarado esto, el primer paso sería crear una base de datos. Para ello, la forma más cómoda es usar la interfaz gráfica.



ENLACE DE INTERÉS

Descarga MongoDB desde su página oficial:





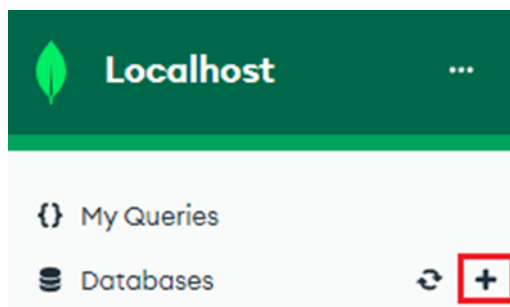
VÍDEO DE INTERÉS

Visualiza cómo instalar MongoDB de manera rápida y sencilla:



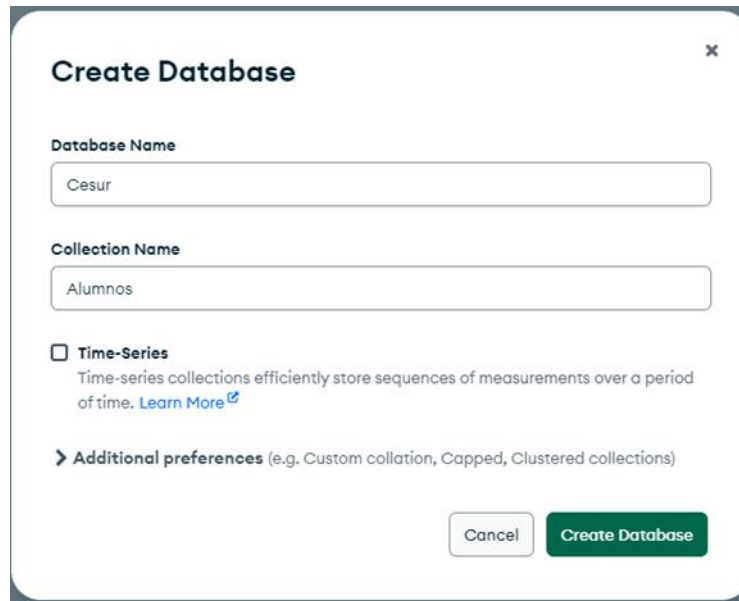
5.1 Creación de Base de Datos

Para crear una base de datos en MongoDB, debemos pulsar en el botón correspondiente.



Creación de base de datos en MongoDB.

Y en la siguiente ventana introduciremos el nombre de la base de datos y de nuestra primera colección.



Create Database

Database Name
Cesur

Collection Name
Alumnos

☐ **Time-Series**
Time-series collections efficiently store sequences of measurements over a period of time. [Learn More](#)

[Additional preferences](#) (e.g. Custom collation, Capped, Clustered collections)

Cancel Create Database

Creación de bases de datos en MongoDB.

Una vez creadas la base de datos y la colección llamada Alumnos, podemos seleccionar esta última para añadir documentos. Es importante recordar que aquí ya no usaremos las funciones SQL o similares a las que estamos acostumbrados, ya que ahora estamos trabajando con una base de datos documental, por lo que insertaremos los nuevos documentos usando un formato JSON.

Para crear la base de datos mediante comando simplemente debemos escribir `use NombreBaseDatos`, si la base de datos con ese nombre ya existe, se mueve a ella, si no, la crea. Es importante señalar que, hasta que no insertemos un valor en una colección, no podremos ver esa base de datos.

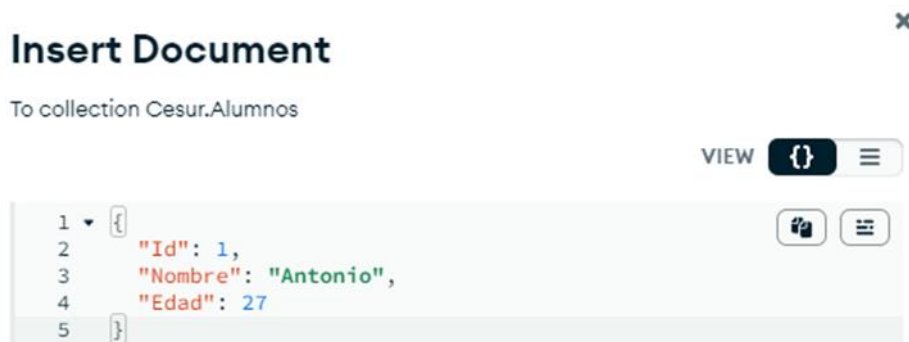
Por otro lado, las colecciones no hace falta crearlas explícitamente ya que, en MongoDB, las colecciones se crean automáticamente cuando insertas el primer documento en ellas. Si quisiéramos crear expresamente la colección, debemos usar la siguiente sintaxis: `db.createCollection("NombreColeccion")`.

Por ejemplo: `db.createCollection("Vehiculos")`.

Es importante resaltar que MongoDB hace distinción entre mayúsculas y minúsculas por lo que, si hemos creado la colección Vehículos (con v mayúscula), siempre tendremos que escribirlo igual. Lo mismo ocurre con los nombres de los campos de las colecciones.

5.2 Inserción de documentos

Para este primer caso insertaremos un ejemplo sencillo.



Inserción de documento en MongoDB.

Una vez insertado el documento, existen diferentes formas de visualizarlo:



Diferentes vistas de una colección.

La parte izquierda de la imagen muestra una sección por documento, mientras que la parte derecha muestra el contenido de la colección en formato JSON. Por último, tenemos también la opción de una vista en forma de tabla.

