Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN SISTEMAS DE INFORMACIÓN

Trabajo Fin de Grado

Curso de Introducción a Big Data y Gestión de Almacenamiento con Bases de Datos no relacionales

Autor: Carlos Espejo Martínez

Tutor/es: Iván González Diego

2019

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN SISTEMAS DE INFORMACIÓN

Trabajo Fin de Grado

Curso de Introducción a Big Data y Gestión de Almacenamiento con Bases de Datos no relacionales

Autor: Carlos Espejo Martínez

Tutor/es: Iván González Diego

TRIBUNAL:

Presidente:

Vocal 1º:

Vocal 2º:

FECHA:

ÍNDICE DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

RESUMEN

Las Bases de Datos no relacionales o *NoSQL* son una respuesta a la necesidad de almacenar y acceder a datos que se almacenan en cantidades (cada vez más) masivas. Estas tecnologías hacen uso de estructuras menos convencionales y del almacenamiento de datos en clústeres de máquinas.

No obstante, estas nuevas tecnologías tienen sus inconvenientes, que hacen que sea necesario cumplir una serie de características para poder aprovechar sus ventajas.

En este trabajo, se elaborará el material necesario para aportar las bases de conocimiento que permitan realizar un análisis de los datos que ayude a tomar decisiones de diseño para aprovechar al máximo las características de estas tecnologías.

El objetivo final de la elaboración de este material es su configuración en forma de curso, orientado a alumnos de FP, donde se introducirán los fundamentos teóricos de las Bases de Datos *NOSQL* acompañados de casos prácticos en forma de pequeñas prácticas.

PALABRAS CLAVE

Bases de datos, Big Data, *NoSQL*, *SQL*, Nuevas Tecnologías, formación.

ABSTRACT

Non-relational databases or NoSQL are the answer to the necessity of storage and manage of increasing volumes of data. These technologies make use of non-conventional data structures, as well as clusters of machines to distribute the workload when managing these databases.

However, these new technologies have shortcomings, that require certain characteristics in the kind of data to storage to be able to provide an advantage over conventional databases.

In this project, we'll develop the required material to obtain basic knowledge of these technologies in order to be able to analyze and make design choices that maximize the advantages of using this kind of databases.

The main purpose of developing this material is for it to be configured in a course like form, aimed towards "FP" students, where the theoretical foundation of NOSQL databases will be introduced along side real world implementations in form of small exercises.

KEYWORDS

Databases, Big data, NoSQL, SQL, emerging technologies, learning.

INTRODUCCIÓN

Este Trabajo de Fin de Grado está orientado a la creación de un curso de formación en Big Data y gestión de Bases de Datos *NoSQL*. En concreto, se centra en la impartición de estos conocimientos a alumnos de titulaciones de Formación Profesional y tendrá una carga de trabajo para el estudiante de 150 horas.

Los estudiantes de Formación Profesional (FP), según el último Informe Del Mercado de Trabajo de los Jóvenes, publicado por el Servicio Público de Empleo Estatal (SEPE) en 2018, se incorporan con mayor facilidad al mercado de trabajo. En 2017, los contratos de alumnos de FP en su primer año de egresados ascendieron a 16'12% frente al 10'43% que representaron los titulados universitarios. Por ejemplo, la contratación de estudiantes de la FP "Técnico en Sistemas Microinformáticos y Redes" ha aumentado un 17'31% entre 2016 y 2017 según estos últimos datos.

Esto no quiere decir que la formación universitaria tenga menos éxito dado que el mismo informe reconoce que, a mayor grado de titulación, mayor facilidad para la inserción laboral. Sin embargo, es especialmente significativo que la "programación, consultoría y otras actividades relacionadas con la informática" representan el sector con mayor número de contratos laborales, con una tasa de "estabilidad contractual" del 45'55%, la segunda mayor sólo por detrás de las "actividades de organizaciones y organismos territoriales". Según indica este informe Estatal, el sector informático es uno de los pocos que presenta una "variación interanual positiva" mayor del 15% en 2017.

Para mantener este panorama, se hace necesario que los estudiantes del área informática amplíen sus conocimientos en sectores tan necesarios, profusos y crecientes como el Big Data.

La Comunidad de Madrid ofrece en el sector informático dos titulaciones básicas de Formación Profesional (Informática y Comunicaciones e Informática de Oficina), una titulación de Grado Medio (Sistemas Microinformáticos y Redes) y tres títulos de Grado Superior (Administración de Sistemas Informáticos en Red, Desarrollo de Aplicaciones Multiplataforma y Desarrollo de Aplicaciones Web). Más de una titulación de las mencionadas expresa en su plan de competencias la necesidad de formarse en Base de Datos para obtener el grado de formación profesional, así como para su desempeño laboral. Sin embargo, no todas contienen formación necesaria en esta materia. A continuación se presenta un cuadro-resumen, ampliado en el *anexo 1,* que hace visible el mapa de oportunidades para este curso de formación en Bases de Datos no relacionales / *NoSQL*:

Tabla 1. Formación y Desarrollo profesional sobre Bases de Datos en Formación Profesional de la Comunidad de Madrid

	Requiere BBDD en sus competencias profesionales	La evolución de sus competencias puede requerir BBDD	Incluye BBDD en su plan formativo	Créditos ECTS específicos de Base de Datos	Formación en BBDD NoSQL
Informática y Comunicaciones	✗	✗	✗	0	✗
Informática de Oficina	✗	✓	✗	0	✗
Sistemas Microinformáticos y Redes	✓	✓	✓	Contenido dentro de otras asignaturas	✗
Administración de Sistemas Informáticos en Red	✓	✓	✓	11	✗
Desasrrollo de Aplicaciones Multiplataforma	✓	✓	✓	11	✗
Desarrollo de Aplicaciones Web	✓	✓	✓	12	✗

Fuente: Elaboración propia, datos de la Comunidad de Madrid.

Una de las funciones de la Universidad Pública desde la que se organizaría este curso de formación es la "preparación para el ejercicio de actividades profesionales que exijan la aplicación de conocimientos (...)" así como la difusión del conocimiento (...) a través de la extensión universitaria" (ampliado en anexo 2; LO 6/2001, de 21 de diciembre, de Universidades, Título Preliminar, Artículo 1, apartados 2b y 2c). Además, las universidades, escuelas y facultades públicas pueden impartir formación "conducente a la obtención de otros títulos" (misma Ley, Título II, Capítulo I, Artículo 8, apartado 1). Lo que significa que la Universidad Pública está firmemente comprometida con el desarrollo del conocimiento no sólo en los grados universitarios, en los que actualiza constantemente su temario y lo adapta a las nuevas tecnologías, sino también en el plano de la Formación Profesional, quienes también juegan un papel importante en el desempeño laboral del sector informático y como se ha visto en el cuadro anterior, disponen de escasa formación en esta materia tan necesaria para ellos.

CAPÍTULO 1. PLANTEAMIENTO DEL TRABAJO

JUSTIFICACIÓN DE LA ELECCIÓN

La elección de esta temática, como se ha introducido en el apartado anterior, reside en la carencia de formación sobre la materia tratada, ausente en los ciclos de Formación Profesional relacionados con el área informática. Una ausencia que debería ser abordada de cara al futuro en los planes de formación, pues se trata de una de las tecnologías que está cobrando más importancia con el desarrollo del Big Data y la creciente necesidad de Bases de Datos, capaces de almacenar y tratar grandes cantidades de datos de forma cada vez más especializada y adecuada a la casuística de cada tipología de datos.

OBJETIVOS

El objetivo principal de este trabajo es desarrollar el material académico y la estructura de un curso de Base de Datos *NoSQL*, orientado a alumnos de Formación Profesional (FP) de la Comunidad de Madrid.

Asimismo se pretende lograr, con este material formativo, que los alumnos de FP obtengan una base de conocimiento y comprensión sobre estas tecnologías en materia de diseño, implementación, usos, ventajas y desventajas, así como nociones prácticas sobre su manejo.

ESTRUCTURA Y DISTRIBUCIÓN DEL CONTENIDO

El contenido ha sido dividido en dos partes, una práctica y una teórica:

La parte teórica consta de 7 temas que abordan desde las Bases de Datos relacionales, como base de conocimiento necesario para comprender las nuevas tecnologías y el porqué de las necesidades que éstas cubren, hasta el desarrollo de estos nuevos tipos de Bases de Datos.

La parte teórica se presenta en forma de dos prácticas que engloban las competencias de las dos mitades del contenido teórico.

En conjunto, los temas que se desarrollan en el contenido teórico son los siguientes:

- **Introducción a Big Data**: este tema servirá como la base del problema que se tratará de solucionar con las Bases de Datos no relacionales, aportando puntos críticos en los que las Bases de Datos *SQL* actúan como cuello de botella en el desarrollo del Big Data.
- **Repaso de las Bases de Datos Relacionales**: para poder entender estas nuevas bases de datos, es necesario primero entender el estándar del que se parte, su

funcionamiento básico, tratamiento de datos, etc. De esta forma, se podrá utilizar esta tecnología como línea de partida con la que comparar las demás.

- **Bases de Datos clave-valor. *Redis***: este tema desarrollará el funcionamiento básico, con casos de uso reales, de las Bases de Datos clave-valor.
- **Bases de Datos documentales. *MongoDB***: en este tema se mostrará el funcionamiento básico, tipo de dato y casos de uso de las Bases de Datos documentales.
- **Bases de Datos columnares. *Cassandra***: este tema tratará las Bases de Datos columnares, los formatos de datos que emplea y su funcionamiento en redes de máquinas.
- **Bases de Datos para grafos. *Neo4J***: en este apartado se analizará el tipo de dato, forma de almacenamiento y usos comunes de las Bases de Datos de grafos.
- ***HDFS.* Almacenamiento de datos**: este módulo difiere de los demás en que no trata un tipo de Bases de Datos como tal, sino un sistema de almacenamiento orientado a Big Data.

CAPÍTULO 2. MARCO TEÓRICO Y DESARROLLO DEL TEMARIO DEL CURSO

INTRODUCCIÓN AL BIG DATA Y GESTIÓN DE ALMACENAMIENTO

INTRODUCCIÓN AL BIG DATA

Big Data es el término empleado para las estrategias y tecnologías no tradicionales utilizadas en la organización y proceso de información almacenada en Bases de Datos de gran tamaño.

Es una respuesta a la problemática de trabajar con cantidades de datos que superan la capacidad computacional de un simple ordenador. Esta problemática, aunque no es nueva, ha aumentado de forma exponencial en los últimos años.

DEFINICIÓN

No existe una regla establecida para determinar el tamaño necesario de una Base de Datos con el fin de ser considerada como *Big*. De forma genérica, se hace referencia a Big Data cuando el uso de técnicas y herramientas se hacen necesarias para el procesamiento de los datos almacenados.

DIFERENCIAS

Los requisitos básicos para trabajar con Big Data son los mismos que para trabajar con Bases de Datos de cualquier otro tamaño. No obstante, la escala, velocidad de absorción de información y procesamiento de éstos, así como las características de los datos con los que se debe lidiar en cada etapa del proceso, presenta retos a la hora de diseñar las soluciones. El objetivo de la mayoría de Bases de Datos de estas características, es la de mostrar conexiones en grandes volúmenes de datos heterogéneos que no sería posible descubrir con métodos convencionales.

VOLUMEN:

La escala de información a procesar ayuda a definir los sistemas Big Data. La diferencia de tamaño entre éstas y las Bases de Datos convencionales demanda la aplicación de pasos más desarrollados en cada etapa de procesamiento del ciclo de vida del almacenamiento.

En ocasiones, debido a que los requisitos de la tarea sobrepasan las capacidades de una única máquina, las tareas se convierten en un reto para agrupar, asignar y coordinar los recursos de un conjunto de máquinas. De esta forma, la gestión de clústeres y los

algoritmos capaces de dividir las tareas en subtareas procesables por las máquinas individuales del clúster son mucho más importantes que en una Base de Datos convencional.

VELOCIDAD:

Otro de los elementos que diferencian las Bases de Datos convencionales de los sistemas de Big Data es la velocidad a la que la información viaja a través del sistema. Normalmente, la información proviene de múltiples fuentes y tiene que ser procesada en tiempo real para obtener información y actualizar la comprensión actual del sistema.

Esta necesidad de obtener respuestas casi inmediatas del sistema ha llevado a muchos usuarios de Big Data a alejarse de las técnicas de procesamiento por tandas y a focalizar sus sistemas hacia un sistema de *streaming* en tiempo real. Los datos están siendo constantemente añadidos, procesados y analizados con el objetivo de mantener el ritmo al constante flujo de datos entrantes y ofrecer información de valor cuando sea relevante.

VARIEDAD:

Los problemas que se encuentran al trabajar con Big Data son, en muchas ocasiones, únicos, debido a la cantidad de diversas fuentes de datos siendo procesados y su relativa calidad.

Los datos pueden ser insertados desde sistemas internos como aplicaciones o *server logs*, desde Redes Sociales y APIs externas o desde instrumentos externos como sensores entre otros proveedores. El Big Data trata de procesar la información potencialmente útil independientemente de su procedencia mediante la consolidación de toda la información en un único sistema.

OTRAS CARACTERÍSTICAS:

Algunos individuos han propuesto expandir las “tres V” (volumen, velocidad y variedad). Estas propuestas, generalmente, hacen referencia a retos en el uso de Big Data en lugar de cualidades. Algunas de las adiciones comunes son:

Veracidad: la variedad de fuentes y complejidad de procesamiento puede llevar a dificultades en la evaluación de la calidad de los datos y, por consiguiente, del análisis resultante.

