

Bienvenido al curso **Aprende bases de datos NoSQL con ArangoDB**. Su objetivo es introducir al estudiante en el mundo de las bases de datos **NoSQL** orientadas a documento y clave-valor mediante **ArangoDB**. Al finalizar el curso, el estudiante conocerá estos tipos de bases de datos, conociendo su terminología, además de trabajar con **ArangoDB**.

La lección comienza introduciendo el concepto de sistema de bases de datos y sus principales arquitecturas de desarrollo. A continuación, se introduce al estudiante en el mundo **NoSQL**, describiendo qué es **NoSQL** y los tres modelos más utilizados hoy en día: los almacenes clave-valor, los almacenes de documentos y los almacenes de grafos. Todos ellos soportados nativamente por **ArangoDB**. Después se presenta las principales características de los sistemas de gestión de bases de datos **NoSQL**. Finalmente, se introduce **ArangoDB** y se proporciona información sobre el curso.

Al finalizar la lección, el estudiante sabrá:

- Qué es un sistema de gestión de bases de datos.
- Cuáles son las dos arquitecturas principales de desarrollo de sistemas de bases de datos: la biblioteca integrable y el motor cliente/servidor.
- Qué es **NoSQL**.
- Cuáles son las principales características de los motores **NoSQL**.
- Qué es una base de datos clave-valor y para qué se suele utilizar.
- Qué es una base de datos de documentos y para qué se suele utilizar.
- Qué es una base de datos de grafos y para qué se suele utilizar.
- Qué es una base de datos multimodelo.
- Qué modelos de bases de datos soporta **ArangoDB** nativamente.

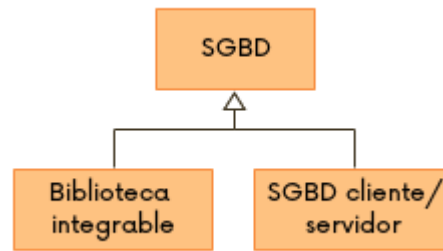
Introducción

Un **sistema de gestión de bases de datos** (*database management system*) o **SGBD** (*DBMS*) es un software que gestiona bases de datos. La idea que se esconde tras este tipo de productos es delegar en ellos la gestión, administración y el control de acceso a los datos, mediante un producto especializado en estas funciones. Esto mejora el rendimiento final de las aplicaciones y la integridad de los datos almacenados así como facilita el desarrollo de aplicaciones e incrementa la productividad de los equipos de desarrollo.

Las principales funciones de un SGBD son:

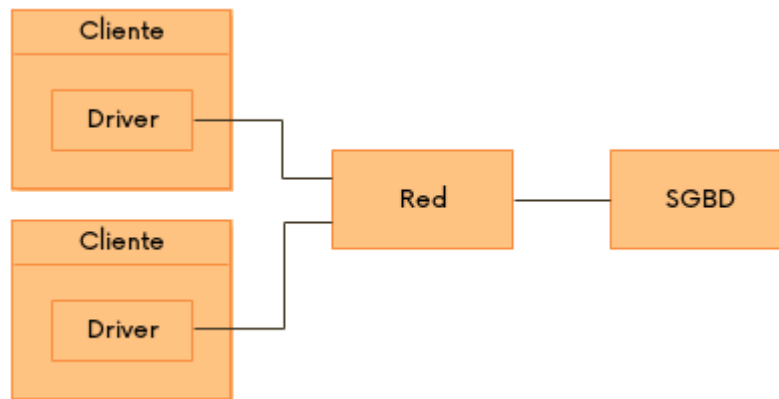
- Administrar adecuadamente el almacenamiento de los datos en los dispositivos de almacenamiento.
- Administrar adecuadamente la memoria utilizada por el SGBD para que no robe toda la memoria del sistema e impida la ejecución de otros programas en la misma máquina.
- Controlar o regular el acceso a los datos según el usuario que se encuentre detrás. No todos los usuarios tiene por qué poder acceder a todos los datos.

Atendiendo a su arquitectura, podemos distinguir básicamente dos tipos de motores de bases de datos, los que se implementan mediante una arquitectura cliente/servidor y las bibliotecas integrables.



SQLite y **PouchDB** son ejemplos de **bibliotecas integrables** (*embeddable libraries*), componentes de software que se pueden integrar dentro de las propias aplicaciones. El motor lo implementa un equipo especializado de ingenieros que proporcionan una biblioteca reutilizable que se integra muy fácilmente en las aplicaciones. Se ejecutan dentro del proceso de la propia aplicación. No son recomendables para proyectos medianos y grandes que requieren control de acceso desde varias aplicaciones y/o por varios usuarios. Son muy útiles.