Alumnos			
	_id ObjectId	Id Int32	Nombre String
1	ObjectId('64d1f5083eeb2226250...	1	"Antonio"

Vista en formato tabla.

Como se puede apreciar, MongoDB ha añadido automáticamente al documento un ObjectId, que es único para cada uno de los documentos y con el cual puede identificarlos.

Para el siguiente ejemplo vamos a insertar dos nuevos documentos al mismo tiempo. Para ello, debemos insertarlos en forma de array de objetos. Los corchetes delimitan el inicio y fin del array mientras que las llaves identifican los diferentes objetos del mismo.

```
1  [
2  {
3    "Id":2,
4    "Nombre":"María",
5    "Edad":19
6  },
7  {
8    "Id":3,
9    "Nombre":"Laura",
10   "Edad":47,
11   "Dirección": "Calle Gran Vía"
12  }
13 ]
```

Inserción múltiple.

Nótese que para el segundo documento se ha añadido un nuevo campo que no estaba presente en los otros dos. Esta es una de las ventajas que supone no estar limitado por la rigidez del modelo relacional ya que, al definir una tabla, todos los registros que se encuentren en ella tendrán obligatoriamente los mismos campos.

```
_id: ObjectId('64d1f5083eeb2226250ee52a')
Id: 1
Nombre: "Antonio"
Edad: 27
```

```
_id: ObjectId('64d1f97d3eeb2226250ee52d')
Id: 2
Nombre: "María"
Edad: 19
```

```
_id: ObjectId('64d1f97d3eeb2226250ee52e')
Id: 3
Nombre: "Laura"
Edad: 47
Dirección: "Calle Gran Vía"
```

Lista de documentos.

Esto mismo podemos hacerlo desde la consola de comandos, aunque dependiendo de la versión instalada de MongoDB, podremos o no utilizarla. Para insertar un único documento se debe usar el comando **insertOne()** con el siguiente formato: **db.NombreColección.insertOne({objeto JSON})**.

```
db.Alumnos.insertOne(
  {
    "Id":1,
    "Nombre":"Antonio",
    "Edad":27
  }
)
```


Si lo que necesitamos es insertar varios documentos al mismo tiempo debemos usar la función **insertMany()** con el siguiente formato:

db.NombreColección.insertMany({Array objetos JSON}).

```
db.Alumnos.insertMany([
  {
    "Id":2,
    "Nombre":"María",
    "Edad":19
  },
  {
    "Id":3,
    "Nombre":"Laura",
    "Edad":47
  }
])
```



EJEMPLO PRÁCTICO

Hemos recibido dos nuevas matrículas para el curso y debemos dar de alta a esos dos alumnos en la base de datos. ¿Cómo se realizaría?

Solución:

```
db.Alumnos.insertMany([
  {
    "Id":4,
    "Nombre":"David",
    "Edad":35,
    "Ciudad":"Murcia"
  },
  {
    "Id":5,
    "Nombre":"Marta",
    "Edad":27,
    "Ciudad":"Almería"
  }
])
```

5.3 Modificación de documentos

Veamos ahora un ejemplo un poco más complejo. Imaginemos que queremos guardar todas las asignaturas de las que está matriculado un alumno. Para poder almacenar esta información, estas asignaturas deben manejarse dentro de un array. En este caso, vamos a modificar el primer documento que insertamos. Para ello, podemos pulsar sobre el botón modificar del documento en cuestión.



Botón para edición de documento.

Ahora es cuando podemos añadir el array con las asignaturas.

```
1 {
2   "_id": {
3     "$oid": "64d1f5083eeb2226250ee52a"
4   },
5   "Id": 1,
6   "Nombre": "Antonio",
7   "Edad": 27,
8   "Asignaturas": ["Base de datos", "Programación", "Entornos de Desarrollo"]
9 }
```

Array de asignaturas.

Imaginemos ahora que, además de la asignatura, queremos almacenar también la nota de cada una de ellas. Para ello, dentro del array Asignaturas, debemos crear un objeto por cada asignatura, el cual contenga los campos Nombre y Calificación.

```
1 {
2   "_id": {
3     "$oid": "64d1f5083eeb2226250ee52a"
4   },
5   "Id": 1,
6   "Nombre": "Antonio",
7   "Edad": 27,
8   "Asignaturas": [
9     {
10      "Nombre": "Base de datos",
11      "Calificación": 8.5
12    },
13    {
14      "Nombre": "Programación",
15      "Calificación": 10
16    },
17    {
18      "Nombre": "Entornos de Desarrollo",
19      "Calificación": 7.5
20    }
21  ]
22 }
```

Alumnos con asignaturas y calificaciones.

Para poder representar esto en el modelo relacional, hubiésemos necesitado tres tablas: Alumnos, Asignaturas y Calificaciones. Esta última tabla tendría un PK compuesta por Id_Alumno e Id_Asignatura, además del campo para la calificación.

Al igual que en el caso de las inserciones, podemos hacer modificaciones desde la consola de comandos. Para modificar un solo documento, disponemos de la función **updateOne()**, con el siguiente formato:

```
db.collection.updateOne(  
  <filter>,  
  <update>,  
  $set {<campo1>:<valor1>, ...}  
  {  
    upsert: <boolean>,  
    writeConcern: <document>,  
    collation: <document>,  
    arrayFilters: [ <filterdocument1>, ...]  
  }  
)
```

A continuación, se explican los diferentes parámetros que se pueden usar en la función:

- **<filter>**: es el criterio por el que se va a filtrar la modificación, en otras palabras, el WHERE de un UPDATE en SQL.
- **<update>**: son las modificaciones a aplicar, es decir, el SET en un UPDATE en SQL.
- **upsert**: es opcional y, si está a true, crea un nuevo documento si no encuentra uno que coincida con los criterios establecidos en el apartado <filter>. Por defecto su valor es false.
- **writeConcern**: es una configuración que determina el nivel de garantía que se solicita del servidor de base de datos después de realizar una operación de escritura, como la inserción, actualización o eliminación de documentos. También es opcional.
- **collation**: es opcional y permite a los usuarios especificar reglas específicas del lenguaje (tildes, la letra ñ etc.).
- **arrayFilters**: un conjunto de documentos de filtro que determina qué elementos del array se modificarán en una operación de actualización en un campo de tipo array. Es opcional.