Variabilidad: las variaciones en datos llevan también a una gran variación en calidad. Fuentes adicionales pueden requerir de identificación, procesamiento o filtro de datos de baja calidad para hacerlos más útiles.

Valor: la mayor dificultad en Big Data es obtener valor. En ocasiones, los sistemas y procesos que se utilizan son lo suficientemente complejos como para que la utilización de datos y extracción de verdadero valor se pueda volver difícil.

EL CICLO DE VIDA DEL BIG DATA

EN el procesamiento de datos en un sistema de Big Data, aunque difiere en ciertos aspectos en su implementación, existen estrategias comunes, así como software que se emplea de forma generalizada.

Estos pasos, aunque posiblemente no sean aplicables a todos los casos, representan las líneas generales el *workflow* de un sistema de Big Data:

Inserción de datos en el sistema.

Persistencia de datos en almacenamiento.

Computación y análisis de datos.

Visualización de los resultados.

Antes de entrar en detalle con estos elementos del *workflow*, es importante entender el concepto de *clustered computing*, una estrategia aplicada en la mayoría de soluciones Big Data y que, en ocasiones, actúa como la base tecnológica para cada etapa del ciclo de vida.

COMPUTACIÓN EN CLÚSTER

Debido a las cualidades del Big Data, los ordenadores individuales carecen de las características para manejar los datos en la mayoría de las etapas. Para solucionar este problema, se emplean clústeres de ordenadores.

Los clústeres de máquinas en Big Data buscan aportar los siguientes beneficios:

Agrupación de recursos: combinando las capacidades de almacenamiento disponibles de máquinas más pequeñas se obtiene un claro beneficio, así como la agrupación de la capacidad computacional y de memoria, ya que en el procesamiento de largas cantidades de datos se requieren grandes cantidades de estos tres elementos.

Alta disponibilidad: Los clústeres pueden producir diversas cantidades de tolerancia a errores y garantías de continuidad del servicio para prevenir que fallos de software o hardware afecten al funcionamiento o acceso a los datos y procesamiento. Éstos son especialmente importantes con la necesidad de analíticas en tiempo real.

Fácil escalabilidad: los clústeres proporcionan una base fácilmente escalable de forma horizontal mediante la adición de nuevas máquinas al clúster; esto supone la capacidad de rápida adaptación de sistema a cambios de requisitos.

El uso de clústeres requiere de una solución de gestión de miembros de clúster, coordinación de recursos compartidos y organización del trabajo en nodos individuales. Los miembros del clúster y la localización de recursos pueden manejarse a través de software como *Hadoop's YARN.*

Las máquinas involucradas en la computación por clúster también se suelen emplear en la gestión del espacio de almacenamiento compartido. Esto será muy relevante a la hora de analizar la persistencia de datos.

INSERCIÓN DE DATOS EN EL SISTEMA

La inserción de datos es el proceso de añadir datos en crudo al sistema. La complejidad de esta operación depende, en gran medida, del formato y calidad de los datos y las fuentes y de cuánto difiere del estado deseable para ser procesado.

Existen herramientas dedicadas para la inserción de datos como *Apache Sqoop*, capaz de coger una base de datos existente y añadirla a un sistema de Big Data también ya existente.

Durante la inserción de datos, distintos niveles de análisis, ordenación y etiquetación se llevan a término. Este proceso se suele llamar *ETL (Extract, Transform, Load),* que es un término que suele referirse a procesos de *Legacy Warehousing Databases,* también aplicables a los conceptos de sistema de Big Data. Las operaciones típicas suelen incluir modificación de datos para darles formato, categorización y etiquetado, filtro de datos no necesitados o erróneos y potencialmente validar que se ciñan a ciertos requisitos.

Con esas capacidades en mente, los datos capturados deberían ser lo más crudos posible para aumentar la flexibilidad en el futuro.

PERSISTENCIA DE DATOS EN ALMACENAMIENTO

El proceso de inserción normalmente cederá los datos a los componentes que gestionan el almacenamiento para que sea almacenado en el disco de forma persistente y fiable. Aunque esto parece una operación sencilla, el volumen de datos entrantes, los requisitos de disponibilidad y la capa de computación distribuida hacen que un sistema de almacenamiento complejo sea necesario.

Esto supone aprovechar un sistema de archivos distribuido para obtener capacidad de almacenamiento en bruto. Soluciones como el sistema *HDFS* de *Apache Hadoop* permiten que grandes cantidades de datos puedan escribirse a través de múltiples nodos del clúster. Esto garantiza el acceso a los datos almacenados por los recursos, puede cargarse en la *RAM* del clúster para operaciones en memoria y, además, permite que puedan manejarse errores en los componentes.

COMPUTACIÓN Y ANÁLISIS DE DATOS

Una vez los datos ya están disponibles, el sistema puede comenzar a procesar los datos para generar la información realmente útil. La capa computacional es la parte más diversa del sistema, ya que los requisitos y procedimientos pueden variar mucho en función de qué tipos de resultados se busca obtener.

Proceso por lotes es un método de computación sobre grandes Bases de Datos. El proceso supone la separación del trabajo en partes más pequeñas; asignando cada

parte a una máquina concreta, reordenando los datos en función de los resultados intermedios y calculando y montando el resultado final. Los pasos que se llevan a término en este proceso son:

Splitting: para poder procesar los datos de forma paralela, éstos se dividen en grandes porciones llamadas *Splits.*

Mapping: los *Splits* resultantes de la primera fase son asignados a las máquinas para ser procesados y obtener lo que se denominan "resultados intermedios".

Shuffling: durante los procesos anteriores la información se estaba almacenando en los discos locales de las máquinas. Es en esta fase cuando la información comienza a viajar por la red del clúster para unirse y volver a ser particionada como preparación para la fase de *Reducing.*

Reducing: en esta fase, los datos son procesados para generar una lista final de clave-valor. Puede producirse en una única fase si todos los datos pueden ser procesados por una única máquina.

Assembling: en caso de no poder procesar todos los datos en una única máquina, se tiene que realizar este proceso en el cual las porciones de datos resultantes se unirán y, en caso de necesidad, volverán a ser asignadas y procesadas hasta obtener la lista definitiva.

Esta es la estrategia que emplean herramientas como *MapReduce* de *Apache Hadoop*. Este tipo de procesamiento (por lotes) es más útil cuando se está trabajando con grandes Bases de Datos que requieren de mucha computación.

Mientras que el procesamiento de datos por lotes se ajusta perfectamente a ciertos tipos de datos, otros requieren de procesamiento en tiempo real. El procesamiento en tiempo real necesita que la información se procese y esté disponible de forma inmediata. Una forma de lograr esto es el procesamiento en *stream,* el cual opera en un flujo continuo de datos compuesto de datos individuales. Otra característica común de los procesadores en tiempo real es la computación en memoria, que trabaja con representaciones de los datos en la memoria del clúster para evitar tener que escribir en memoria.

Apache Storm, Apache Flink y *Apache Spark* proveen de diferentes métodos para conseguir computación en tiempo real o tiempo casi real. Por supuesto, estas tecnologías suponen un compromiso para obtener la capacidad de procesamiento deseada, y estos compromisos, pueden suponer la diferencia entre cuál de los sistemas será más apropiado para nuestra aplicación. En general, la computación en tiempo real es más indicada para analizar lotes de datos más pequeños que están siendo bien cambiados bien añadidos al sistema con mucha frecuencia.

VISUALIZACIÓN DE LOS RESULTADOS

Debido al tipo de información que está siendo procesada en sistemas de Big Data, el reconocimiento de patrones o cambios en los datos a lo largo del tiempo es, en muchas ocasiones, más importante que los valores de los datos almacenados. La visualización de resultados de procesamiento de datos es una de las formas más valiosas de detectar patrones lógicos y sacar conclusiones de grandes cantidades de datos.

El procesamiento en tiempo real es usado con frecuencia para visualizar métricas de aplicación y servidores. Los datos cambian con frecuencia y, gran cantidad de detalles en las métricas, suelen indicar impactos significativos en la salud de los sistemas y organizaciones. En estos casos, proyectos como *Prometheus* pueden ser útiles para procesar los flujos de datos en tablas de tiempos que permiten visualizar esos datos.

Un método de visualización popular es *Elastic Stack*, previamente conocido como *ELK stack*. Se compone de *Logstack* para la colección de datos, *Elasticsearch* para la indexación de datos y *Kibana* para la visualización. *Elastic Stack* puede usarse con sistemas Big Data para visualizar interfaces con los resultados de cálculos y métricas sin procesar. Se puede conseguir un resultado similar con el uso de *Apache Solr* para indexar y un *fork* de Kibana llamado *Banana* para visualizar. La pila de datos resultante de llama *Silk*.

GLOSARIO

A continuación, se presenta un detalle de la terminología relacionada con Big Data, seleccionado por ser el que con mayor frecuencia se emplea:

Tabla 2. Glosario de terminología básica en Big Data

Big Data	Big Data es el término paraguas para Bases de Datos que no pueden ser gestionadas de forma tradicional por máquinas o herramientas debido a su tamaño, velocidad y variedad. Este término también se aplica a tecnologías y estrategias para trabajar con este tipo de datos.
Procesamiento en lotes	El procesamiento por lotes o *Batch processing* es una estrategia de computación que implica el proceso de datos en grandes sets. Esto, normalmente, es ideal para trabajo que no sea sensible al tiempo y trate con grandes cantidades de datos. El proceso comienza y tiempo después se reciben los resultados.
Computación en clúster	La computación en clúster es la práctica de agrupar los recursos de múltiples máquinas y gestionar sus capacidades colectivas para completar tareas. Los clústeres de máquinas requieren de una capa de gestión del clúster que se encarga de la comunicación entre los nodos individuales y coordina la asignación de trabajo.
Data lake	Este es un término para largos repositorios o colecciones de datos en un estado relativamente “crudo”. Esto se usa frecuentemente para referirse a

	las colecciones de datos en sistemas Big Data que pueden estar sin estructurar y cambiando con frecuencia. Esto difiere en espíritu de los *Data Warehouses.*
Data mining	Es un amplio término que hace referencia a la práctica de tratar de encontrar patrones en grandes sets de datos. Es el proceso de redefinir una masa de datos en un set de información más cohesivo y comprensible.
Data Warehouse	Son grandes y ordenados repositorios de datos que pueden usarse para el análisis y reporte. En contraste con el *Data Lake*, un *data warehouse* está compuesto de datos que han sido limpiados, integrados con otras fuentes, y están generalmente bien ordenados. Los *Data Warehouses* suelen relacionarse con Big Data, pero normalmente son componentes de sistemas más convencionales.
ETL	*Extract Transform and Load*. Hace referencia al proceso de preparar datos "crudos" para ser procesados por el sistema. Tradicionalmente es un proceso relacionado con los *Data Warehouses*, pero las características de este proceso son comunes a la integración de tuberías de Sistemas de Big Data.
Hadoop	Es un proyecto de *Apache* que fue un temprano éxito *Open Source* en Big Data. Consiste en sistema de ficheros distribuido llamado *HDFS*, con un gestor de clúster y programador de recursos llamado *YARN* (*Yet Another Resource Negotiator*). *MapReduce* proporciona capacidades de conmutación por lotes a este sistema. En desarrollos contemporáneos de *Hadoop* se pueden usar otros sistemas de computación y análisis de datos en conjunto a *MapReduce*.
Computación en memoria	Es una estrategia que implica mover las colecciones de datos en su totalidad a la memoria colectiva del clúster. Los cálculos intermedios no son escritos en disco, sino que se mantienen en memoria. Esto supone una gran ventaja en cuanto velocidad sobre sistemas de *input-output*, un ejemplo de computación en memoria es *Apache Spark*, un sistema *I/O* convencional sería *MapReduce* de *Hadoop*.
Machine learning	Es el estudio y práctica de diseños de sistemas que pueden aprender, ajustarse y mejorar en base a la información que reciben. Generalmente, esto supone la implementación de algoritmos predictivos y estadísticos que pueden, de forma continua, ir acercándose a un comportamiento definido y correcto a medida que van recibiendo más información.
***Map reduce* (algoritmo de Big Data)**	(No confundir con el motor de computación de *Hadoop*). Es un algoritmo para gestionar el trabajo de un clúster de computación. El proceso supone la división del problema (mapeándolo a distintos nodos) y

	procesando las divisiones para obtener resultados inmediatos. Se alinean los resultados para obtener colecciones de datos y se reducen los resultados sacando un único valor de cada set.
NoSQL	Es un término que hace referencia a las Bases de Datos diseñadas fuera del modelo tradicional relacional. Las Bases de Datos *NoSQL* suponen una variedad de compromisos respecto a las Bases de Datos convencionales, pero con frecuencia son las más adecuadas para sistemas Big Data debido a su flexibilidad y arquitectura.
Stream Processing	Es la práctica de procesar elementos individualmente según se mueven por el sistema. Esto permite un análisis en tiempo real de los datos que se están insertando en el sistema y es útil para operaciones que requieren una respuesta en un corto espacio de tiempo gracias al uso de métricas de alta velocidad.

Fuente: Elaboración propia.

CONCLUSIÓN

Big Data es un término amplio que evoluciona rápidamente. Mientras que no se ajusta perfectamente a todos los tipos de computación, muchas organizaciones están invirtiendo en Big Data para ciertas aplicaciones. Esto es debido a que, el Big Data, es único en su capacidad para detectar patrones y comportamientos que con medios tradicionales serían imposibles de encontrar.