En cambio, un **sistema cliente/servidor** (*client/server system*) es aquel que distingue entre aplicación **cliente** (*client*), aquella que desea acceder a los datos, y aplicación **servidora** (*server*), aquella que se encarga de administrar y controlar el acceso a los datos. Cada una de ellas se ejecuta en un proceso aparte e incluso, lo más frecuente, en máquinas distintas. La mayoría de SGBDs son de este segundo tipo como, por ejemplo, **ArangoDB**, **Cassandra**, **CouchDB**, **MariaDB**, **MongoDB**, **PostgreSQL**, **RethinkDB** y **SQL Server**.



En un modelo cliente/servidor, los datos pueden ser accedidos por distintas aplicaciones clientes, lo que no ocurre en el modelo de biblioteca integrable donde sólo la aplicación que integra el motor puede acceder a los datos.

Para acceder a los **SGBDs**, las aplicaciones utilizan **drivers**, componentes de software especializados para el acceso a servidores de base de datos. En el caso de una biblioteca integrable, el *driver* suele llevar consigo la propia biblioteca. En el caso de **SGBDs** cliente/servidor, los *drivers* accederán a los servidores de bases de datos mediante la red. A la hora de elegir un **SGBD** u otro, una de las cosas que hay que mirar es si dispone de *driver* para nuestra plataforma de desarrollo y, a ser posible, oficial del fabricante para asegurarnos de que se actualiza a un ritmo razonable con respecto a la evolución del producto.

Por otra parte, principalmente, existe dos tipos de sistemas de bases de datos, los relacionales o **SQL** y los **NoSQL**. El objeto del presente curso es el sistema de gestión de bases de datos **ArangoDB** con el que aprender sobre bases de datos **NoSQL**.

NoSQL es un tipo o familia de sistemas de gestión de bases de datos que *no* sigue los principios de los sistemas relacionales o **SQL**. El término **NoSQL** tiene básicamente dos acepciones: no **SQL** o *Not Only SQL* (no sólo **SQL**). La primera hace hincapié en que *no* usa los principios relacionales, mientras que la segunda indica que **SQL** *no* es la única tecnología de bases de datos.

Los sistemas de gestión de bases de datos **NoSQL** no tienen como objeto, en ningún caso, sustituir a los sistemas de gestión **SQL**. Pero sí proporcionar una herramienta adicional o alternativa que permita atender mejor aquellas situaciones para las que las bases de datos **SQL** no son óptimas o presentan un mal rendimiento. Diferentes tipos de aplicación requiere diferentes tipos de bases de datos. Es

inevitable pues que los especialistas de bases de datos conozcan y usen distintos SGBD, tanto SQL como, por ejemplo, MariaDB, PostgreSQL o SQL Server, así como NoSQL como ArangoDB, Cassandra, CouchDB, MongoDB o RethinkDB.

Bases de datos NoSQL

Aproximadamente, en 2005 comenzó una nueva revolución de bases de datos, orientada a NoSQL, con un objetivo principal: llenar ese vacío donde SQL proporciona un mal rendimiento debido a su forma de trabajar. Son muchos los productos NoSQL que han aparecido desde entonces. Los principales son ArangoDB, Cassandra, CouchDB, InfluxDB, MongoDB y RethinkDB. Cada uno de ellos tiene como objeto llenar uno o más huecos de SQL.

Una **base de datos** (*database*) es una colección estructurada de datos y/u objetos, organizada atendiendo a la especificación de un determinado modelo de base de datos. Ya sabemos que existe diversos tipos de bases de datos como, por ejemplo, las relacionales, las orientadas a documentos, las de clave-valor o las de grafos.

Como vemos, existe varios modelos de datos, donde un **modelo de datos** (*data model*) lo que hace es describir una manera de trabajar. Una forma de organizar, almacenar y acceder a los datos. En estos momentos, el más conocido y usado en todo el mundo es el modelo relacional implementado por los motores SQL. Otros menos conocidos pero que van abriéndose paso rápidamente son algunos NoSQL como, por ejemplo, el de documentos, el de clave-valor y el de grafos.

Algunos sistemas de gestión de bases de datos soportan sólo un único modelo, mientras que otros soportan varios. Estos últimos se conocen formalmente como **bases de datos multimodelo** (*multimodel databases*). La idea que se esconde detrás de un sistema multimodelo es muy sencilla: combinar varios modelos de datos en un único producto, usando el mismo lenguaje de consulta para todos ellos.