Es importante mencionar que, si el criterio de filtro establecido coincidiera con más de un documento, `updateOne()` modificaría únicamente el primero que encontrara.

Sin embargo, si lo que necesitamos es modificar varios documentos a la vez, podemos utilizar la función **`updateMany()`** con el siguiente formato:

```
db.collection.updateMany(  
  <filter>,  
  <update>,  
  $set {<campo1>:<valor1>, ...}  
  {  
    upsert: <boolean>,  
    writeConcern: <document>,  
    collation: <document>,  
    arrayFilters: [ <filterdocument1>, ... ],  
    hint: <document|string>})
```

Como se puede apreciar, los parámetros son los mismos excepto por el último de ellos. Hint es un parámetro opcional que permite especificar el índice a utilizar.



EJEMPLO PRÁCTICO

Nos hemos dado cuenta de que la edad de Antonio, uno de los alumnos, es incorrecta, por lo que es necesario corregirla. Su edad real es 25. Debes escribir el comando necesario para hacer la corrección. ¿Cómo procederías?

Solución:

```
db.Alumnos.updateOne({Id:1}, {$set: {Edad:25}}).
```

Si quisiéramos modificar más campos, por ejemplo, el nombre, se haría de la siguiente manera:

```
db.Alumnos.updateOne({Id:1}, {$set: {Edad:25, Nombre:"Pepe"}}).
```



EJEMPLO PRÁCTICO

Queremos añadir el campo Ciudad a la colección Alumnos, que tendrá por defecto el valor Madrid. ¿Cómo se realizaría?

Solución:

```
db.Alumnos.updateMany(  
  {}, // Sin filtro, se aplicará a todos los documentos  
  { $set: { Ciudad: "Madrid" } } // Agregar un nuevo campo a todos  
  los documentos )
```



EJEMPLO PRÁCTICO

Si quisiéramos añadir el nuevo campo solo a algunos documentos podríamos añadir el filtro deseado en la primera línea, por ejemplo:

Solución:

```
db.Alumnos.updateMany(  
  { edad: { $gt: 20 } }, // solo se aplicará a los alumnos con más de  
  20 años  
  { $set: { Ciudad: "Madrid" } } // Agregar un nuevo campo a los  
  documentos que cumplen el filtro  
  )
```



EJEMPLO PRÁCTICO

Queremos decrementar en 1 la edad de todos los alumnos que sean de Madrid. ¿Cómo se realizaría?

Solución:

```
db.Alumnos.updateMany(  
  { Ciudad: "Madrid" }, // Filtro para seleccionar los alumnos de  
  Madrid  
  { $inc: { edad: -1 } } // Actualización para decrementar la edad  
  en 1  
  )
```



EJEMPLO PRÁCTICO

Queremos incrementar en 1 la calificación de la asignatura "Historia" para el DNI "34567890M". ¿Cómo procederías?

Solución:

```
db.Coleccion.updateOne(  
  { "dni": "34567890M",  
  "Asignaturas.Nombre": "Historia" },  
  { $inc: {  
    "Asignaturas.$.Calificación": 1 } })  
{ $inc: { "Asignaturas.$.Calificación": 1 } }:
```

En este caso, \$ se utiliza para referenciar el elemento del array que coincidió con el filtro. En este caso, "Asignaturas.\$.Calificación" se refiere a la calificación de la asignatura "Historia".

5.4 Búsqueda de documentos

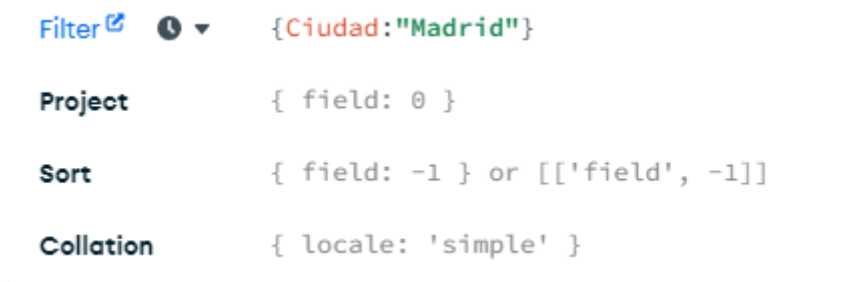
Lógicamente, MongoDB también ofrece la posibilidad de realizar consultas, tanto de manera gráfica si usamos Compass, como de manera manual si usamos por ejemplo Studio 3T o la consola.

Siguiendo con el ejemplo de los alumnos, supongamos que tenemos la siguiente estructura:

```
{  
  "Nombre": "Juan",  
  "Apellidos": "García Pérez",  
  "edad": 18,  
  "Ciudad": "Madrid",  
  "Dirección": "Calle Principal, 123",  
  "dni": "12345678A",  
  "Asignaturas": [  
    { "Nombre": "Matemáticas", "Calificación": 8 },  
    { "Nombre": "Historia", "Calificación": 7 },  
    { "Nombre": "Inglés", "Calificación": 9 }  
  ]  
},
```

Estructura objeto Alumno.