REPASO DE BASES DE DATOS RELACIONALES

INTRODUCCIÓN

Las Bases de Datos relacionales son la solución más comúnmente implementada en la necesidad de guardar datos de forma estructurada y que respondan a una lógica predeterminada, que permita explotar la relación entre los datos al máximo.

La principal forma de interacción con Bases de Datos relacionales es *SQL* (*Structured Query Language*), lenguaje que permite realizar consultas complejas a la Base de Datos para extraer la información deseada.

MODELO RELACIONAL Y DIAGRAMAS ENTIDAD RELACIÓN

La base del modelo de Bases de Datos relacionales es el diseño basado en el modelo relacional. En este modelo, las entidades que serán representadas por tablas se relacionan entre sí utilizando distintos tipos de relaciones que determinan la dependencia que tienen unas de otras.

Las entidades generan tablas y los atributos generan las columnas que compondrán estas tablas, aunque algunos de los atributos pueden generar otras tablas.

Las relaciones son el elemento que más valor le aporta al diseño. Existen varios tipos de relaciones en función de la cardinalidad de la relación:

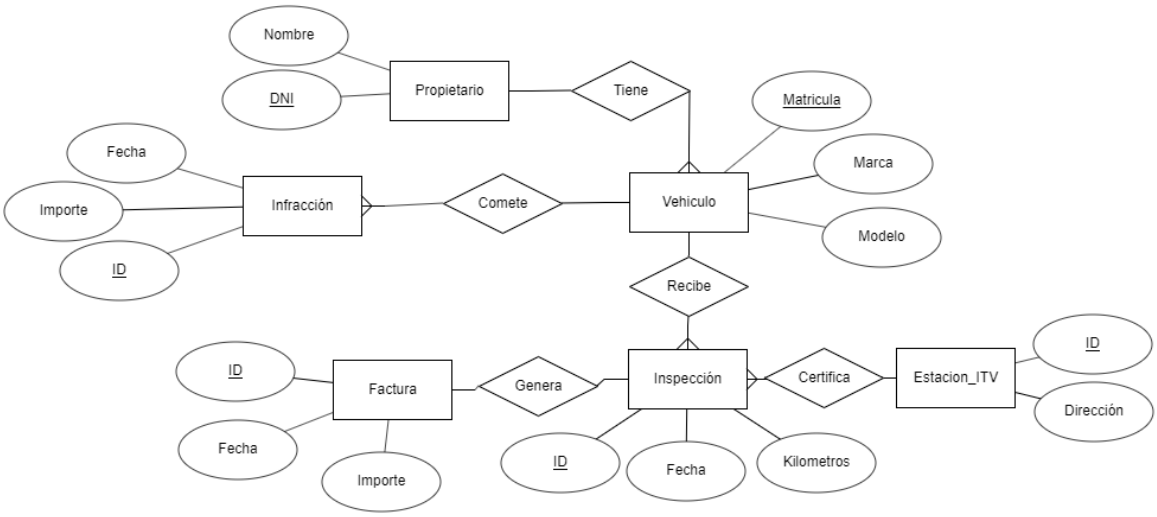
1:1, por cada entidad de un extremo de la relación, existe una entidad del otro.

1:N, por cada entidad de un extremo de la relación, existen una o más entidades del otro.

N:M, varias instancias de cada extremo de la relación existen relacionadas entre sí. Este es el tipo de relación más complejo, y existe una tabla entre ambas relaciones que recoge los identificadores de las entidades de cada extremo.

A continuación, podemos ver un modelo entidad relación:

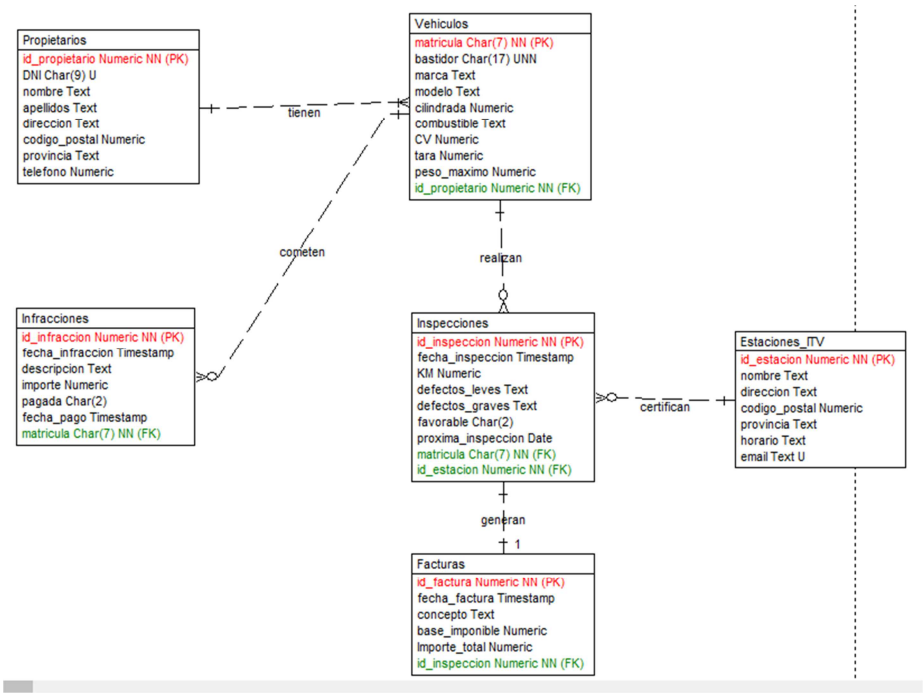
Ilustración 1. Diagrama Entidad Relación(I)



Fuente: Elaboración propia con *erdplus.com*

Una vez se ha transformado el diagrama entidad-relación en un modelo relacional donde solo aparecen tablas con sus campos y las relaciones entre los campos, se procede a exportar el modelo a una Base de Datos *SQL.*

Ilustración 2. Diagrama modelo relacional (I)



Fuente: Elaboración propia con *Toad Data Modeler*

Este es un ejemplo de modelo relacional realizado con *Toad Data Modeler*, en él se puede ver las entidades: propietarios, vehículos, etc. Estas entidades pasarán a ser las tablas en una Base de Datos relacional.

Los atributos, *IDs*, DNIs, nombres, etcétera, serán las columnas de estas tablas.

Mediante las relaciones, se determinarán las *foreign keys* y *primary keys* de estas tablas, que permitirán realizar consultas más eficientes y complejas a las tablas, reuniendo o relacionando los datos que se encuentran distribuidos entre todas las tablas.

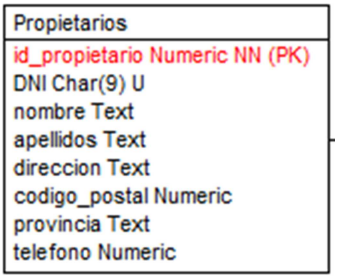
DISEÑO

Las Bases de Datos relacionales están compuestas por tablas, cada tabla representa una entidad de la que existirán numerosas instancias y, cada una de estas instancias, será una línea o registro. Las tablas están compuestas por columnas y, estas columnas, representan los campos que componen la tabla. Cada columna tiene un tipo de dato asignado, así como otras cualidades que determinarán si el dato debe ser único o si es una *key*.

Las *keys* son las bases de la estructura de una Base de Datos relacional, son el elemento que permite generar relaciones entre las tablas. Una *key* representa un campo único dentro de una tabla, puede ser un *ID.* Este campo se compartirá con otras tablas con las que la tabla inicial esté relacionada; de esta forma, en las tablas que tengan una dependencia de esta tabla, se encontrará el campo *foreign key* que representa la *key* de la entidad en concreto de la tabla inicial de la que depende ese elemento de la tabla actual.

Es decir, suponer que se tiene una tabla con los datos de un conductor:

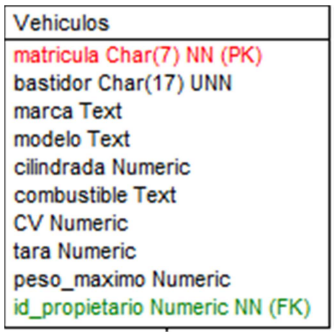
Ilustración 3. Detalle de Ilustración 2 (II)



Fuente: Elaboración propia con *Toad Data Modeler*

En este caso, se puede ver que la *primary key* es el *id*_propietario: este es el valor que se compartirá con las tablas que tengan una relación de dependencia con esta. Por ejemplo, el vehículo de este conductor.

Ilustración 4. Detalle de Ilustración 2 (III)



Fuente: Elaboración propia con *Toad Data Modeler*

Como se puede ver, en la parte inferior de la tabla se encuentra la *foreign key* que dirá a qué propietario pertenece este vehículo.

Qué tabla lleva la *PK* de la otra tabla en forma de *FK* se determinará en función de la dirección de la relación, o de qué tabla es dependiente de la otra (tabla-padre tiene la *PK* y la tabla-hija tiene la *FK*). Además, hay que tener en cuenta los tipos de relaciones entre las tablas, que pueden ser las siguientes:

1:1, lo que quiere decir que, por cada elemento de una tabla, se encontrará como máximo un elemento de la otra que corresponda al primero.

1:N, lo cual significa que a cada elemento de una tabla le corresponden “0”, “1” ó varios elementos de la otra.

N:M, cada registro en ambas tablas puede estar relacionado por varios registros de la otra. Estas tablas requieren de una tercera tabla intermedia de asociación.

Una vez determinadas las entidades y relaciones se pasa a realizar el diseño, donde es importante tener en cuenta las formas normales. Las formas normales representan grados de optimización del diseño de la base de datos para intentar repetir la menos cantidad de

información en las tablas; a mayor forma normal mayor optimización en el diseño de la Base de Datos y por tanto menor redundancia de datos.

1ª Forma normal establece que el número de valores por cada columna de un elemento será 1. Es recomendable evitar almacenar listas de elementos en una columna de un registro.

2ª Forma normal establece la dependencia total de los atributos de la *primary key.*

3ª Forma normal establece la independencia de las columnas que no sean clave de las demás columnas, para evitar que se produzcan cambios no deseados al modificar una columna no clave.

Existen más formas normales, algunas de las más elevadas se consideran únicamente teóricas y pueden incluso afectar negativamente al rendimiento de la base de datos. Lo común es normalizar hasta 3ª forma normal.

OPERACIONES BÁSICAS *SQL*

SQL es el lenguaje que se emplea para comunicarse con las Bases de Datos relacionales. Existen numerosas operaciones que pueden realizarse sobre una Base de Datos en *SQL* para obtener información, borrar, actualizar, etc. Por el momento, este material formativo se centrará en las operaciones más básicas:

SELECT

Select es la operación más básica de consulta. Esencialmente se le está pidiendo a la Base de Datos las columnas de una tabla o tablas que desea ver. La sintaxis sería la siguiente:

```
SELECT * FROM NOMBRE_TABLA
```

En este caso, al utilizar **" * "** se estaría indicando que se solicita ver todas las columnas de los elementos almacenados en esta tabla. Se podrían acotar de la siguiente manera:

```
SELECT NOMBRE_COLUMNA1,NOMBRE_COLUMNA2 FROM
NOMBRE_TABLA
```

De esta forma, se está pidiendo ver dos columnas de todos los registros de la tabla. No obstante, se podría acotar aún más el resultado limitando la cantidad de registros que se quieren ver, esto sería de la siguiente forma:

```
SELECT TOP 5 NOMBRE_COLUMNA1,NOMBRE_COLUMNA2 FROM
NOMBRE_TABLA
```

De esta forma solo se verán los 5 primeros registros de la anterior consulta. Esto no es muy eficiente de por sí, ya que por defecto se ordenarán por el valor de *ID*, pero también se le puede indicar a la consulta en qué orden se desea extraer los resultados:

```
SELECT TOP 5 NOMBRE_COLUMNA1,NOMBRE_COLUMNA2 FROM
NOMBRE_TABLA ORDER BY NOMBRE_COLUMNA1 DESC
```

En este caso se mostrarían los 5 primeros registros en función de la primera columna en orden de valor descendente, pero, ¿y si hay valores repetidos que no se desea ver?:

SELECT DISTINCT TOP 5 NOMBRE_COLUMNA1,NOMBRE_COLUMNA2

FROM NOMBRE_TABLA

ORDER BY NOMBRE_COLUMNA1 DESC

De esta forma sólo se verá los resultados que sean distintos de *NOMBRE_COLUMNA1.* Se podría averiguar cuántos valores distintos hay dentro de esta columna de la siguiente forma:

SELECT COUNT (DISTINCT NOMBRE_COLUMNA1) FROM NOMBRE_TABLA

Es importante recordar que, con ciertos operadores, es necesario hacer uso de la cláusula *GROUP BY* cuando se quiera tratar con más de una columna, como es el caso de los operadores *MIN, MAX, COUNT*, *AVG* y *SUM*, también determinado como funciones agregadas, que obtienen el mínimo, el máximo, el número de elementos, la media y la suma de una columna.