Los sistemas NoSQL se utilizan ampliamente y cada día más. Organizaciones de cualquier tamaño e índole como, por ejemplo, Adobe, Amazon, AOL, Barclays, Canonical, Cisco, Disney, EA, eBay, Ericsson, Facebook, Forbes, Google, HP, IBM, LinkedIn, McDonald's, Microsoft, MTV, NASA, RedHat, Telefónica, The Guardian, The New York Times y Twitter están utilizando sistemas NoSQL. Especialmente las *startups* son las que las utilizan con fervor para desarrollar sus productos más rápidamente.

Almacenes clave-valor

Un **almacén clave-valor** (*key-value store*) es un tipo de sistema NoSQL que almacena datos en forma de tabla o *array* asociativo. Los datos se agrupan en pares, donde un **par** (*pair*) no es más que un objeto o registro de datos formado por dos elementos: una clave y un valor. La **clave** (*key*) representa el identificador a utilizar para acceder al objeto o registro. Y el **valor** (*value*) contiene los datos.



La clave suele ser de tipo texto, pero el valor puede ser de cualquier otro tipo como, por ejemplo, un número, un objeto, un *array* u otro texto.

Este tipo de sistemas se diseñan básicamente con dos objetivos: sencillez y rendimiento. Como son el tipo más simple de base de datos, son fáciles de implementar, no presentan grandes complejidades. Por otra parte, se diseñan buscando extraer el máximo rendimiento del sistema de bases de datos, dentro de su margen de maniobra.

Pero al igual que cualquier otro tipo sistema de bases de datos, los almacenes clave-valor no se pueden utilizar en cualquier proyecto. Sus principales usos son:

- El cacheo de datos.
- El almacenamiento de información de sesión.
- El almacenamiento de datos de *reporting*.
- El almacenamiento de datos temporales.

- El almacenamiento de catálogos de productos.
- El almacenamiento de registros de mensajes o eventos.

Entre los principales productos de este tipo encontramos [Amazon SimpleDB](#), [ArangoDB](#), [Redis](#) y [Riak KV](#).

Algunas compañías suelen usar almacenes clave-valor como caché poniéndola delante de otros motores. En estos casos, el almacén principal de datos es el otro motor, independientemente de su tipo. Pero si la aplicación requiere mucho procesamiento de datos, lecturas y escrituras intensivas, se suele utilizar almacenes clave-valor. Estos motores suelen tener mejor rendimiento que otros en L/E. Cuando se ha terminado el procesamiento intensivo, entonces se cogen los datos y se vuelcan al almacén principal. Por ejemplo, en un portal web con mucho tráfico, la información de sesión, la publicidad, las páginas webs consultadas, etc. se puede almacenar en un almacén clave-valor. Una vez se cierra la sesión, se coge la información recopilada y se vuelca al sistema principal de almacenamiento sea [SQL](#) u otro [NoSQL](#).

Almacén de documentos

Otro de los tipos de bases de datos [NoSQL](#) más utilizado es el [almacén de documentos](#) (*document store*), que tiene como objeto almacenar objetos de datos, tanto estructurados como desestructurados, conocidos formalmente como [documentos](#) (*documents*). A diferencia de los almacenes clave-valor donde los datos se acceden generalmente mediante la clave, en los almacenes de documentos, las cosas no son iguales, y se puede acceder a un documento a partir de cualquiera de sus campos.

Una de las principales ventajas de este tipo de almacenes es que pueden contener en un único documento toda su información. Lo que facilita su acceso. A diferencia de las bases de datos relacionales donde para obtener toda la información hay que utilizar [JOINS](#), operaciones de reunión de datos procedentes de distintas tablas.

El desarrollo de aplicaciones con bases de datos orientadas a documentos es más fácil y rápido que con una base de datos relacional. Pero obviamente, sacrifica algunos aspectos para conseguirlo como, por ejemplo, la integridad de los datos.

Sus principales usos son:

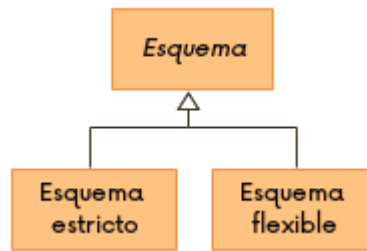
- El cacheo de datos.
- El almacenamiento de información de sesión.
- La consolidación de datos mediante puntos de vista particulares.
- El análisis de datos a partir de datos consolidados.
- El almacenamiento de datos de *reporting*.
- El almacenamiento de catálogos de productos.
- El almacenamiento de registros de mensajes o eventos.
- El almacenamiento de datos personalizables por los usuarios.
- La gestión y la publicación de contenido.
- El almacenamiento de datos de sitios webs.

Entre los principales productos de este tipo encontramos [ArangoDB](#), [CouchDB](#), [DocumentDB](#), [MongoDB](#) y [RethinkDB](#).