Supongamos que queremos mostrar todos los alumnos que son de Madrid. De manera gráfica, usando Compass, lo haríamos de la siguiente manera:



Filtrado donde la ciudad es Madrid.

A continuación, se explican las cuatro secciones que aparecen en la imagen:

- **Filter:** son los filtros que se aplican a la consulta.
- **Project:** son los campos que queremos ver en la consulta.
- **Sort:** sirve para indicar un campo de ordenación.
- **Collation:** podemos especificar si queremos que tenga en cuenta los acentos, mayúsculas etc.

Para realizar búsquedas mediante el uso de comandos, debemos usar la función **find()**.

El ejemplo anterior quedaría de la siguiente manera:

```
db.Alumnos.find({Ciudad: "Madrid"}).
```

Siguiendo con el ejemplo anterior, supongamos que sólo queremos ver los campos Nombre, Apellidos y DNI. De manera gráfica quedaría así:



Selección de campos.

Se debe especificar con el valor 1 qué campos se deben devolver en la búsqueda. A excepción del campo interno *_id*, no es necesario especificar con un 0 el hecho de que no se quiere ese campo en la consulta.

La misma búsqueda hecha mediante comandos resultaría de la siguiente manera:

```
db.Alumnos.find(  
  {Ciudad: "Madrid"}, // Filtro para seleccionar los alumnos de  
  Madrid  
  {Nombre: 1, Apellidos: 1, dni: 1, _id: 0} // Proyección para  
  mostrar solo los campos deseados  
)
```

Para ordenar estos resultados por el campo nombre de manera ascendente:

Filter	{Ciudad:"Madrid"}
Project	{Nombre:1 , Apellidos:1, dni:1, _id:0}
Sort	{Nombre:1}
Collation	{ locale: 'simple' }

Ordenación de campos.

Para ordenarlo de manera descendente, en la sección Sort se debería poner Nombre: -1.

Para hacer los mismo desde la consola de comandos:

```
db.Alumnos.find(  
  { Ciudad: "Madrid" },  
  { Nombre: 1, Apellidos: 1, dni: 1, _id: 0 }  
) .sort({Nombre:1})
```

5.4.1 Operador LIKE

En MongoDB no existe el operador LIKE tal y como lo conocemos en el lenguaje SQL. Para obtener el resultado equivalente, se deben usar expresiones regulares. De esta forma, tendremos las siguientes equivalencias con el operador LIKE:

- **cadena% → /^cadena/**: coincidencias que empiecen con cadena.
- **%cadena% → /cadena/**: que contenga la cadena.
- **%cadena → /cadena\$/**: que termine por cadena.

Además, podemos añadir el operador i después de la segunda barra para indicar que no sea sensible a mayúsculas y minúsculas.

Para obtener los alumnos cuya ciudad empiece por M ordenados por Edad y Nombre:

Filter	{Ciudad:/^m/i}
Project	{ field: 0 }
Sort	{edad:1, Nombre:1}
Collation	{ locale: 'simple' }

Alumnos cuya Ciudad empieza por M ordenados por edad y nombre.

El equivalente de esta operación en comando quedaría así:

```
db.Alumnos.find(  
  { Ciudad:/^m/i}  
) .sort ({edad:1, Nombre:1})
```

5.4.2 Operadores AND, OR y NOT

Existen dos maneras distintas de usar el operador AND, de forma implícita y explícita. La forma implícita es usando la coma (,) como separador:

Filter	{Ciudad:/^m/i, Nombre:/n\$/i}
Project	{ field: 0 }
Sort	{ field: -1 } or [['field', -1]]
Collation	{ locale: 'simple' }

Filtro por Ciudad y Nombre.

En este caso, buscamos los alumnos cuya ciudad empiece por m y su nombre termine por n. El comando para esto es el siguiente:

```
db.Alumnos.find(  
  { Ciudad:/^m/i, Nombre:/n$/i}  
)
```

La forma explícita consiste en usar el operador \$and, donde el valor que se le pasa es un array con las diferentes condiciones de búsqueda, por ejemplo:

```
db.Alumnos.find(  
  {  
    $and: [{Ciudad:/^m/i}, {Nombre:/n$/i}]  
  })
```

Para los operadores OR y NOT no existe forma implícita por lo que se debe usar la nomenclatura anterior. Para ver los alumnos cuya ciudad sea Barcelona o Sevilla:

```
db.Alumnos.find(  
  {  
    $or: [{Ciudad:"Barcelona"}, {Ciudad:"Sevilla"}]  
  })
```

Si solo quisiéramos ver los campos Nombre, Apellidos, DNI y Ciudad:

```
db.Alumnos.find(  
  {  
    $or: [{Ciudad:"Barcelona"}, {Ciudad:"Sevilla"}]  
  },  
  {Nombre: 1, Apellidos: 1, dni: 1, Ciudad:1, _id: 0}  
)
```

Nótese que, al añadir este segundo bloque con los campos a visualizar, cada uno de ellos va entre llaves ({}).

Para obtener los alumnos cuyas ciudades no sean ni Barcelona ni Sevilla podemos usar el operador \$NOT de la siguiente manera:

```
db.Alumnos.find(  
  {  
    Ciudad: {  
      $not: {  
        $in: ["Barcelona", "Sevilla"]  
      }  
    }  
  },  
  {  
    Nombre: 1, Apellidos: 1, dni: 1, Ciudad:1, _id: 0  
  }  
)
```

En este caso, además de \$not, debemos usar el operador \$in para especificar un array con los valores a excluir.

5.4.3 Operadores \$gt, \$gte, \$lt, \$lte y \$eq

Estos son los operadores que Mongo ofrece para la comparación de valores.