Cabe destacar, también, que no siempre se encontrarán todos los resultados en una única tabla. En estos casos, es importante haber hecho un buen trabajo en el diseño de la Base de Datos y saber emplear las relaciones:

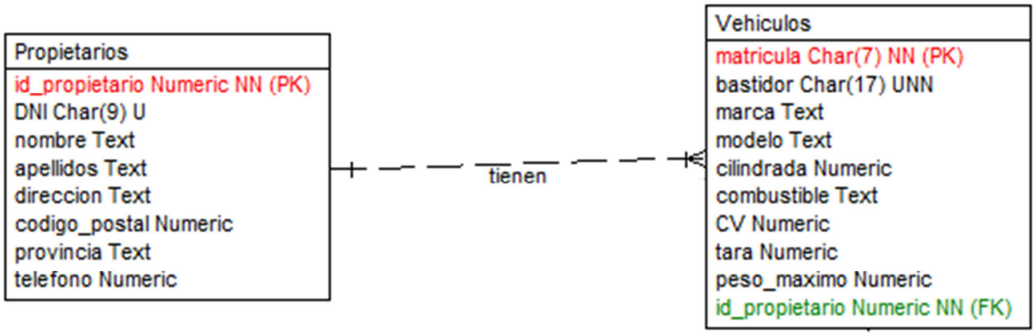
SELECT TABLA1.COLUMNA1,TABLA2.COLUMNA1

FROM TABLA1 INNER JOIN TABLA2 ON

TABLA1.PRIMARYKEY=TABLA2.FOREIGNKEY

En este caso, se está extrayendo dos columnas de dos tablas distintas que están relacionadas a raíz de la *primary key* de la ilustración 2 que hace de *foreign key* en la ilustración 3. Para ver más claramente este último ejemplo, se detallará un caso más claro a partir de la tabla de conductores y vehículos empleada de ejemplo en el apartado de diseño.

Ilustración 5. Relación entre tablas



Fuente: Elaboración propia con *Toad Data Modeler*

WHERE

SELECT…FROM…WHERE es una de las estructuras más comunes a la hora de realizar consultas a una Base de Datos. Esta cláusula permite aplicar una condición a la consulta que se está realizando.

Si se quisiera ver el nombre de los dueños de vehículos de una determinada marca, así como el modelo del vehículo, se realizaría la siguiente consulta:

```
SELECT Propietarios.nombre,Vehiculos.modelo
FROM Propietarios
INNER JOIN Vehiculos ON Propietarios.id_propietario=Vehiculos.id_propietario
WHERE Vehiculos.marca= 'opel'
```

DELETE

Se utiliza para borrar registros en una tabla y la sintaxis es similar a la de *SELECT*. Es importante acompañarla siempre de una cláusula *WHERE* para acotar los registros que deben borrarse.

```
DELETE NOMBRE_COLUMNA FROM NOMBRE_TABLA WHERE
NOMBRE_COLUMNA = 'CONDICIÓN'
```

INSERT

Este comando permite introducir registros en una tabla:

```
INSERT INTO NOMBRE_TABLA (COLUMNA1,COLUMNA2,COLUMNA3)
VALUES (VALOR1,VALOR2,VALOR3)
```

Si se está añadiendo valores a todas las columnas de la tabla no es necesario especificar las columnas después del nombre de la tabla, pero sí será preciso seguir el orden en el que se encuentran las columnas en la tabla cuando esta se creó.

UPDATE

Permite cambiar un registro dentro de una tabla. Al igual que con *DELETE,* es importante acompañarlo de un *WHERE* para asegurarse de no sobrescribir registros que no se quieran cambiar.

```
UPDATE NOMBRE_TABLA
SET NOMBRE_COLUMNA1 = VALOR1, NOMBRE_COLUMNA2 = VALOR2
WHERE ID_TABLA = N
```

CASOS DE USO

Pese a las limitaciones que presentan las Bases de Datos relacionales, que han propiciado la aparición de nuevas tecnologías para cubrir las necesidades de usuarios que habían topado con estas, *SQL* y las Bases de Datos relacionales siguen ocupando un espacio extremadamente relevante en el mercado. Su portabilidad y sencillez, así como el soporte de gigantes de la industria como *Microsoft, IBM* u *Oracle*, hacen que las Bases de Datos relacionales sigan siendo la alternativa por defecto para cualquier aplicación, pese a que, como se verá en este curso, las nuevas tecnologías le ganan terreno en ciertos campos.

BASES DE DATOS CLAVE-VALOR

INTRODUCCIÓN

Existen varios tipos de Bases de Datos *NoSQL*, que podrían dividirse en las siguientes categorías:

- Clave-Valor
- Documental
- Columnar
- Para Grafos

Dentro de estas, *Redis* se clasificaría como una Base de Datos clave-valor, esto quiere decir que los valores almacenados son mapeados por una *key*. Los valores pueden variar entre ellos y no siguen una estructura particular, no obstante, ninguno de ellos puede ser extraído si no se conoce la *key*.

Redis es un base de datos *NoSQL* de código abierto, escrita en *ANSI C* y mayoritariamente soportada en *Linux* y *SO* relacionados, aunque también existen formas de correr *Redis* en Windows.

FUNCIONAMIENTO

Redis funciona con colecciones de datos de memoria, de esta forma se obtiene una gran ventaja en rendimiento de cara a la persistencia de datos. Se puede programar un volcado de las colecciones de datos en determinados momentos o adjuntando cada comando a un *log*. La persistencia puede ser desactivada si sólo se necesita una memoria en caché con funcionalidades y en una red de trabajo.

Redis también soporta replicación asíncrona maestro-esclavo, transacciones, *Pub/sub*, *Lua Scripting* y *Keys* con *TTL* (*Time To Live,* similar al de los paquetes de red) limitado, entre otros.

Debido a la volatilidad de los datos que, esencialmente, están almacenados en RAM, *Redis* se emplea con frecuencia como una Base de Datos secundaria que soporta a una primaria con mayor persistencia pero menor rendimiento.

TIPOS DE DATOS *REDIS*

Redis no es simplemente un almacenador de valores por *key*; su funcionamiento es el de un servidor de estructuras de datos, soportando diferentes tipos de valores. Esto significa que, mientras en un almacenador de valores por *key* tradicional se asociaron *strings keys* con valores *string*, en *Redis*, el valor no está limitado a un simple *string*, sino que puede almacenar estructuras más complejas, desde datos binarios hasta datos de tipo *stream*, de los cuales se hablará más adelante.

LAS CLAVES DE *REDIS*

Las claves de *Redis* son *Binary safe*, esto quiere decir que puede emplearse cualquier secuencia binaria para construir la *key*, desde un *string* hasta el contenido de un archivo *jpeg*. Un *string* vacío también sería un a *key* válida.

Las claves muy largas no son recomendables; por ejemplo, una *key* de 1024 bytes sería muy costoso tanto en memoria como a la hora de realizar una búsqueda.

De forma similar, las claves muy cortas también pueden resultar problemáticas; por ejemplo, no existe beneficio en escribir *u1000flw* como clave cuando podemos escribir *user:1000:followers.* El segundo formato ofrece una lectura más sencilla y el espacio añadido es mínimo en comparación con el usado por el objeto *key* como tal y el objeto valor.

Las *keys* pueden configurarse como *keys* con *TTL* limitado, esto permite añadir un *timeout* a cualquier *key*; cuando este expira, la *key* se destruye. Estas *key* pueden configurarse con precisión de segundos y milisegundos. La información se vuelca en disco cuando la *key* está a punto de expirar.

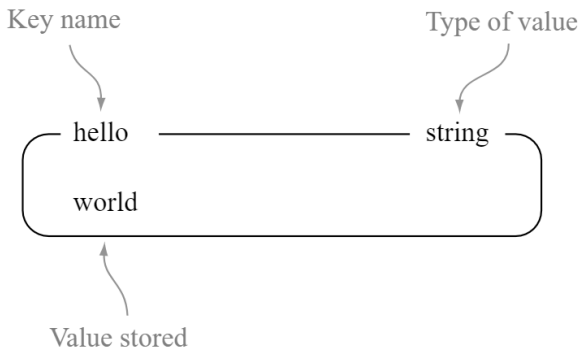
REDIS STRINGS

Los *strings* de *Redis* son el tipo de valor más simple que se puede asociar con una *key Redis*. Es el único tipo de dato en *Memcached*, por eso es recomendable para principiantes de *Redis.*

Ya que las *keys* de *Redis* son *strings*, cuando se usa un valor *string* en el almacenamiento, se está mapeando un *string* a otro. Los *strings* son valiosos en numerosas aplicaciones, como atrapando fragmentos HTML o páginas.

El manejo de estos valores se realiza, generalmente, a través de comandos *Get* y *Set,* aunque, por supuesto, otros comandos son empleados, como pueden ser *INCR,* que parsea elementos *string* a *integer*.

Ilustración 6. *Redis String*



Fuente: *Redis in action*, ebook.

REDIS LISTS

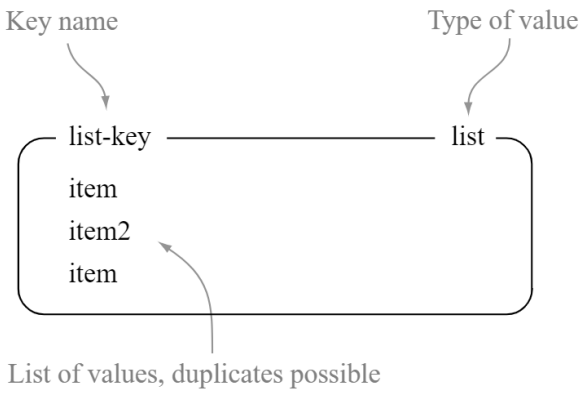
Desde un punto de vista general, las listas son secuencias ordenadas de elementos; no obstante, las características de una lista implementada usando un *array* difieren de las de una lista implementada con *linked list*.

Las listas de *Redis* son implementadas como *linked list*. Esto quiere decir que, aunque haya millones de elementos almacenados dentro de una lista, la operación de añadir un elemento a la cabeza o cola de la lista se realiza en tiempo constante. La velocidad de añadir un nuevo elemento con el comando *LPUSH* a la cabeza de una lista con 10 elementos es la misma que añadirlo a una con 10 millones.

¿Cuál es el contratiempo de esto? Acceder a un elemento a través de *index* es muy rápido en listas implementadas como *arrays*, que tienen acceso en tiempo constante al *index*, pero no es tan rápido en listas implementadas como *linked list*, donde el acceso a un elemento requiere una cantidad de trabajo proporcional al índice del elemento al que se pretende acceder.

Entonces, ¿por qué están las listas de *Redis* implementadas de esta forma? Para un sistema de Bases de Datos es crucial poder añadir elementos a una lista muy larga de forma rápida. Otra gran ventaja de las listas de *Redis* es que pueden llevarse a longitudes constantes en tiempo constante.

Ilustración 7. *Redis List*



Fuente: *Redis in action*, ebook.

CASOS DE USO COMUNES PARA LAS LISTAS

Entre las posibles tareas para las que se pueden emplear estas listas se encuentran, entre otros, los siguientes ejemplos:

- Recordar las últimas actualizaciones de un usuario en una Red Social.
- Comunicación entre procesos, utilizando un patrón de consumidor-productor donde el productor "empuja" elementos en la lista y el consumidor los consume y ejecuta acciones (*Redis* dispone de listas específicas para realizar este escenario de forma más fiable y eficiente).

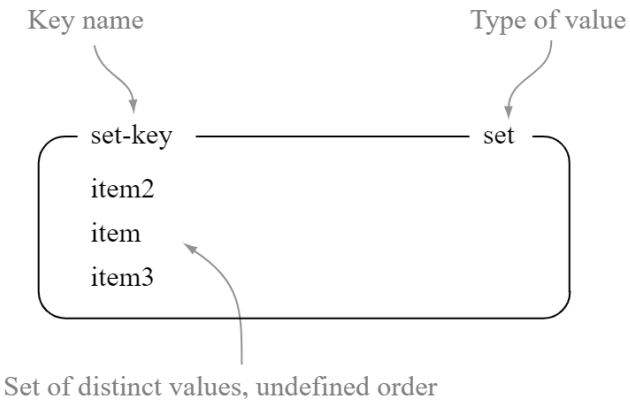
Algunos ejemplos de librerías *Ruby* que utilizan listas *Redis* son *resque* y *sidekiq*. Ya que *Redis* suele utilizarse como Base de Datos secundaria, estas librerías son empleadas para realizar trabajos en segundo plano.

Twitter utiliza las listas *Redis* para guardar los últimos tuits publicados por usuarios.

REDIS SETS

Los sets de *Redis* son colecciones de *strings* no ordenados. Los comandos *SADD* añaden nuevos elementos al set. También es posible realizar un número de operaciones a los sets como probar si un elemento ya existe dentro del set o realizar una intersección, unión o diferencia entre múltiples sets.

Ilustración 8. *Redis* Set

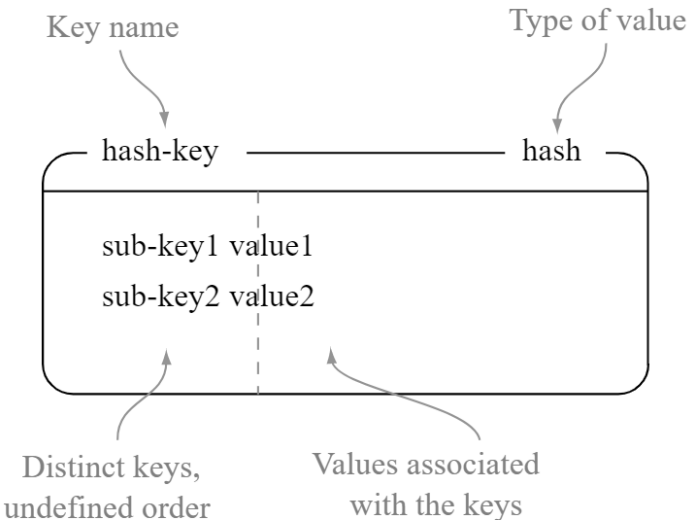


Fuente: *Redis in action*, ebook.