Esquema de documento

Un [esquema](#) (*schema*) define el conjunto de campos que debe presentar un documento. Los documentos se almacenan dentro de colecciones, las cuales representan contenedores donde almacenar documentos.

Conceptualmente, en bases de datos, independientemente de si son [SQL](#) o [NoSQL](#), los esquemas se clasifican principalmente en estrictos y flexibles.



Un **esquema estricto** (*strict schema*), también conocido como **esquema estático** (*static schema*), es aquel que define exactamente qué campos debe presentar cada documento de la colección, así como los tipos de los valores. Todo documento debe cumplir necesariamente el esquema de su colección. No se puede añadir un documento a una colección si no cumple con el esquema definido en la colección en la que se almacena. Es el tipo de esquema que predomina en motores **SQL** y algunos **NoSQL** como, por ejemplo, **Cassandra**, **PostgreSQL**, **SQL Server** y **SQLite**.

En cambio, un **esquema flexible** (*flexible schema*), también conocido como **esquema dinámico** (*dynamic schema*) o **sin esquema** (*schemaless*), es más flexible y no define ningún tipo de restricción en los campos que debe presentar el documento. Esto quiere decir que en una colección se podrá almacenar documentos con esquemas distintos y tipos de datos distintos para campos homónimos. Aunque generalmente, por convenio, se suele seguir un esquema muy similar. Se utiliza en sistemas de gestión de bases de datos como, por ejemplo, **ArangoDB**, **CouchDB**, **MongoDB** y **RethinkDB**.

En un esquema estricto, todos los documentos presentan los mismos campos y sus valores, a su vez, deben tener los mismos tipos. En cambio, en un esquema flexible, esto no es necesario y cada documento puede tener su propio esquema, o sea, puede disponer de cualquier número de campos con valores de cualquier tipo. Cuando se utiliza un esquema dinámico, es obligación de las aplicaciones asegurar que los datos insertados cumplen los esquemas *pactados* si el motor no proporciona algún mecanismo nativo para hacerlo.

Los esquemas flexibles ayudan enormemente al desarrollo rápido de aplicaciones, sobre todo en el desarrollo de aplicaciones webs, pues aumenta la productividad. Una de las principales razones por las que se utiliza bases de datos de documentos y clave-valor. En **SQL**, si añadimos una nueva columna a una tabla, hay que garantizar que todas las filas ya existentes cumplen con las restricciones de la nueva columna. Esto es sencillo de ver. ¿Qué ocurre si añadimos una nueva columna a la tabla y ésta es obligatoria, es decir, toda fila debe proporcionar un valor para la columna? Si la tabla ya tiene filas, éstas no tendrán datos para esa nueva columna, por lo que habrá que hacer un trabajo extra: primero, declarar la columna como opcional; después, ejecutar una consulta que añada el valor para esa nueva columna en todas las filas existentes; y finalmente, definir la columna como obligatoria. Como puede observar, es sencillo pero algo tedioso. En los esquemas flexibles, esto no es necesario.

Algunos fabricantes de almacenes **NoSQL** consideran el uso de los esquemas flexibles como una ventaja. Esto lo es, pero a su vez no lo es. Por un lado, lo es, porque se desarrolla más rápidamente al no tener que definir los esquemas de las colecciones. Pero no lo es porque el esquema flexible hace que los datos ocupen más espacio en el dispositivo de almacenamiento y en memoria.

Debe quedar claro que como las colecciones no tienen asociado ningún esquema, los documentos que pueden almacenar pueden presentar cualquier esquema. Esto hace que la instancia deba almacenar, para cada documento, tanto su esquema como los valores de los campos. Esto hará que si una colección contiene un millón de documentos, todos ellos con el mismo esquema, digamos por ejemplo de cinco campos, como la colección no fija ningún esquema, cada documento almacenará el nombre de los cinco campos con sus respectivos valores y tipos, ¡para el millón de documentos! En sistemas con esquemas estáticos, habría un millón de documentos, se fijaría el orden en que cada campo se debe almacenar, de tal manera que el primer valor pertenecería siempre al primer campo; el segundo al segundo; y así sucesivamente. Y por otra parte, cada campo contendrá siempre un valor de un tipo determinado. Pero no se almacenaría un millón de veces el nombre de cada campo. Así pues, cuanto mayor sea el nombre, más espacio será requerido en memoria y en disco. A su vez, cuanto más espacio ocupa un documento en disco, más operaciones de E/S será necesario realizar; siendo las operaciones de E/S el principal cuello de botella de las bases de datos.