- **\$gt**: es equivalente a mayor que:
- **\$gte**: mayor o igual que:
- **\$lt**: menor que:
- **\$lte**: menor o igual que:
- **\$eq**: igual que

Para ver los alumnos mayores de 17 años:

```
db.Alumnos.find(  
  {  
    edad: { $gt: 17 }  
  }  
)
```

Para obtener el nombre, apellidos, DNI y ciudad de los alumnos cuya ciudad no se ni Barcelona ni Sevilla y además sean mayores de 17 años:

```
db.Alumnos.find(  
  {  
    Ciudad: {  
      $not: {  
        $in: ["Barcelona", "Sevilla"]  
      }  
    },  
    edad: { $gt: 17 }  
  },  
  {  
    Nombre: 1, Apellidos: 1, dni: 1, Ciudad:1, _id: 0  
  }  
)
```

5.4.4 Acceso a valores de un array

Recordemos que uno de los campos que contiene la colección Alumnos es una array llamado Asignaturas. Para acceder a los campos de este array podemos proceder de varias maneras. Si por ejemplo quisiéramos mostrar los alumnos que han sacado más de un seis en historia, la manera más sencilla de hacerlo sería la siguiente:

```
db.Alumnos.find({  
  "Asignaturas.Nombre": "Historia",  
  "Asignaturas.Calificación": { $gt: 6 }  
})
```

Otra manera de hacerlo sería usar el operador `$elemMatch` de la siguiente manera:

```
db.Alumnos.find({
  "Asignaturas": {
    $elemMatch: {
      "Nombre": "Historia",
      "Calificación": { $gt: 6 }
    }
  }
})
```

\$elemMatch se usa especialmente cuando se trabaja con arrays en documentos y se necesita aplicar una condición a elementos dentro de ese array. Es útil cuando se desea filtrar documentos basados en una condición específica en un array anidado.

Si queremos añadir alguna condición más, por ejemplo, que muestre los alumnos que, o bien hayan cursado Historia y sacado más de un 6, o bien hayan obtenido menos de un 9 en Música.

```
db.Alumnos.find({
  "Asignaturas": {
    $elemMatch: {
      $or: [
        {
          "Nombre": "Historia",
          "Calificación": { $gt: 6 }
        },
        {
          "Nombre": "Música",
          "Calificación": { $lt: 9 }
        }
      ]
    }
  }
})
```

Por último, si quisiéramos mostrar solo unos valores concretos:

```
db.Alumnos.find({
  $or: [
    {
      "Asignaturas": {
        $elemMatch: {
          "Nombre": "Historia",
          "Calificación": { $gt: 6 }
        }
      }
    },
    {
      "Asignaturas": {
        $elemMatch: {
          "Nombre": "Música",
          "Calificación": { $lt: 9 }
        }
      }
    }
  ],
  {
    "Nombre": 1,
    "Apellidos": 1,
    "Asignaturas": {
      $elemMatch: {
        "Nombre": { $in: ["Historia", "Música"] }
      }
    }
  },
  "_id": 0
})
```

Dentro de la sección donde indicamos los valores a mostrar, especificamos que la asignatura debe ser Historia o Música si no, devolvería todas las asignaturas del alumno, aunque solo una de ellas cumpliera las condiciones.

Como se puede apreciar, hay diferentes formas de conseguir los mismos resultados ya que, unas veces usamos primero el operador \$or y otras veces \$elemMatch.

```
db.Alumnos.find({
  "Asignaturas": {
    $elemMatch: {
      $or: [
        {
          "Nombre": "Historia",
          "Calificación": { $gt: 6 }
        },
        {
          "Nombre": "Música",
          "Calificación": { $lt: 9 }
        }
      ]
    }
  }
},
{
  "Nombre": 1,
  "Apellidos": 1,
  "Asignaturas": {
    $elemMatch: {
      "Nombre": { $in: ["Historia", "Música"] }
    }
  },
  "_id": 0
})
```

Por último, si quisiéramos obtener los alumnos que hayan sacado, al mismo tiempo, más de un 6 en Historia y menos de un 9 en Música:

```
db.Alumnos.find({
  $and: [
    {
      "Asignaturas": {
        $elemMatch: {
          "Nombre": "Historia",
          "Calificación": { $gt: 6 }
        }
      }
    },
    {
      "Asignaturas": {
        $elemMatch: {
          "Nombre": "Música",
          "Calificación": { $lt: 9 }
        }
      }
    }
  ]
})
```

5.4.5 Agregaciones

Las agregaciones en MongoDB son una herramienta poderosa que permite realizar operaciones de procesamiento de datos más complejas y flexibles que las operaciones de consulta regulares. Las agregaciones son útiles cuando se necesita transformar, filtrar, agrupar y analizar datos de manera más avanzada que lo que se puede lograr con consultas simples. Algunas de las razones por las cuales las agregaciones son útiles son:

- **Agrupación y Resumen:** permite agrupar datos basados en ciertos campos y luego calcular estadísticas o resúmenes sobre esos grupos. Esto es útil para generar informes o presentar datos de manera más comprensible.
- **Combinación de Datos:** las agregaciones permiten combinar información de múltiples documentos o colecciones en una única salida. Esto es útil cuando necesitas consolidar datos de diferentes fuentes.
- **Transformación de Datos:** se puede modificar la estructura de los datos durante el proceso de agregación, renombrar campos, crear nuevos campos calculados y más.
- **Filtros y Proyecciones Personalizadas:** las etapas de agregación como `\$match` y `\$project` permiten filtrar y proyectar datos de manera más específica y personalizada.