REDIS HASHES

Los *hashes* de *Redis* son simplemente *hashes* con pares de campo valor.

Ilustración 9. *Redis Hash*



Fuente: *Redis in action*, ebook.

BITMAPS

Los *bitmaps* no son exactamente tipos de datos, sino un set de operaciones orientadas a bits y definidas en un tipo *string*. Como los *string* son *binary safe* y su máxima medida es 512MB pueden guardar hasta 2^{32} bits distintos.

TIPOS DE DATOS *STREAM*

Stream es un tipo de dato Introducción *Redis* 5.0. Este modela una estructura de *log* de datos de forma abstracta, no obstante, la estructura de *log* se mantiene intacta: como fichero *log*, normalmente implementado como un archivo abierto únicamente de forma adjunta, *Redis Streams* son principalmente solo adjuntos. Al menos de forma conceptual, ya que, al ser un tipo de dato abstracto representado en memoria, implementan operaciones más potentes para sobrepasar las limitaciones del archivo *log*.

Lo que hace de *Redis Stream* el tipo más complejo de *Redis*, a pesar de la estructura simple, es que implementa adicionalmente características no obligatorias: un set de operaciones de bloqueo permitiendo a consumidores esperar a que nuevos elementos sean añadidos a un *stream* por el productor.

Otro elemento de *Redis Stream* son los grupos de consumidores, introducidos inicialmente por el sistema de mensajería *Kafka*. *Redis* implementa una idea similar en términos totalmente distintos, aunque el objetivo es el mismo: permitir a un grupo de clientes cooperar consumiendo una porción diferente del mismo *stream* de mensajes.

CONCEPTOS BASICOS *REDIS STREAM*

Streams son estructuras de datos a los que se accede de forma adjunta, el comando con el que se escriben es *XADD* y adjunta una nueva entrada al *stream* dado. Una entrada en *stream* no es un simple *string,* sino un compuesto de múltiples pares campo valor. De esta forma, cada entrada en el *stream* está estructurada, como un archivo solo adjunto escrito en *CSV* donde varios campos separados están presentes en cada línea.

IDS DE ENTRADA

El *ID* de entrada es proporcionado por el *XADD* e identifica de forma inequívoca cada entrada en el *stream*. Está compuesto de dos partes:

<millisecondsTime>-<sequenceNumber>

La parte de milisegundos es el tiempo local del nodo de *Redis* que está generando el *ID*, No obstante, si el valor es menor al de la entrada anterior, el tiempo de la entrada anterior es el que se usa, de forma que, si el reloj se atrasa, se mantiene la propiedad incremental del valor de milisegundos del *ID*.

El número de secuencia se usa para entradas creadas en el mismo milisegundo, tiene un peso de 64 bit. En términos prácticos, no existe un límite al número de entradas que pueden generarse dentro de un mismo milisegundo.

La razón para este formato de *IDs* es que *Redis* soporta *queries* por rango, al utilizar tiempo como *ID*, *Redis* permite realizar *queries* por rango de tiempo de forma “gratuita”.

PROGRAMAR CON *REDIS*

Las interacciones con *Redis* se realizan a través de comandos, algunos de los cuales ya se han expuesto al explicar los tipos de datos. Todos los comandos disponibles, así como documentación relacionada a ellos, puede encontrarse en la documentación oficial de *Redis.*

Hay algunas estrategias que pueden resultar útiles a la hora de gestionar una Base de Datos con *Redis*, como es *pipelining*, que supone la capacidad de enviar varios comandos a la vez, así como otras funciones básicas de gestión de Bases de Datos como *PUB/SUB*, *Lua scripting*, optimización de memoria, etc.

PIPELINING

Un servidor puede ser configurado de forma que pueda procesar nuevas peticiones, aunque el cliente no haya leído respuestas antiguas. De esta forma es posible enviar múltiples comandos al servidor sin esperar a las respuestas y, cuando estén listas, leer todas en un solo paso.

Esta técnica no supone una novedad; el protocolo *POP3* soporta esta función, de forma que aumenta drásticamente la velocidad de descarga de nuevos ***mails*** del servidor.

PIPELINING VS *SCRIPTING*

Mediante el uso de *Redis Scripting*, se puede afrontar un número de casos de uso de forma más eficiente que realizando el trabajo necesario en el servidor. Una de las ventajas de *scripting* es que es posible leer y escribir con mínima latencia, lo que hace que operaciones como leer, computar o escribir sean muy rápidas (en este caso *pipelining* no podría resultar de ayuda, ya que para escribir necesitaría de la respuesta del comando “leer”).

REDIS PUB/SUB

Es la implementación del paradigma publicar/subscribir. Esto hace que los mensajes se encaminen en canales; los subscriptores muestran interés por el contenido de estos canales y reciben el contenido que viaja por ellos, lo cual permite mayor escalabilidad y una topología de red más dinámica frente al paradigma de enviar mensajes individuales a cada subscriptor.

REDIS LUA SCRIPTING

Lua es un lenguaje de programación, es ligero y está diseñado para ser embebido en *scripts*. Soporta programación procedural, programación orientada a objetos, programación funcional, programación orientada a datos y descripción de datos.

Dentro de *Redis* se dispone de *EVAL* y *EVALSHA.* Estos son los comandos que utilizamos para evaluar los *scripts* con el interpretador de *lua* integrado en *Redis.*

El funcionamiento es el siguiente: el primer argumento del *EVAL* será un *script* de *Lua* el cual no llevará funciones de *Lua*, símplemente será un programa que correrá en el contexto del servidor de *Redis.*

El segundo argumento del *EVAL* será el número de argumentos que siguen al *script*, comenzando por el tercero, que representan los nombres de las *keys* de *Redis*. *Lua* puede acceder a los argumentos utilizando la variable global de las llaves en forma de *array* base *(KEYS[1],KEYS[2],...).* Todos los argumentos adicionales no son representados por llaves y *Lua* accede a ellos a través de la variable global *ARGV,* de forma similar a las llaves *(ARGV[1], ARGV[2],...).*

Un ejemplo del comando al completo es el siguiente:

```
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
    1) "key1"
    2) "key2"
    3) "first"
    4) "second"
```

Redis incorpora un *debugger* de *Lua* llamado *LDB.*

OPTIMIZACIÓN DE MEMORIA

Redis incorpora una serie de "*checks*" que permiten evitar posibles problemas de memoria. Algunos de estos elementos son:

- Codificación especial de tipos de pequeños datos agregados.
- Uso de instancias de 32bits.
- Operaciones a nivel de bit y byte.
- Uso de ***hashes*** cuando sea posible.
- Uso de ***hashes*** para abstraer un plan llave-valor muy eficiente en memoria por encima de *Redis.*
- Reserva de memoria.

Redis es una herramienta en constante desarrollo, de forma que muchos de estos elementos se encuentran en *beta* y bien podrían no representar la totalidad de las herramientas y funciones disponibles en el momento de lectura.

INSERCIÓN EN MASA DE *REDIS*

En ocasiones las instancias de *Redis* tienen que cargarse con una gran cantidad de datos generados por el usuario preexistentes al momento de la carga, los cuales deben

cargarse en el tiempo corto de forma que millones de *keys* se creen tan rápido como sea posible.

Esto se llama inserción en masa y se realiza mediante la generación de un protocolo. El protocolo *Redis* es extremadamente sencillo de generar y parsear. Para esto, se tienen que llevar a ejecución una serie de comandos.

El archivo que tiene que generarse para realizar la inserción en masa está compuesto por comandos. *Redis* incorpora *pipe mode,* que fue diseñado de cara a realizar inserciones en masa. Este es uno de los comandos recomendados para realizar inserciones en masa.

PARTICIONADO

El particionado es el proceso de dividir los datos en múltiples instancias, de forma que cada instancia solo contendrá una porción de las *keys.*

¿Por qué es esto útil? Particionar Bases de Datos en *Redis* permite alcanzar principalmente dos metas:

> Permite Bases de Datos mucho más grandes, utilizando la suma de memoria de un clúster de ordenadores. Sin particionar, se está limitado a la cantidad de memoria de una única máquina.
>
> Permite escalar el poder computacional a múltiples núcleos y múltiples máquinas y el ancho de banda de varios adaptadores de red.

Existen distintos criterios de partición. El método más sencillo de partición es la partición por rangos, y se consigue mediante el mapeo de rangos de objetos en instancias específicas de *Redis.* Por ejemplo, usando *ID* de 0 a 10k. De esta forma, las particiones quedarían así:

R0= ID0 a ID 10000

R1=ID10001 a ID20000

Este método, aunque en la práctica se utiliza, tiene la desventaja de requerir una tabla que mapee los rangos de instancias. Esta tabla tiene que ser gestionada y una instancia de tabla es necesaria por cada tipo de objeto, de forma que la partición por rango no es la solución más deseable por su ineficiencia respecto a otras alternativas.

La alternativa a la partición por rangos es la partición por *has,* que funciona con cualquier *key*. Se toma el nombre de la *key* y se usa una función *hash* y se convierte en un nombre. Después, se emplea un módulo de operación para transformar el *hash* entre “0” y “3”, de forma que este puede ser mapeado a una de las cuatro instancias de *Redis* -esto sería en caso de tener un clúster de cuatro máquinas.

Existen diferentes implementaciones de particionamiento en *Redis* y puede ser responsabilidad de distintas partes de la pila de software:

Particionado en cliente: el cliente directamente selecciona el nodo correcto donde escribir o leer una *key* en concreto. Muchos clientes implementan Particionamiento por cliente.

Particionado asistido por *Proxy*: el cliente envía una petición a un *proxy* que tiene la capacidad de comunicarse con el protocolo *Redis*, en lugar de enviarlo directamente a la instancia de *Redis.* El *proxy* será el encargado de mandar la petición a la instancia de *Redis* que considere correcta y reenviará las respuestas de vuelta al cliente.

Ruteado por *Query:* se envía la petición a una instancia al azar. Esta instancia se encargará de enviar la petición al nodo adecuado. *Redis* Clúster implementa un sistema híbrido en el que la petición no se envía directamente de una instancia de *Redis* a otra, sino que un cliente es redireccionado al nodo adecuado.

DESVENTAJAS DEL PARTICIONADO:

El particionado también tiene repercusiones negativas a tener en cuenta, estas son:

Operaciones que involucran múltiples llaves no suelen estar soportadas.

Transacciones que requieren de varias ***keys*** no están soportadas.

La granularidad de la partición es la llave, de forma que no es posible encapsular una Base de Datos con una sola gran llave como si fuese una colección de datos ordenada.

Cuando se usa particionamiento, el manejo de datos se vuelve más complejo, por ejemplo, es necesario manejar varios archivos *RDB/AOF* y, para realizar un *backup*, se tienen que agregar los ficheros persistentes desde varias instancias y huéspedes.

Añadir y quitar capacidad se vuelve complejo. Por ejemplo, los clústeres de *Redis* soportan un rebalanceo de los datos más o menos transparente con la habilidad de añadir y quitar nodos en tiempo de ejecución, pero otros sistemas como el particionamiento desde cliente y *proxies* no soportan esta funcionalidad. Existe una funcionalidad llamada *pre-sharing* que puede ayudar con este problema.

APLICACIONES

La aplicación general de *Redis* es como Base de Datos secundaria que se emplea de soporte a una Base de Datos primaria, lo cual se debe a la ligereza y velocidad de *Redis*. Un ejemplo de

esto es *Twitter,* que utiliza lo que han denominado la arquitectura de entrega en tiempo real, donde los tuits más recientes se encuentran en una Base de Datos *Redis.*

Github también hace uso de esta tecnología para hacer el funcionamiento del portal más fluido mediante llamadas a *Redis* que corre en los servidores de Bases de Datos.

Stackoverflow usa *Redis* como capa de caché para toda la red.

Craigslist utiliza *Redis Presharing* para gestionar los clústeres de sus sistemas.

BASES DE DATOS DOCUMENTALES

INTRODUCCIÓN

MongoDB es una Base de Datos documental, lo que significa que un registro en *MongoDB* es un documento; esto es, una estructura de datos compuestas de pares de campos y valores. Los documentos de *MongoDB* son similares a los Objetos *JSON*. Los valores de los campos pueden incluir otros documentos, *arrays* y *arrays* de documentos.

Ilustración 10. Documento en *MongoDB*

```
{
  name: "sue",                          ←— field: value
  age: 26,                              ←— field: value
  status: "A",                          ←— field: value
  groups: [ "news", "sports" ]          ←— field: value
}
```

Fuente: Documentación Oficial *MongoDB* 4.0

Las **ventajas** de usar documentos son:

- Los documentos corresponden con los datos nativos de muchos lenguajes de programación, por ejemplo, objetos.
- Los documentos embebidos y ***arrays*** reducen la necesidad de ***join*** que pueden ser costosos en recursos.
- Los esquemas dinámicos soportan un polimorfismo fluido.

Las **características clave** que hacen que *MongoDB* sea una opción a la hora de seleccionar un sistema de almacenamiento son las siguientes:

- **Alto rendimiento**: *MongoDB* ofrece alto rendimiento con persistencia de datos, en particuar:
  - Soporte de modelos de datos embebidos que reduce las entradas y salidas en el sistema.

Los índices soportan ***queries*** más rápidas que pueden incluir llaves de documentos embebidos y *arrays.*

Lenguaje rico en *Queries*: *MongoDB* soporta un lenguaje rico en *queries* para mantener operaciones de lectura y escritura (*CRUD)* así como:

Agregación de datos.

Búsqueda de texto y *queries* geoespaciales.