En resumen, la utilización de sistemas de gestión de bases de datos **NoSQL** tiene ventajas y desventajas. Cuando se utilizan, se entra al *juego*, se acepta las cosas como son, con lo bueno y lo malo. Al igual que

cuando se utiliza un sistema relacional.

Almacenes de grafos

Un **almacén de grafos** (*graph store*) permite el almacenamiento de datos relacionados en forma de grafos. Se basan en la teoría matemática de grafos, la cual tiene como objeto representar datos mediante puntos y líneas que se utilizan para resolver problemas de análisis.

Las principales áreas en las que se usa este tipo de bases de datos son:

- La medicina.
- El transporte y la logística.
- Las redes sociales.
- La seguridad como, por ejemplo, detección de fraudes.
- La gestión de tráfico.

Básicamente, lo que se busca con este tipo de bases de datos es facilitar el análisis de los datos representados mediante grafos para extraer hechos o datos que nos ayuden a tomar decisiones.

Entre los principales productos que soportan este tipo de bases de datos encontramos **ArangoDB**, **Neo4j**, **OrientDB** y **Titan**.

Características de las bases de datos NoSQL

Por lo general, los sistemas de gestión de bases de datos **NoSQL** presentan las siguientes características:

- Modelo de transacciones **ACID**, **AID** o **BASE**.
- Alta disponibilidad.
- Alta escalabilidad.
- Facilidad de uso.
- Modelos de datos desnormalizados.
- Precio.

Modelo de transacciones ACID, AID o BASE

Algunos sistemas **NoSQL** suelen usar el modelo de transacciones **ACID**, soportado íntegramente por los sistemas **SQL**. Y otros sólo **AID** o **BASE**. **ACID** es el acrónimo de *Atomic* (atómico), *Consistent* (consistente), *Isolated* (aislado) y *Durable*.

Atomicidad

Con la **atomicidad** (*atomicity*) lo que se busca es que los cambios realizados por toda transacción se confirmen. En caso de que un sólo cambio no se confirme, entonces todos los cambios realizados por la transacción se deshacen dejando el sistema como estaba antes del inicio de la propia transacción. En el caso de **SQL**, las transacciones además suelen estar formadas por un conjunto de una, dos o más operaciones **DML**. En algunos sistemas de gestión de bases de datos **NoSQL**, las transacciones no suelen ser multioperación, es más, si una operación modifica varios registros, algunas bases de datos **NoSQL** suelen ser transaccional sólo a nivel de un único registro, lo que significa que garantizan que todo el registro se actualiza y si no se consigue, no realiza la operación; pero si la operación actualiza varios registros, si en alguno de ellos falla, no deshacen las modificaciones previas, sólo garantizan que la que ha fallado no se realizará.

Como acabamos de mencionar, la transaccionalidad a nivel de registro es casi un estándar de facto en los sistemas **NoSQL**, aunque por suerte cada día son más las que soportan el modelo **ACID** como, por ejemplo, **ArangoDB**. Debido a que no tienen que preocuparse por un conjunto de operaciones ni por un conjunto de registros, los sistemas de gestión de bases de datos **NoSQL** presentan mejor rendimiento que las **SQL** porque se quitan un peso de encima. Obviamente, si la aplicación usuaria requiere transacciones multioperación o multiregistro, es muy probable que se dese utilizar una base de datos

que soporte **ACID**, porque si no se hace así, recaerá en la propia aplicación garantizar la atomicidad a nivel de operación completa.

Por lo general, las bases de datos **NoSQL** no suelen ser adecuadas para sistemas críticos u OLTP.

Consistencia

Cuando se habla de **consistencia** (*consistency*), de lo que se habla es de que una transacción no puede dejar una base de datos en un estado inconsistente. En las bases de datos **SQL**, la consistencia se consigue mediante la utilización de restricciones de integridad, principalmente de integridad referencial o de dominio. Muchos sistemas **NoSQL** no implementan la integridad, haciendo a las aplicaciones responsables de mantener esta integridad de datos. Así, por ejemplo, si en **SQL** tenemos dos tablas padre-hijo, si se define una restricción de integridad referencial en la que cuando se elimine la fila padre, automáticamente las hijas se supriman también, esto hará que nunca haya un hijo referenciando a un padre inexistente, manteniéndose así una integridad referencial y una consistencia de datos. Recordemos que para relacionar los padres con sus hijos se utilizan las reuniones (**JOINS**).

En **NoSQL**, generalmente no es así, no se puede definir la integridad referencial en la base de datos. Así pues, si un registro referencia a otro, si la aplicación no se encarga de mantener la integridad referencial, habrá inconsistencia en la base de datos. Esto ayuda a los sistemas de gestión de bases de datos **NoSQL** a tener que preocuparse por menos cosas cuando realizan las operaciones de modificación. Esta descarga de responsabilidades hace que estos sistemas sean más rápidos y presenten mejores rendimientos, pero a cambio obligan a las aplicaciones a preocuparse por estas cosas.