La función **aggregate()** es la que nos permitirá hacer este tipo de cálculos, aunque también se puede usar para realizar consultas sencillas. Veamos un ejemplo:

```
db.Alumnos.aggregate([
  {
    $match: {
      edad: { $gt: 18 }
    }
  }
])
```

Este código realiza una agregación en la colección "Alumnos" y utiliza el comando **\$match** para filtrar los documentos donde la edad sea mayor que 18.

Si por ejemplo quisiéramos obtener los alumnos con alguna nota superior a 7:

```
db.Alumnos.aggregate([
  {
    $match: {
      "Asignaturas.Calificación": { $gte: 7 }
    }
  }
])
```

Para mostrar unos campos concretos dentro de la función **aggregate** debemos usar el comando **\$project** de la siguiente manera.

```
db.Alumnos.aggregate([
  {
    $match: {
      "Asignaturas.Calificación": { $gte: 7 }
    }
  },
  {
    $project: {
      "Nombre": 1,
      "Apellidos": 1,
      "Asignaturas": 1,
      "_id": 0
    }
  }
])
```

Si, además, quisiéramos ordenar el resultado de la consulta por el campo **Nombre** de manera ascendente:

```
db.Alumnos.aggregate([
  {
    $match: {
      "Asignaturas.Calificación": { $gte: 7 }
    }
  },
  {
    $project: {
      "Nombre": 1, "Apellidos": 1,
      "Asignaturas": 1, "_id": 0
    }
  },
  {
    $sort: {
      "Nombre": 1
    }
  }
])
```


Veamos ahora algunos ejemplos de funciones de agregación propiamente dichas. Por ejemplo, imaginemos que queremos contar cuántas notas hay iguales o superiores a 9.

```
db.Alumnos.aggregate([
  {
    $match: {
      "Asignaturas.Calificación": { $gte: 9 }
    }
  },
  {
    $count: "Alumnos_con_notas_alta"
  }
])
```

Lo primero que hacemos es, mediante `$match`, buscar las asignaturas con nota mayor o igual a 9 y, con `$count`, contamos dichas asignaturas y renombramos el campo.

Para el siguiente ejemplo, lo que haremos será sumar todas esas notas superiores o iguales a 9.

```
db.Alumnos.aggregate([
  {
    $unwind: "$Asignaturas"
  },
  {
    $match: {
      "Asignaturas.Calificación": { $gte: 9 }
    }
  },
  {
    $group: {
      _id: null,
      totalNotas: { $sum: "$Asignaturas.Calificación" }
    }
  }
])
```

\$unwind: "\$Asignaturas": este paso descompone los documentos de la colección "Alumnos" en múltiples documentos, uno por cada elemento en el array "Asignaturas". Esto es útil cuando se tiene un array dentro de un documento y se desea tratar cada elemento del mismo como un documento independiente en la agregación.

\$Asignaturas: el símbolo "\$" se utiliza para hacer referencia a campos en documentos. En el contexto de esta agregación, "\$Asignaturas" se utiliza para hacer referencia al campo "Asignaturas" dentro de los documentos de la colección "Alumnos".

El campo "Asignaturas" en este caso es un array, y al usar "\$Asignaturas" en la agregación, estamos indicando que deseamos operar en cada elemento individual dentro de ese array. El operador "\$unwind" se encarga de descomponer el array en documentos individuales, permitiendo que podamos realizar operaciones en cada asignatura de forma independiente.

\$match: es una etapa de filtrado que selecciona los documentos que cumplen ciertos criterios. En este caso, se filtran los documentos para encontrar aquellos donde la "Calificación" en la subcolección "Asignaturas" sea mayor o igual a 9.

\$group: esta etapa agrupa los documentos que han pasado las etapas anteriores y realiza cálculos en ellos. En este caso, se está creando un solo grupo con `_id` establecido como `null` (lo que significa que todos los documentos se agruparán juntos, o lo que es lo mismo, realmente no queremos agrupar por nada) y se calcula la suma total de las calificaciones de las asignaturas usando el operador `$sum`.

Para sumar todas las asignaturas sin el filtro de la calificación:

```
db.Alumnos.aggregate([
  {
    $unwind: "$Asignaturas"
  },
  {
    $group: {
      _id: null,
      SumaCalificaciones: { $sum: "$Asignaturas.Calificación" }
    }
  }
])
```

Para sumar todas las notas de la asignatura Matemáticas:

```
db.Alumnos.aggregate([
  {
    $unwind: "$Asignaturas"
  },
  {
    $match: {
      "Asignaturas.Nombre": "Matemáticas"
    }
  },
  {
    $group: {
      _id: null,
      SumaCalificacionesMat: { $sum: "$Asignaturas.Calificación" }
    }
  }
])
```

Para sumar estas mismas notas cuando la ciudad empiece por m o M:

```
db.Alumnos.aggregate([
  {
    $unwind: "$Asignaturas"
  },
  {
    $match: {
      "Asignaturas.Nombre": "Matemáticas",
      "Ciudad": /^m/i
    }
  },
  {
    $group: {
      _id: null,
      SumaCalifMat: { $sum: "$Asignaturas.Calificación" }
    }
  }
])
```

Para hacer este mismo ejemplo, pero usando el operador \$and en lugar de su forma implícita:

```
db.Alumnos.aggregate([
  {
    $unwind: "$Asignaturas"
  },
  {
    $match: {
      $and: [
        { "Asignaturas.Nombre": "Matemáticas" },
        { "Ciudad": /^m/i }
      ]
    }
  },
  {
    $group: {
      _id: null,
      SumaCalifMat: { $sum: "$Asignaturas.Calificación" }
    }
  }
])
```