Alta disponibilidad: la habilidad de replicación de *MongoDB*, llamada *Replica Set*, proporciona:

***Failover* automático.**

Redundancia de datos.

Una ***Replica Set*** es un grupo de servidores *MongoDB* que mantienen la misma colección de datos, ofreciendo redundancia y aumentando la disponibilidad de los datos.

Escalabilidad Horizontal: *MongoDB* proporciona escalabilidad horizontal como parte de su funcionalidad *Core*:

Sharing distribuye la información a través del clúster de máquinas.

La creación de zonas de datos basadas en claves compartidas. En un clúster balanceado, *MongoDB* direcciona las lecturas y escrituras cubiertos por una zona únicamente a los elementos que están en esa zona.

Soporte para múltiples motores de almacenamiento: *MongoDB* soporta múltiples motores de almacenamiento:

WiredTiger Storage Engine (incluye soporte para *Encryption at Rest*).

In-Memory Storage Engine.

MMAPv1 Storage Engine (Obsoleto en *MongoDB* 4.0).

Adicionalmente, *MongoDB* proporciona una API de motor de almacenamiento que permite a terceros desarrollar motores de almacenamiento para *MongoDB.*

FUNCIONAMIENTO

MongoDB guarda documentos *BSON* en colecciones y estas colecciones en Bases de Datos.

BASES DE DATOS

En *MongoDB*, las Bases de Datos guardan colecciones de documentos.

La interacción con Bases de Datos se realiza a través de comandos, en este caso, el comando *USE.* Este comando permite seleccionar la Base de Datos que se va a utilizar. En caso de que la Base de Datos no exista, *MongoDB* la creará en el momento en que se inserte el primer dato.

Use DATABASE_NAME

Las inserciones de datos se realizan a través del comando *insertOne(),* que crea e inserta el registro en la Base de Datos en caso de que esta no exista. Es importante tener en cuenta que existen restricciones a la hora de nombrar las Bases de Datos, las cuales son sensibles al sistema operativo en el que esté desplegada, pero generalmente hacen referencia a caracteres especiales.

db.name.insert({"name":"record name"})

db.name.insertOne({"name":"record name"})

db.name.insertMany([{"name":"record name"},{"name":"record name"}])

COLECCIONES

Los documentos se guardan en colecciones en *MongoDB*. Las colecciones son lo análogo a las tablas en Bases de Datos relacionales.

Para crear una colección, basta con usar el comando *insertOne()* o el comando *createIndex().* Al igual que con las Bases de Datos, existen restricciones en los nombres que se le pueden dar a las colecciones.

Creación explícita:

MongoDB proporciona el comando *db.createCollection()* para crear de forma explícita una colección con varias opciones, como establecer la máxima capacidad de reglas de validación documental. Es posible modificar estas opciones a través de *collMod.*

db.createCollection("colName", {capped: true, autoindex: true })

{"name": "record name"}

Validación de documento

Por defecto, una colección no necesita que sus documentos tengan la misma estructura, los documentos que están en la misma colección no tienen por qué compartir los mismos campos o tipos de datos por campos; pueden diferir en los documentos de la colección.

Modificación de las estructuras de los documentos

Para cambiar la estructura de los documentos en una colección, como añadir nuevos campos, quitar campos existentes o cambiar los valores de un campo a un tipo distinto, se realiza una actualización en la estructura de los documentos.

Identificadores únicos

A las colecciones les son asignadas un *UUID*. El *UUID* se mantiene en todos los miembros del set réplica y nodos en un núcleo compartido.

DOCUMENTOS

MongoDB guarda los registros en documentos *BSON*. Los *BSON* son representaciones binarias de documentos *JSON* y contienen más datos que estos *(JSON).*

Estructura de los documentos

Los documentos *MongoDB* están compuestos por pares de campos y valores *field1: value1.*

El valor de un campo puede ser cualquiera de los tipos de datos *BSON*, incluyendo otros documentos, *arrays* y *arrays* de documentos.

Nombres de campo

Los nombres de los campos son *string*. Los documentos tienen restricción en los nombres que pueden tener los campos.

Los documentos *BSON* pueden tener más de un campo con el mismo nombre. Sin embargo, la mayoría de las interfaces de *MongoDB* representan la Base de Datos con una estructura que no soporta los nombres duplicados, por ejemplo, tablas *hash*.

Algunos documentos creados por *MongoDB* de forma interna pueden tener campos duplicados, pero ningún proceso de *MongoDB* añadirá campos duplicados a un documento ya existente.

Límite de valor del campo

Para colecciones de índices, los valores de los campos indexados tienen una longitud de llave máxima.

NOTACIÓN ***DOT***

MongoDB utiliza la notación *dot* para acceder a los elementos de un *array* y a los elementos embebidos en un documento.

Arrays: para especificar o acceder a un elemento de un ***array*** por la posición del índice basado en “0”, se concatena el nombre del ***array*** con punto (.) y el índice de posición basado en cero *("<array>.<index>"),* por ejemplo en el *array*:

contribs: ["Turing machine", "Turing test", "Turingery"]

Si se quisiera acceder al tercer elemento del ***array*** se emplearía:

“contribs.2”

Documentos embebidos: para especificar o acceder al campo de un documento embebido con la notación *dot*, se concatena el nombre del documento embebido con *(.)* y el nombre del campo, entre comillas *("<embedded document>.<field>"),* por ejemplo:

name: { first: "Alan", last: "Turing" },

contact: { phone: { type: "cell", number: "111-222-3333" } },

Si se deseara especificar el campo llamado "*last*" en el campo "*name*", se utilizaría la notación *dot* de la siguiente forma:

"name.last"

Si se quisiera especificar el "*number*" en el documento "*phone*" en el campo "*contact*" usando la notación *dot*, se realizaría de la siguiente forma:

"contact.phone.number"

LIMITACIONES DE LOS DOCUMENTOS

Los documentos tienen los siguientes atributos:

Límite del tamaño de documento

El tamaño máximo de un documento *BSON* es de 16 megabytes. Este tamaño máximo de un documento ayuda a asegurar que un único documento no pueda usar una cantidad excesiva de la *RAM* o, durante las transmisiones, exceder el máximo de ancho de banda. Para guardar documentos de mayor tamaño que el límite***,*** *MongoDB* proporciona la API *GridFS.*

Orden de los campos del documento

MongoDB preserva el orden de los campos en el documento siguiendo las operaciones de escritura, a excepción de los siguientes casos:

El campo *_id* siempre será el primer campo del documento.

Las actualizaciones que incluyen renombrar nombre de campos pueden resultar en el reordenamiento de los campos en el documento.

El campo *_id*

En *MongoDB,* cada documento que se guarda en una colección necesita un campo *_id* único que actúa como *primary key*. Si un documento insertado omite el campo *_id*, *MongoDB* genera automáticamente un *ObjectId* para el campo *_id.*

Esto también aplica a documentos insertados a través de operaciones *update* con *upsert: true.*

El campo *_id* tiene los siguientes comportamientos y límites:

Por defecto, *MongoDB* crea un índice único del campo *_id* al crear una colección.

El campo *_id* siempre es el primer campo en un documento. Si el servidor recibe un documento que no tenga el campo *_id* en esta posición, el servidor moverá el campo a la primera posición.

El **c**ampo *_id* puede contener valores de cualquier tipo de dato *BSON* además de *arrays.*

Para evitar problemas con las funciones de replicación, es recomendable no guardar valores del tipo de expresiones regulares de *BSON* en el campo *_id.*

OTROS USOS DE LA ESTRUCTURA DE DOCUMENTO

Además de los registros de datos*, MongoDB* utiliza la estructura del documento para otras tareas como filtrar *queries*, actualizar e indexar documentos específicos.

Documentos de filtrados de *Query*

Los documentos de filtrado de *queries* especifican las condiciones que determinan qué registros se seleccionan para lectura, actualización y borrado.

Se utiliza las expresión *<field1>: <value1>* para la condición de igualdad y la expresión de los operadores de *query* *<field2>: { <operator>: <value> }*

Documentos específicos de actualización

Estos documentos utilizan los operadores de actualización para especificar las modificaciones a realizar sobre un campo específico durante una operación *db.collection.update().* El resultado tiene este aspecto:

```
<operator1>: { <field1>: <value1>, ... },
<operator2>: { <field2>: <value2>, ... },
```

Documentos específicos de indexación

Estos documentos definen los campos a indexar y el tipo de índice. Tienen el siguiente aspecto:

```
{ <field1>: <type1>, <field2>: <type2>, ...  }
```

TIPOS DE *BSON*

BSON es un formato binario de serialización usado para guardar documentos y hacer llamadas de procesamiento remotos en *MongoDB.*

Cada tipo de *BSON* tiene tanto un identificador *integer* como un identificador *string*. Algunos ejemplos son los siguientes:

Tabla 3. Identificadores en tipos de *BSON*

Type	Number	Alias
Double	1	"double"
String	2	"string"
Object	3	"object"
Array	4	"array"
Binary data	5	"binData"

Fuente: Documentacion Oficial *MongoDB* 4.0

Estos valores se pueden usar con el operador *$type* para hacer *queries* a documentos en función del tipo. El operador agregado *$type* devuelve el tipo de una expresión operador utilizando el campo *string* del tipo *BSON*.

Existen consideraciones especiales para tipos de *BSON* específicos:

OBJECTID

Son pequeños, únicos, rápidos de generar y ordenados. Los valores de *ObjectId* consisten en 12 bytes, donde los 4 primeros bytes son el *timestamp* que refleja la creación del *ObjectId*. La estructura es la siguiente:

Un valor de 4 bytes representando los segundos desde el *Unix epoch.*

Un valor *random* de 5 bytes.

Un contador de 3 bytes, comenzando con un valor *random.*

En *MongoDB,* cada documento guardado en una colección necesita de un *_id* único que actúa como *primary key*, funciona de la misma forma que en los documentos.

En la *shell* de *MongoDB*, se puede acceder al tiempo de creación del *ObjectId* usando el método *ObjectID.GetTimestamp()*

Ordenando por campo *_id* que guarda valores *ObjectId* es equivalente a ordenar por orden de creación.

STRING

Los *string* de *BSON* son *UTF-8*. En general, los ***drivers*** de cada lenguaje de programación convierten del formato de *string* propio del lenguaje a *UTF-8* al serializar y deserializar. Esto hace posible guardar la mayoría de caracteres internacionales en *strings BSON*. Además, *$regex* de *MongoDB* soporta *UTF-8* en *strings regex*.

TIMESTAMPS

BSON tiene un tipo especial de *timestamp* para uso interno de *MongoDB* y no está asociado con el tipo *Date* normal. Los valores de *timestamp* son valores de 64 bit:

Los primeros 32 bits son el valor *time_t* desde el *Unix epoch.*

Los 32 bits posteriores son un *ordinal* que va incrementándose para operaciones entre un determinado segundo.

En una única instancia de *MongoDB*, los valores de *timestamp* son siempre únicos.

En replicación, el *oplog* tiene un campo *timestamp*. Los valores en este campo reflejan el tiempo de operación, que usa un valor *timestamp* de *BSON.*

Si se inserta un documento que contenga un *timestamp* de *BSON* vacío en un campo del nivel superior, el servidor de *MongoDB* reemplazará el valor vacío con el valor actual del *timestamp*. Si el *timestamp* fuese un campo en un documento embebido, el servidor lo dejaría como un valor *timestamp* vacío.

DATE

Los *Date* de *BSON* son *integers* de 64 bit que representan el número de milisegundos desde el *Unix epoch*. Esto es una fecha representable de un rango de 290 millones de años hacia el pasado y futuro.

La documentación oficial de *BSON* hace referencia al tipo *Date de BSON* como a un tipo *UTC datetime.*

APLICACIONES

El uso general de *MongoDB* es como sustitutivo de Bases de Datos relacionales, para manejar contenido semiestructurado, analíticas y registro de datos en tiempo real y cacheo con alta estabilidad.

No obstante, está limitado en su aplicación por la preexistencia de sistemas que requieren de *SQL* y en aplicaciones con alta densidad de transacciones.

Muchas compañías utilizan *MongoDB*, desde *craiglist* hasta *GitHub*, en muchas ocasiones como Base de Datos principal apoyada por una Base de Datos secundaria que funciona en la *RAM* del clúster de máquinas como puede ser *Redis.*

El razonamiento para seleccionar esta tecnología para, en la mayoría de los casos, reemplazar Bases de Datos *SQL* convencionales, es la incapacidad de archivar datos de forma rápida al hacer muchas modificaciones en las columnas de las Bases de Datos del clúster, las cuales quedarían apiladas en la Base de Datos de producción. Asimismo, ofrece mayor capacidad de escalabilidad que alternativas más convencionales como *MySQL*.

BASES DE DATOS COLUMNARES

INTRODUCCIÓN

Apache Cassandra es un Base de Datos no relacional, masivamente escalable, que ofrece disponibilidad constante. Otros elementos significativos son la simplicidad operacional y facilidad de distribución de datos a lo largo del sistema y zonas disponibles de la nube y centros de datos.

Originalmente fue un proyecto de *Facebook* de código abierto desde 2008 y un proyecto de *Apache* desde 2010.

Por supuesto, *Cassandra* es una Base de Datos *NoSQL.* Estas Bases de Datos responden a arquitecturas más adheridas a los diseños de aplicaciones modernas, aunque hacen compromisos con respecto a las Bases de Datos convencionales, de acuerdo con el teorema de *CAP.*

Entre estos compromisos destacan la carencia de *Join* entre tablas automáticas y transacciones *ACID* que se verá más adelante; en función de la aplicación, los compromisos pueden resultar interesantes por la posibilidad de que reporten mayor beneficio.