Algunos fabricantes de sistemas de gestión de bases de datos hablan de consistencia relajada, pero generalmente no hay consistencia, ni siquiera relajada. Parece más un término de marketing que una realidad.

Aislamiento

Con el **aislamiento** (*isolation*), se está hablando de que los cambios realizados por una transacción no pueden interferir con los realizados por otra o, en otras palabras, un cambio realizado por una transacción no puede ser visto por otra hasta que la primera confirme que ha acabado y haya acabado con éxito. Tal como hemos comentado con la propiedad de atomicidad, las transacciones de las bases de datos **NoSQL** son a nivel de registro, asegurando que cualquier cambio sobre un registro no sea visible hasta que el cambio en el registro haya finalizado.

Durabilidad

Finalmente, la propiedad de **durabilidad** (*durability*) hace referencia a que cuando una transacción ha finalizado con éxito, sus datos son permanentes aún en caso de que se produzca un fallo en el sistema de gestión de bases de datos. Esto se consigue mediante el uso de registro de transacciones. Algunos sistemas **NoSQL** lo implementan pero con una variante que puede llevar a pérdida de datos, algo que también es posible en algunos sistemas **SQL**.

Alto rendimiento

Los sistemas **NoSQL** suelen presentar mejores rendimientos que los sistemas **SQL**. Como se ha presentado en el anterior apartado, los sistemas **NoSQL** se descargan de ciertas responsabilidades que tienen las **SQL**, por lo tanto, si no tienen que preocuparse de esas cosas, no tendrán que hacer ciertas comprobaciones internas y, por ende, tendrán mejores tiempos de respuesta.

Además, los sistemas **NoSQL** se diseñan teniendo muy claro que tienen que mejorar enormemente la E/S a disco, uno de los principales cuellos de botella que tiene cualquier sistema de gestión de bases de datos, independientemente de su tipo.

Generalmente, las bases de datos **NoSQL** permiten almacenar toda la información de registros relacionados juntos. Esto hará que cuando se tenga que extraer toda la información, se pueda hacer mediante una única operación de lectura, algo muy importante y que generalmente en los sistemas **SQL** requiere varias operaciones.

Por ejemplo, supongamos una orden de pedido. En una base de datos **SQL**, tendremos, por un lado, la tabla padre Pedido que contiene los datos generales del pedido como, por ejemplo, la fecha, el

cliente, la dirección de entrega, etc. Por otro lado, una tabla aparte contendrá las líneas del pedido, cada una de las cuales contendrá la información de un artículo: su número de artículo, el precio del artículo y las unidades solicitadas. Esto significa que cuando queremos obtener la información completa del pedido, hay que juntar ambas tablas mediante una operación de reunión (**JOIN**). Además, las filas de un mismo pedido no tienen porque estar adyacentes, pueden estar repartidas por todo el disco; esto reduce todavía más el rendimiento de la base de datos. En cambio, por lo general, los sistemas **NoSQL** permiten almacenar toda esa información junta, lo que hará mucho más rápido su acceso que un sistema **SQL**, al no tener que realizar reuniones (**JOINS**) para obtenerla.

Los sistemas de gestión de bases de datos **NoSQL** suelen encontrarse optimizados para operaciones de lectura e inserción, o sea, abundan las lecturas de datos y las inserciones, no así las actualizaciones o supresiones. Esto no quiere decir que no sean óptimas para la actualización o supresión de registros de la base de datos, pero sí serán menos óptimas que esas mismas operaciones en una base de datos relacional. A cambio, si la base de datos presenta principalmente lecturas e inserciones, como ocurre en los *data warehouses* y en muchas aplicaciones webs, son mucho más óptimas que las relacionales.

Alta disponibilidad

La **alta disponibilidad** (*high availability*) hace referencia a que las bases de datos tienen que estar trabajando todo el tiempo posible, llegando incluso a no poder dejar de dar servicio bajo ningún concepto. En muchos casos, la parada de una base de datos durante horario de oficina puede conllevar la pérdida de cientos e incluso miles de euros por cada hora de inactividad.