Supongamos que ahora queremos obtener la nota más baja:

```
db.Alumnos.aggregate([
```

```
{
  $unwind: "$Asignaturas"
},
{
  $sort: {
    "Asignaturas.Calificación": 1
  }
},
{
  $limit: 1
},
{
  $project: {
    _id: 0,
    "Nombre": 1,
    "Apellidos": 1,
    "Asignaturas.Nombre": 1,
    "Asignaturas.Calificación": 1
  }
})
```

Lo que hacemos aquí es ordenar las calificaciones de manera ascendente y limitar los documentos devueltos a 1, usando para ello el operador **\$limit**. Esta consulta tiene un problema y es que, en el caso de haber más de una asignatura con la mínima nota, sólo devolverá una de ellas. Si queremos que devuelva todas las que haya, la cosa se complica un poco. El código para hacer esto sería el siguiente:

```
db.Alumnos.aggregate([
  {
    $unwind: "$Asignaturas"
  },
  {
    $group: {
      _id: null,
      lowestGrade: { $min: "$Asignaturas.Calificación" },
      alumnos: { $push: "$$ROOT" }
    }
  },
  {
    $unwind: "$alumnos"
  },
  {
    $match: {
      $expr: {
        $eq: ["$alumnos.Asignaturas.Calificación", "$lowestGrade"]
      }
    }
  },
  {
    $project: {
      _id: 0,
      "alumnos.Nombre": 1,
      "alumnos.Apellidos": 1,
      "alumnos.Asignaturas.Nombre": 1,
      "alumnos.Asignaturas.Calificación": 1
    }
  }
])
```

Descompongamos el código paso a paso:

1. **`\$unwind: "\$Asignaturas"`**: descompone los documentos en la colección "Alumnos" por cada asignatura en el array "Asignaturas". Esto crea múltiples documentos, uno por cada asignatura de cada alumno.
2. **`\$group`**: en esta etapa, todos los documentos descompuestos anteriormente son agrupados. Como **`_id: null`**, todos los documentos se agrupan en un solo grupo.
 - a. **lowestGrade: { \$min: "\$Asignaturas.Calificación" }**: Calcula la calificación mínima de todas las asignaturas en el grupo y la almacena en "lowestGrade".

- b. **alumnos: { \$push: "\$\$ROOT" }**: crea un array llamado "alumnos" que contiene todos los documentos originales del grupo (es decir, todos los documentos descompuestos en el paso 1). **\$\$ROOT** es una variable de sistema para las agregaciones de MongoDB que hace referencia al documento actual que está siendo procesado en la etapa actual de la agregación. Podemos usar **\$\$ROOT** para acceder a todo el contenido del documento en esa etapa.
3. **\$unwind: "\$alumnos"**: en esta fase, cada documento en el array "alumnos" se descompone nuevamente en documentos individuales. Esto es necesario para poder trabajar con cada documento por separado en las etapas siguientes.
4. **\$match**: filtra los documentos para incluir solo aquellos en los que la calificación de la asignatura del alumno actual sea igual a la calificación mínima calculada en el paso 2.
5. **\$project**: se proyectan solo los campos seleccionados en la salida final.

Veamos ahora cómo podríamos obtener las diferentes notas agrupadas por asignatura:

```
db.Alumnos.aggregate([
  {
    $unwind: "$Asignaturas"
  },
  {
    $group: {
      _id: "$Asignaturas.Nombre",
      calificaciones: { $push: "$Asignaturas.Calificación" }
    }
  }
])
```

Lo primero que hacemos es descomponer el array Asignaturas en documentos individuales, para después agrupar por el campo nombre especificado en `_id`. Para terminar, creamos un array llamado calificaciones donde, mediante `$push`, vamos añadiendo las diferentes notas.

Por último, vamos a ver cómo realizar lo más parecido a un JOIN que ofrece MongoDB. Para ello, debemos usar el operador **\$lookup**. Recordemos que la estructura de la colección Alumnos es la siguiente:

```
{
  "Nombre": "María",
  "Apellidos": "López Rodríguez",
  "edad": 17,
  "Ciudad": "Barcelona",
  "Dirección": "Avenida Central, 456",
  "dni": "23456789B",
  "Asignaturas": [
    { "Nombre": "Física", "Calificación": 7 },
    { "Nombre": "Literatura", "Calificación": 8 },
    { "Nombre": "Química", "Calificación": 6 }
  ]
}
```

Además, ahora también tenemos una colección Profesores con la siguiente estructura:

```
{
  Nombre: "Mariano",
  Apellido: "Pérez Sánchez",
  Edad: 35,
  "Asignaturas": [
    { "Nombre": "Matemáticas" },
    { "Nombre": "Música" }
  ]
}
```

Si queremos ver qué asignaturas han cursado los alumnos y qué profesor imparte esa asignatura, debemos usar el operador \$lookup de la siguiente manera:

```
db.Alumnos.aggregate([
  {
    $unwind: "$Asignaturas"
  },
  {
    $lookup: {
      from: "Profesores",
      localField: "Asignaturas.Nombre",
      foreignField: "Asignaturas.Nombre",
      as: "Profesor"
    }
  },
  {
    $unwind: "$Profesor"
  },
  {
    $project: {
      _id: 0,
      "Nombre Alumno": "$Nombre",
      "Nombre Asignatura": "$Asignaturas.Nombre",
      "Nombre Profesor": "$Profesor.Nombre"
    }
  }
])
```

from: especifica la colección desde la cual se tomarán los datos para combinar. En este caso, estamos buscando en la colección "Profesores" para obtener información relacionada con las asignaturas.

localField: especifica el campo en la colección actual (en este caso, "Alumnos") que se utilizará para la comparación. Estamos utilizando "Asignaturas.Nombre" para tomar los nombres de las asignaturas de los alumnos.

foreignField: especifica el campo en la colección "Profesores" que se utilizará para la comparación con el localField. Nuevamente, estamos utilizando "Asignaturas.Nombre" para tomar los nombres de las asignaturas de los profesores.