En la siguiente ilustración se puede ver la estructura de las columnas que se almacenan en *Cassandra*; se aprecia claramente uno de sus beneficios, y es que el espacio solo se usa para las columnas presentes, en lugar de almacenar valores “nulos” cuando no hay un valor para el atributo en cuestión, algo que se hace especialmente notable a la hora de almacenar grandes cantidades de registros.

Ilustración 11 . Representación de columnas

7b976c48...	name: Bill Watterson	state: DC	birth_date: 1953
7c8f33e2...	name: Howard Tayler	state: UT	birth_date: 1968
7d2a3630...	name: Randall Monroe	state: PA	
7da30d76...	name: Dave Kellett	state: CA	

Fuente: *Datastax.*

DIFERENCIA ENTRE LAS BASES DE DATOS *SQL* Y *CASSANDRA*

Las Bases de Datos no relacionales disponibles en el mercado hoy en día presentan distintas ventajas y desventajas. *Cassandra* difiere de las Bases de Datos convencionales de las siguientes formas:

VELOCIDAD: la velocidad a la que se manejan las entradas en el sistema es notablemente mayor.

FUENTES: *Cassandra* recibe datos desde múltiples fuentes, mientras que las bases de datos relacionales suelen obtener información únicamente de una fuente o un grupo reducido de ellas.

MANEJO DE TIPOS DE DATOS: *Cassandra* maneja todo tipo de datos, mientras que las bases de datos relacionales manejan datos principalmente estructurados.

COMPLEJIDAD DE LAS TRANSACCIONES: *Cassandra* maneja transacciones simples, mientras que las Bases de Datos relacionales soportan transacciones mucho más complejas.

PUNTOS DE INTERRUPCIÓN: las Bases de Datos relacionales tienen puntos de interrupción singulares con detención del servicio, sin embargo, *Cassandra* no tiene puntos de interrupción y no detiene el servicio nunca.

VOLUMEN DE DATOS SOPORTADOS**:** el volumen de datos soportado por una Base de Datos relacional es moderado en comparación con *Cassandra.*

DESPLIEGUES**:** *Cassandra* soporta despliegues descentralizados, al contrario que las Bases de Datos relacionales.

LOCALIZACIÓN DE LOS DATOS**:** los datos en *Cassandra* son escritos en múltiples localizaciones de forma redundante para mejorar la persistencia; en Bases de Datos relacionales, la información se escribe de forma mayoritaria en una única localización.

ESCALABILIDAD**:** *Cassandra* soporta escalabilidad de lectura y escritura, mientras que las Bases de Datos relacionales sólo soportan la escalabilidad de lectura y a costa de sacrificios en la consistencia.

ORIENTACIÓN DEL DESPLIEGUE**:** *Cassandra* se despliega de forma horizontal y, en cambio, las Bases de Datos relacionales se descargan en escala vertical de abajo a arriba.

FUNCIONALIDADES Y BENEFICIOS CLAVE

Hay una serie de funcionalidades o ventajas que pueden marcar la diferencia para que *Cassandra* sea más apropiada para nuestra aplicación de Base de Datos:

Escalabilidad: *Cassandra* tiene un diseño sin máster donde todos los nodos son iguales, lo que proporciona simplicidad operacional y facilidad de escalabilidad.

Activo en todos los nodos: interacciones de escritura y lectura se pueden hacer desde cualquier nodo.

Escala lineal de rendimiento: la habilidad de añadir nodos sin parar el servicio con aumentos de rendimiento predecibles.

Disponibilidad completa: ofrece redundancia tanto de datos como de función de nodos, lo que elimina los puntos de fallo y proporciona un tiempo constante de servicio.

Detección y recuperación de fallos transparente: los nodos que fallan pueden ser fácilmente recuperados o reemplazados.

Modelo de datos flexible y dinámico: soporta tipos de datos modernos con alta velocidad de lectura y escritura.

Fuerte protección de datos: un *log* de *commits* asegura que no se pierda datos con seguridad incorporada para realizar *backups* y restauraciones que mantiene los datos a salvo.

Consistencia de datos ajustable: ofrece soporte para mantener una consistencia de datos tanto fuerte como eventual a través de un clúster muy distribuido.

Replicación de múltiples centros de datos: cruza centros de datos a través de múltiples geografías y soporta la escritura y lectura a través múltiples nubes.

Compresión de datos: hasta un 80% de compresión sin coste al rendimiento

Cassandra Query Language o ***CQL***: un lenguaje similar a *SQL* que hace que la migración de una Base de Datos relacional a una Base de Datos *Cassandra* sea menos abrupta, ya que tiene muchas de las funcionalidades de *SQL*.

Estas son las ventajas. Por supuesto, hay que tener en cuenta que las Bases de Datos no relacionales realizan compromisos respecto a las Bases de Datos relaciones.

Existen desventajas en el uso de *Cassandra*, como son:

No existe integridad referencial y no hay concepto de conexiones *JOIN.*

Las opciones para recuperar datos a través de *queries* son muy limitadas.

La ordenación de datos es una decisión de diseño, y se realiza a través de métodos predefinidos; pueden recuperarse datos en el mismo orden, pero no existen *ORDER BY* o *GROUP BY*.

La desnormalización es buena. Se desnormalizan los datos para tener redundancia (algo que las reglas de *Codd* no aprueban) y los datos se guardan en el orden que se recuperan.

Diseño de Bases de Datos diferente: en las Bases de Datos relacionales se modela antes de crear las *queries*, al contrario que en *Cassandra*, donde han de tenerse en cuenta cuáles van a ser las consultas más comunes y modelar la Base de Datos alrededor de esas consultas.

TIPOS DE DATOS DE *CASSANDRA*

MODELO DE DATOS

Cassandra es una Base de Datos columnar que utiliza un modelo altamente desnormalizado diseñado para capturar y consultar datos de forma activa, a pesar de

que los objetos de *Cassandra* se parecen a las Bases de Datos relacionales (tablas, *primary keys*, índices, etc…).

Tabla 4. Apariencia de Base de Datos de *Cassandra*

Table Name: Products_Inventory			
Row Key	**ts**	**Column Family Products**	**Column Family Inventory**
P001	t1	Products: Classes = "TV"	
	t2	Products: Title = "SONY 55 inch 4K OLED Smart Networked TV"	
	t3	Products: Descriptions = "TBD"	
	t4	Products: Price = "24999"	
	t5		Inventory: Quantity = "10"
	t6		Inventory: Place = "1A"
P002	t7	Products: Classes = "Laptop"	
	t8	Products: Title = "ACER SF514 14-inch laptop"	
	t9	Products: Descriptions = "TBD"	
	t10	Products: Price = "31000"	
	t11		Inventory: Quantity = "20"
	t12		Inventory: Place = "2A"
P003	t13	Products: Classes = "Mobile phone"	
	t14	Products: Title = "ZenFone 5Z"	
	t15	Products: Descriptions = "TBD"	
	t16	Products: Price = "5000"	
	t17		Inventory: Quantity = "8"
	t18		Inventory: Place = "2B"

Fuente: Chen, JK y Lee WZ (2019: 7)

Las técnicas de modelado de datos de *Cassandra* difieren de las tradicionales debido a enfoques que no son apropiados para *Cassandra*, como los atributos entidad-relación.

Al contrario que las Bases de Datos convencionales, que penalizan por tener una gran cantidad de columnas en una tabla*, Cassandra* está diseñado para funcionar con tablas que contienen miles de columnas.

OBJETOS

Existen varios tipos básicos de objetos en *Cassandra*:

Keyspace: un contenedor de tablas de datos e índices, análogo a una Base de Datos en Base de Datos relacionales, es el nivel en el que se defina la replicación.

Table: las tablas son semejantes a sus homólogos en Bases de Datos *SQL*, pero capaces de proporcionar una gran capacidad de almacenamiento y alta velocidad de inserción de registros y lectura a nivel de columna.

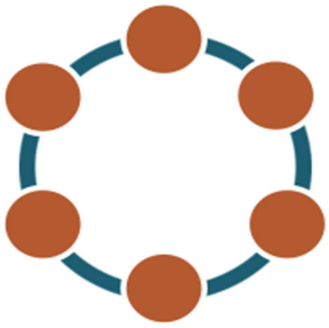
Primary Key: se emplean para identificar el registro dentro de la tabla. También distribuye el registro a lo largo del clúster.

Index: es similar al índice en las tablas relacionales.

ARQUITECTURA

La arquitectura de *Cassandra* es responsable por la capacidad de escalar, funcionar y ofrecer servicio constante. En lugar de utilizar la estructura maestro-esclavo clásica o un diseño compartido más difícil de mantener, *Cassandra* emplea un diseño de anillo en su arquitectura, de forma que es más sencillo de montar y mantener.

Ilustración 12. Arquitectura de *Cassandra*



Fuente: *Datastax Academy*

En *Cassandra*, todos los nodos tienen los mismos roles, no hay concepto de nodo maestro, con todos los nodos comunicándose entre ellos con un protocolo distribuido escalable llamado *gossip*.

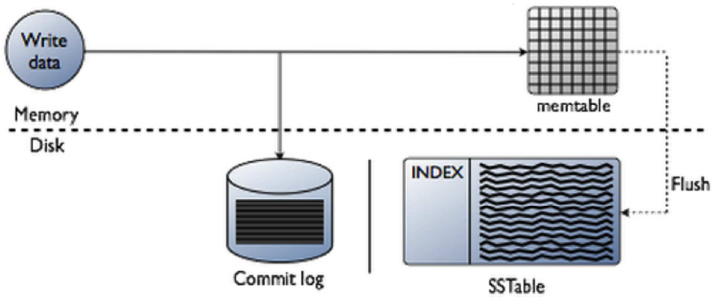
Cassandra tiene una arquitectura orientada a escalar, lo que significa que es capaz de mantener grandes cantidades de datos de muchos usuarios concurrentes u operaciones por segundo, incluso a través de múltiples centros de datos y con la sencillez de simplemente añadir un nodo nuevo en caso de necesitar más capacidad sin necesitar parar el servicio.

La arquitectura de *Cassandra* también significa que no existe un punto de error único; si un nodo cae, el sistema sigue funcionando.

ESCRITURA Y LECTURA

Cassandra busca una escritura de alta duración y rendimiento. Los datos escritos en un nodo de *Cassandra* son primero grabados en un *commit log* en disco y, luego, pasados a una estructura en memoria llamada *memtable*. Cuando el tamaño de una *memtable* supera el límite configurable, los datos son escritos en un archivo inmutable en disco, llamado *SSTable*. Que el *buffering* escriba en memoria de esta forma permite que todas las operaciones de escritura sean siempre operaciones totalmente secuenciales, con muchos megabytes de disco ocupados por operaciones de *I/O* al mismo tiempo en lugar de una operación detrás de otra en un largo periodo de tiempo. Esta arquitectura le da a *Cassandra* alto rendimiento en escritura.

Ilustración 13. Escritura de *Cassandra. SStable*



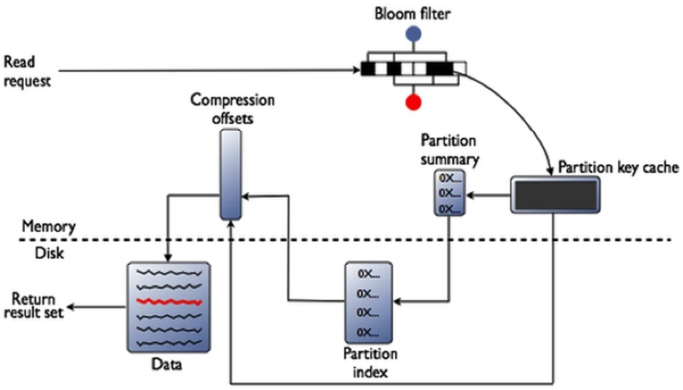
Fuente: *Datastax Academy.*

Muchas *SSTables* pueden existir a la vez para una única tabla lógica de *Cassandra.* Un proceso llamado *compaction* o compactación se realiza de forma periódica, uniendo múltiples tablas *SSTable* en una para un acceso de lectura más rápido.

Leer datos de *Cassandra* supone un número de procesos que pueden incluir varios cachés de memoria y otros mecanismos diseñados para producir una rápida respuesta de lectura.

Para una petición de lectura*, Cassandra* consulta una estructura de datos en memoria llamada *Bloom filter,* que comprueba la probabilidad de que una *SSTable* tenga los datos necesarios. El *Bloom filter* puede determinar de forma rápida si el archivo dispone de esta información. Si la respuesta es un "sí" tentativo, *Cassandra* consulta otra capa de caché en memoria y, entonces, toma los datos comprimidos en el disco. Si es "no", *Cassandra* pasa a la siguiente tabla.

Ilustración 14. Lectura de *Cassandra*



Fuente: *Datastax Academy.*

DISTRIBUCIÓN Y REPLICACIÓN DE DATOS

Pese a que ya se tiene una compresión de las operaciones de lectura y escritura en el contexto de un único nodo, la actividad de *I/O* del clúster es bastante más sofisticada ya que la Base de Datos está distribuida y tiene una arquitectura que carece de maestro. Los dos conceptos que

más afectan a la Base de Datos en las actividades de escritura y lectura son la distribución y la replicación.