Los sistemas **NoSQL** suelen presentar mejor disponibilidad que las **SQL**, aunque eso no significa que las **SQL** no presenten esta propiedad. Así, por ejemplo, sistemas de gestión de bases de datos como **ArangoDB** y **Cassandra** son sistemas distribuidos **NoSQL** que no presentan ningún punto de fallo. Si un nodo cae, cualquiera de los demás puede atender su carga sin pérdida de servicio, aunque claro está sí habrá peor rendimiento porque ahora hay un miembro menos para atender la carga de los usuarios. Otros sistemas como **MongoDB**, sí presentan punto de fallo, aunque ayudan a levantar el entorno en caso de fallo. Pero hoy en día, casi todos los sistemas **SQL** que se precien como, por ejemplo, **PostgreSQL** y **SQL Server** proporcionan también una muy buena alta disponibilidad.

Alta escalabilidad

Cuando se habla de **alta escalabilidad** (*high scalability*), se habla de lo fácil que es ampliar un entorno, añadiendo principalmente más recursos. Generalmente, los motores **NoSQL** permiten escalabilidad de manera muy sencilla. Se puede repartir muy fácilmente los datos entre varias instancias. Además de ser muy fácil la añadidura o la supresión de una instancia asociada a un clúster. Pero también, es fácil hacerlo en sistemas **SQL** como **PostgreSQL** y **SQL Server**.

Facilidad de uso

Los sistemas **NoSQL** son muy fáciles de usar y administrar. Se diseñan teniéndolo en mente. Generalmente, son más fáciles de usar y administrar que los sistemas **SQL**, pero huelga decir que porque son, digamos sutilmente, mucho más pequeños y potentes.

Modelos de datos desnormalizados

Por lo general, cuando se usa sistemas **NoSQL**, se utiliza modelos de datos desnormalizados, a diferencia de los sistemas **SQL** donde se utiliza el modelo normalizado. Esto significa que en muchas ocasiones una base de datos **NoSQL** tiene datos duplicados para, así, mejorar su acceso.

Precio

El precio de instalación y uso de un sistema de gestión de bases de datos **NoSQL** es generalmente bastante más barato que uno **SQL**. En el caso de **SQL**, existe sistemas muy buenos y gratuitos como, por ejemplo, **PostgreSQL**. En el mundo **NoSQL**, también encontramos versiones gratuitas de **ArangoDB**, **Cassandra** y **MongoDB**. Otros fabricantes como **Oracle** y **Microsoft** disponen de sus propios sistemas de gestión de bases de datos, con precios bastante elevados, principalmente **Oracle**.

De cara a grandes empresas cuyas paradas debido a fallos puede conllevar unos costes elevados, existe versiones de pago con soporte del fabricante. Por ejemplo, **DataStax** proporciona una versión de

pago y más potente de [Cassandra](#); [MongoDB Inc.](#) hace lo mismo para [MongoDB](#); [ArangoDB GmbH](#) para [ArangoDB](#); al igual que [EnterpriseDB](#) o [VMware](#) hacen con [PostgreSQL](#).

ArangoDB

[ArangoDB](#), el objeto del curso, es un sistema de gestión de base de datos [NoSQL](#) multimodelo, soporta o combina varios modelos de datos en uno. Todo ello desde un único producto, con el mismo lenguaje de consulta para todos los tipos. Su desarrollo está dirigido por la empresa alemana [ArangoDB GmbH](#). Y su página oficial es [arangodb.com](#).

Está escrito en [C++](#) y [JavaScript](#).

La usan compañías de distintos sectores y tamaños como por, ejemplo, [Craneware](#), [Douglas](#), [FlighStats](#), [Institute for Research in Immunology and Cancer](#), [Giant Swarm](#), [Karlsruhe Institute of Technology](#), [Liaison](#), [Noho Care](#) y [SEOlytics](#). Es muy recomendada para *startups* que necesitan arrancar su producto rápidamente, principalmente, por su facilidad de uso, su soporte multimodelo, su lenguaje de consulta y su extensibilidad mediante [JavaScript](#).

Principales características

Las principales características de [ArangoDB](#) son las siguientes:

- Dispone de *drivers* para las principales plataformas de desarrollo como, por ejemplo, [Java](#), [JavaScript](#), [.Net](#), [Node.js](#), [PHP](#) y [Python](#).
- Es multiplataforma, disponiéndose de ejecutables para [Docker](#), [Linux](#), [OS X](#) y [Windows](#), entre otras.
- Es multimodelo, soportando bases de datos de documentos, de clave/valor y de grafos.
- Utiliza esquemas dinámicos, lo que favorece el desarrollo rápido de aplicaciones.
- Permite el uso de datos desnormalizados y con esquemas flexibles, lo que facilita el desarrollo rápido de aplicaciones.
- Dispone de un lenguaje de consulta muy potente y fácil de aprender, [AQL](#).
- Permite alta disponibilidad.
- Es fácilmente escalable mediante replicación y particionamiento.
- Presenta un buen rendimiento.
- Es extensible mediante [JavaScript](#) y su *framework* integrado [ArangoDB Foxx](#).
- Es de código abierto.
- Dispone de dos ediciones de producto: una gratuita y otra de pago.
- Soporta transacciones [ACID](#).
- Soporta colecciones volátiles, o sea, en memoria. Lo que favorece el acceso a su contenido.
- Permite la reunión de datos mediante operaciones similares a los [JOINS](#) de [SQL](#).
- Soporta búsqueda de texto (*full-text search*).