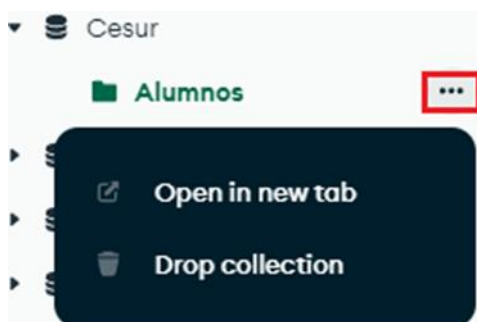
as: especifica el nombre del nuevo campo donde se almacenarán los resultados combinados. En este caso, estamos utilizando "Profesor" como nombre para este campo.

Es necesario recordar que el signo \$ se utiliza para acceder a los valores en el contexto del documento actual o de los resultados anteriores en la agregación. En el caso de **\$Nombre**, se refiere al valor del campo "Nombre" en cada documento de la colección "Alumnos" a medida que pasa por las diferentes etapas de la agregación.

Para terminar, es importante comentar que el comando **\$lookup** es el equivalente a INNER JOIN de SQL por lo que, si solo tenemos registrado a un profesor que da dos asignaturas, los alumnos que no cursen ninguna de esas asignaturas no aparecerán en el resultado de la búsqueda.

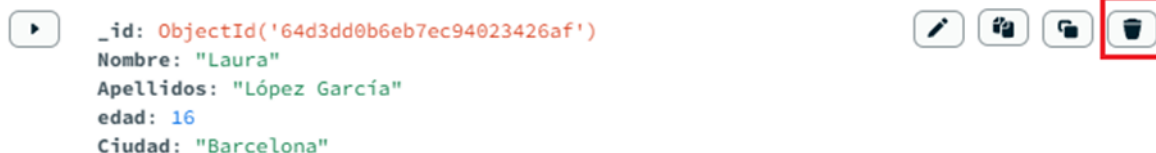
5.5 Eliminación de documento y colecciones

Eliminar documentos y colecciones mediante la interfaz gráfica es tremendamente sencillo, lo único que debemos hacer es pulsar la opción habilitada para ello. Para eliminar una colección, pulsamos sobre los tres puntos que aparecen a la derecha de su nombre y seleccionamos la opción Drop collection.



Eliminación de la colección Alumnos.

Para eliminar un documento de una colección, pasamos el ratón por encima del documento en cuestión y pulsamos el botón de eliminar.



Eliminación de documento

Para hacer esto mismo mediante código, al igual que para insertar y modificar, volvemos a tener dos funciones: **deleteOne()** y **deleteMany()**. Sobre deleteOne(), es importante recordar que en el caso de que más de un documento cumpla la condición, solo se borrará uno de ellos. Para eliminar al alumno con DNI 22222222J:

```
db.Alumnos.deleteOne({  
  "dni": "22222222J"  
})
```

Si, por ejemplo, quisiéramos eliminar a todos los alumnos que sean de Murcia y que sean mayores de 50 años:

```
db.Alumnos.deleteMany({  
  Ciudad: "Murcia",  
  Edad: { $gt: 50 }  
})
```

Para eliminar a los alumnos con la asignatura de inglés suspensa:

```
db.Alumnos.deleteOne({  
  "Asignaturas.Nombre": "Inglés",  
  "Asignaturas.Calificación": { $lt: 5 }  
})
```

Al igual que cuando usábamos **find()** o **aggregate()**, podemos jugar con las diferentes opciones y comandos. La misma consulta usando el operador **\$and** quedaría así:

```
db.Alumnos.deleteOne({  
  $and: [  
    { "Asignaturas.Nombre": "Inglés" },  
    { "Asignaturas.Calificación": { $lt: 5 } }  
  ]  
})
```

Eliminar una colección es tan sencillo como utilizar la siguiente línea de código:
`db.Colección.drop()` . Para este caso, bastaría con poner
`db.Alumnos.drop()` .

Si lo que queremos es eliminar la base de datos:

```
use MiBaseDeDatos  
db.dropDatabase()
```



VÍDEO DE INTERÉS

Aprende más sobre MongoDB siguiendo este tutorial:



RESUMEN FINAL

Las bases de datos NoSQL son una categoría de sistemas de almacenamiento de datos que difieren de las bases de datos relacionales tradicionales. Se caracterizan por su flexibilidad en el esquema, permitiendo la gestión de datos no estructurados o semiestructurados. Además, destacan por su escalabilidad horizontal, lo que significa que pueden manejar grandes volúmenes de datos distribuidos en varios nodos. Aunque ofrecen ventajas como rendimiento y alta disponibilidad, también tienen desventajas, como la complejidad en la consistencia de datos.

Existen varios tipos de bases de datos NoSQL, incluyendo bases de datos clave-valor, documentos, columnas y grafos. Cada tipo se adapta a diferentes casos de uso y escenarios. Algunos ejemplos de sistemas gestores de bases de datos NoSQL incluyen MongoDB, una base de datos de documentos; Redis, una base de datos en memoria; Apache Cassandra, que es altamente escalable; y Neo4j, una base de datos de grafos. Cada sistema tiene sus características y casos de uso específicos, lo que los hace útiles para distintas aplicaciones y necesidades. Por último, hemos visto cómo hacer operaciones básicas en MongoDB tanto de manera gráfica como por comandos.