Información del curso

Este curso está dedicado al sistema de gestión de bases de datos [ArangoDB](#).

Tiene como objetivo introducir al estudiante en el mundo de las bases de datos [NoSQL](#), mediante una descripción detallada de este motor de bases de datos por su soporte de los modelos clave-valor y de documentos. Dejándose el modelo de grafos, también soportado por [ArangoDB](#), para un curso aparte.

Al finalizarlo, el estudiante sabrá:

- Qué es [NoSQL](#).
- Qué es un modelo de datos.

- Qué es un sistema de gestión de bases de datos.
- Qué es un almacén clave-valor, para qué se usan y cómo se implementan en [ArangoDB](#).
- Qué es un almacén de documentos, para qué se usan y cómo se implementan en [ArangoDB](#).
- Cómo se organizan los datos en bases de datos multimodelo en [ArangoDB](#).
- Cómo realizar consultas de extracción, inserción, actualización y supresión mediante la API simple de [ArangoDB](#) y el lenguaje de consulta [AQL](#).
- Cómo ejecutar transacciones en [ArangoDB](#).
- Cómo usar el *shell* [arangosh](#) y la interfaz web de [ArangoDB](#).

Conocimientos previos

El estudiante debe tener conocimientos de [JavaScript](#).

Plan del curso

El curso tiene una duración aproximada de **7 horas**. Se divide en **18 lecciones**, cada una de ellas con una parte de teoría. Al finalizar el curso, puede realizar **un examen** de tipo test con el que evaluar los conocimientos adquiridos.

A continuación, se enumera las distintas lecciones y el tiempo estimado para su estudio:

Lección	Teoría	Descripción
1 Introducción	30min	Esta lección.
2 Instancias de bases de datos	10min	Describe qué es una instancia de bases de datos y cómo configurarla, arrancarla y detenerla.
3 Shell e interfaz web de ArangoDB	10min	Describe las principales herramientas clientes con las que acceder a las instancias de ArangoDB .
4 Autenticación de usuarios	20min	Describe el sistema de autenticación usado por ArangoDB para identificar a los usuarios que pueden conectarse a la instancia.
5 Bases de datos	20min	Describe, por un lado, qué es una base de datos y, por otro lado, cómo crearlas, suprimirlas y acceder a ellas, así como el control de acceso.
6 Colecciones de documentos	25min	Describe qué es una colección, qué tipos hay y cómo crear colecciones de documentos.
7 Inserción de documentos	15min	Describe cómo insertar nuevos documentos en colecciones de documentos. También introduce la API simple y el lenguaje de consulta AQL .
8 Selección simple de documentos	25min	Describe cómo extraer documentos mediante la API simple y el lenguaje AQL .
9 Operadores AQL	15min	Describe las expresiones y los operadores disponibles en AQL .
10 Actualización de documentos	20min	Describe cómo actualizar documentos de una colección mediante la API simple y AQL .
11 Supresión de documentos	10min	Describe cómo suprimir documentos de una colección mediante la API simple y AQL .
12 Funciones AQL predefinidas	35min	Presenta las funciones predefinidas de AQL .
13 Funciones AQL de usuario	10min	Describe cómo añadir nuevas funciones AQL a una base de datos.
14 Selección multcolección	15min	Describe cómo realizar consultas de extracción con documentos de varias colecciones.

15	Introducción a los índices	30min	Describe agnósticamente los índices, sus usos y otros aspectos importantes.
16	Índices en ArangoDB	15min	Describe el soporte de índices de ArangoDB.
17	Transacciones	15min	Describe qué es una transacción ACID y su soporte en ArangoDB.
18	Búsqueda de texto	5min	Describe en qué consiste la búsqueda de texto (<i>full-text search</i>) y su soporte en ArangoDB.
	Examen	60min	Evaluación de los conocimientos adquiridos.

Información de publicación

Título Aprende bases de datos NoSQL con ArangoDB

Autor Raúl G. González - raulggonzalez@nodemy.com

Primera edición Enero de 2017

Versión actual 1.0.0

Versión de ArangoDB 3.1

Contacto hola@nodemy.com