DISTRIBUCIÓN DE DATOS AUTOMÁTICA

Algunas Bases de Datos *NoSQL,* así como las Bases de Datos relacionales, requieren métodos manuales desarrollados para distribuir los datos a través del clúster de máquinas. Estas técnicas suelen conocerse como *sharing*. *Sharing* es una técnica antigua pero con éxito en la industria, aunque está limitada por dificultades operacionales y su diseño inherente. En contraste, *Cassandra* distribuye y mantiene datos de forma automática a lo largo del clúster, liberando a los desarrolladores de la tarea de crear estas funcionalidades.

Cassandra tiene un componente interno llamado *Partitioner*, que determina cómo se distribuyen los datos en los nodos que conforman el clúster de la Base de Datos. En resumen, es un mecanismo de *hash* que toma la *primary key* de un registro de una tabla, genera un *token* numérico para él, y se lo asigna a uno de los nodos del clúster de forma que sea predecible y constante.

Partitioner es configurable, por defecto aleatoriza los datos a lo largo del clúster de forma que se asegura una distribución equitativa. *Cassandra* también mantiene el balance de datos en el clúster de forma automática, incluso cuando se eliminan nodos del clúster o se añaden al clúster.

BASES DE REPLICACIÓN

El clúster de *Cassandra* puede tener uno o más *keyspaces***,** que son análogos a las Bases de Datos de *SQL*. La replicación se configura a nivel de *keyspace*, lo que permite distintos modelos de replicación para distintos *keyspaces.*

Cassandra es capaz de replicar datos a múltiples nodos en el clúster, lo que asegura la fiabilidad y acceso constante, así como operaciones de *I/O* más rápidas. El número total de copias que se replican se denomina *replication factor* o factor de replicación. Un factor de replicación de 1 quiere decir que sólo hay una copia de cada registro en el clúster; uno de 3, significa que hay tres copias de cada elemento en el clúster.

Una vez un *keyspace* y su replicación han sido creados, *Cassandra* mantiene de forma automática esa replicación, incluso cuando los nodos son añadidos, quitados o fallan.

SOPORTE DE NUBE Y CENTROS DE DATOS MÚLTIPLES

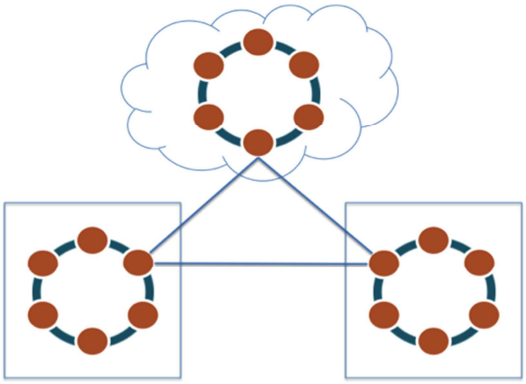
Una forma de asegurar acceso continuo sin caídas, y lectura y escritura rápida, incluso en regiones localizadas, es tener replicación en múltiples centros de datos y nubes.

Cassandra ofrece la posibilidad de replicar los datos en múltiples localizaciones geográficas, lo que permite a los usuarios escribir y leer en el centro que ellos elijan o

sea más cercano a ellos, así como sincronizar de forma automática los datos entre todas las localizaciones.

Los centros de datos pueden configurarse en función del número de copias que se desea guardar en cada uno de ellos.

Ilustración 15. Centros de datos de *Cassandra*



Fuente: *Datastax Academy*

CASOS DE USO

Cassandra se presenta como una Base de Datos genérica no relacional, sin hacer hincapié en ninguna funcionalidad específica u orientarse a un determinado tipo de aplicación, al contrario que otras Bases de Datos *NoSQL* como *Redis***,** que se ofrecen como sistemas de Bases de Datos de soporte o secundarios y que funcionan como una Base de Datos en *RAM* que ofrece mayor velocidad. Pese a esto, hay una serie de casos en los que *Cassandra* destaca:

Aplicaciones de Internet de las Cosas (*IoT*): *Cassandra* es perfecta para esta aplicación ya que tiene la capacidad de procesar muchos datos que proceden de diversos mecanismos y sensores en posiciones distintas.

Catálogos de productos y aplicaciones orientadas a ventas, hacer seguimiento del comportamiento del usuario dentro del portal web, interacción con los productos, etc.

Mensajería: por sus características y velocidad, de forma similar a aplicaciones en *IoT*, *Cassandra* tiene capacidad para gestionar los datos de aplicaciones de mensajería sin problema.

Analiticas de Social Media y motores de recomendaciones: una aplicación común en Big Data es el rastreo del comportamiento de usuarios para determinar los mejores elementos a recomendar al usuario.

Aplicaciones basadas en series de tiempo: gracias a la habilidad de escribir de forma rápida, su diseño de columnas extensas y la habilidad de leer solo las columnas

necesarias para satisfacer la consulta, se presenta como una alternativa idónea para aplicaciones basadas en series de tiempo.

BASES DE DATOS DE GRAFOS

INTRODUCCIÓN

De forma similar a otras Bases de Datos no relacionales, *Neo4J* surge de la necesidad de una Base de Datos que se adapte mejor a ciertos tipos de aplicaciones. *Neo4J* es una Base de Datos de grafos, diseñada para interactuar con datos altamente relacionados, como los que se pueden encontrar en un grafo social.

Uno de los problemas de las Bases de Datos relacionales es la desconexión entre cómo las Bases de Datos relacionales representan entidades y relaciones en contraste con cómo la programación orientada a objetos representa los objetos, incluyendo problemas de herencia y polimorfismo. Estas complicaciones son la razón de la aparición del *Object Relational Mapping (ORM);* herramientas como *Hibernate* e *iBATIS*, las cuales acercan el modelo de una Base de Datos relacional y modelo de programación orientada a objetos, *Neo4J* tiene un modelo de programación que resulta mucho más natural para programaciones orientadas a objetos.

FUNCIONAMIENTO

Neo4J usa grafos para representar datos y las relaciones entre ellos. Un grafo se define con representación gráfica que consta de vértices. Dentro de estas representaciones gráficas existen varios tipos:

Grafos no direccionados: los nodos y las relaciones son intercambiables, su relación puede ser interpretada de cualquier forma. Por ejemplo, relaciones en Redes Sociales como *Facebook* (amistades) entrarían en este tipo.

Grafos direccionados: los nodos y las relaciones no son bidireccionales por defecto. Las relaciones en *Twitter,* por ejemplo, son de este tipo. Un usuario puede seguir a un segundo sin que éste le siga a él.

Grafos con peso: en este tipo de grafos la relaciones entre nodos tienen evaluaciones numéricas, lo cual permite que se realicen operaciones.

Grafos con etiquetas: estos grafos tienen etiquetas incorporadas que pueden definir los vértices y las relaciones entre ellos. En *Facebook* se pueden tener nodos definidos como “amigo” o “compañero de trabajo” y relaciones como “amigo de” o “compañero de”.

Grafos con propiedad: este es un grafo con peso y etiquetas donde se pueden asignar propiedades a ambos nodos de la relación, por ejemplo: nombre, edad o residencia. Este tipo es el más complejo.

Neo4J usa grafos con propiedad para extraer valor de los datos de cualquier compañía con gran rendimiento y agilidad, además de hacerlo de forma escalable y flexible.

RENDIMIENTO

Neo4J ofrece un mejor rendimiento que las Bases de Datos relacionales *SQL*, a pesar de que las consultas aumentan de forma exponencial.

Para alcanzar este rendimiento, las Bases de Datos de grafos responden a peticiones actualizando el nodo y la relación de cada búsqueda y no el grafo completo. Esto optimiza el proceso.

RESPONSIVIDAD

Neo4J afirma que, para superar la capacidad de manejo de datos, serían necesarios 34 billones de nodos. Es importante remarcar que éstos no son los nodos de un clúster, sino la definición de nodo de *Neo4J*; 34 billones de relaciones entre estos nodos, 68 billones de propiedades y 32000 tipos de relaciones.

FLEXIBILIDAD Y ESCALABILIDAD

Pese a que *Neo4J* se presenta como una Base de Datos con alta escalabilidad, se encuentran ciertas limitaciones en su arquitectura. La Base de Datos intenta encajar el grafo en su complejidad en una sola máquina; de esta forma, los datos no se guardan en un clúster como se ha visto en otras Bases de Datos distribuidas, sino que todos los vértices y filos se guardan en una única máquina, aunque esto no supone que todas las operaciones de lectura y escritura tengan que hacerse en la misma máquina.

Neo4J se diseña en una arquitectura de maestro-esclavo. Pese a que siempre hay un nodo encargándose de las escrituras (por motivos de rendimiento sólo debería realizar escrituras), existe la posibilidad de añadir más nodos con réplicas de solo lectura que se pueden utilizar para sacar los datos. De esta forma, la arquitectura de escritura es vertical, como en la arquitectura maestro-esclavo, y horizontalmente para lectura como en una arquitectura de clúster distribuido.

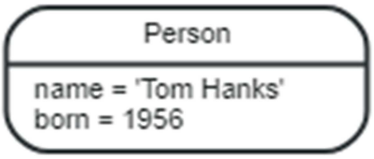
Es importante tener puntos de partición, que indican donde comienza las transversales de diferentes vértices en cada nodo. Esto, en conjunto con la escalabilidad de lectura horizontal, permite aplicar las transversales a subgrafos localizados en distintos nodos del clúster.

CONCEPTOS DE LAS BASES DE DATOS DE GRAFOS

NODOS

Los nodos en *Neo4J* son usados para representar entidades. El grafo más simple posible es un nodo único.

Ilustración 16. Nodo único (grafo)



Fuente: Documentación Oficial *Neo4J*

LABELS

Las etiquetas se utilizan para darle forma al dominio mediante la agrupación de nodos en conjuntos donde todos los nodos tienen la misma etiqueta. De esta forma, se pueden realizar operaciones únicamente a los nodos que tienen una etiqueta determinada.

Como las etiquetas pueden añadirse y quitarse durante ejecución, también pueden usarse para marcar estados temporales de los nodos. Un nodo puede tener desde “0” hasta las etiquetas que sea necesario.

RELACIONES

Una relación conecta dos nodos. Las relaciones organizan los nodos en estructuras, permiten que un grafo se parezca a una lista, un árbol, un mapa o una entidad compuesta, y pueden combinarse para crear estructuras más complejas.

TIPOS DE RELACIONES

Las relaciones sólo pueden tener un tipo de relación, y siempre tiene una dirección. No obstante, no es necesario añadir relaciones en direcciones opuestas si no es necesario para describir el caso de uso.

PROPIEDADES

Las propiedades son pares de nombre-valor que se usan para añadir cualidades a nodos en las relaciones. La parte de valor de la propiedad puede contener diversos tipos de datos como nombre, *string* o *boolean*.

CAMINOS Y TRANSVERSALES

Una transversal es la forma en la que se consulta un grafo de modo que conteste a la pregunta.

Recorrer un grafo supone visitar los nodos siguiendo las relaciones de acuerdo con las reglas. En la mayoría de los casos, sólo un subconjunto de grafo se visita. El resultado del recorrido devuelve el camino.

La longitud del camino equivale a la cantidad de relaciones recorridas. La longitud mínima es de “0”, lo que quiere decir que el primer nodo es la respuesta.

ESQUEMAS

Un esquema en *Neo4J* se refiere a los índices que contiene.

Neo4J puede describirse como de esquema opcional, lo que significa que no es necesario crear grupos de índices. Es posible crear nodos, relaciones y propiedades sin definir esquemas de antemano. Índices y grupos pueden introducirse en cualquier momento, para ganar rendimiento o beneficios de modelado.

> **Índices:** los índices se utilizan para mejorar el rendimiento. Se trabaja con ellos en *Cypher*.
>
> **Grupos:** se emplean para asegurar que los datos se adhieran a las reglas del domino; por ejemplo, que el valor de una propiedad ha de ser único entre los nodos que tienen una etiqueta en concreto.

REGLAS DE NOMBRES Y RECOMENDACIONES

Las etiquetas de nodos, tipos de relaciones y propiedades son *case sensitive*. Existen convenciones a la hora de establecer nombres para cada tipo de elemento.

Tabla 5. Normas de estilo en *Neo4J*

Graph entity	Recommended style	Example
Node label	Camel case, beginning with an upper-case character	`:VehicleOwner` rather than `:vehice_owner`
Relationship type	Upper case, using underscore to separate words	`:OWNS_VEHICLE` rather than `:ownsVehicle`
Property	Lower camel case, beginning with a lower-case character	`firstName` rather than `first_name`

Fuente: Documentación Oficial *Neo4J*

CYPHER

Cypher es el lenguaje de consulta de *Neo4J.*

PATRONES

Los grafos de propiedades de *Neo4J* se componen de nodos y relaciones, ambos pueden tener propiedades. Los nodos representan entidades, y las relaciones conectan pares de nodos.

Los nodos y las relaciones pueden considerarse como bloques de construcción de bajo nivel. La verdadera fuerza del grafo de propiedades reside en su habilidad para codificar patrones de nodos y relaciones conectados.

Cypher está fuertemente basado en patrones. Especialmente, patrones usados para emparejar estructuras de grafos. Cuando una estructura que encaja con otra se crea o se encuentra, *Neo4J* puede usarlo para procesos posteriores.

Un patrón sencillo, que tiene una única relación, conecta dos nodos. Patrones complejos hacen uso de múltiples relaciones, puede expresar conceptos complejos arbitrariamente y soporta una variedad de casos de uso.

Sintaxis de nodo

Cypher paréntesis, “()”, para representar un nodo. La forma más simple, “()”, representa un nodo anónimo sin caracterizar. Si se quiere referenciar ese nodo en otro punto, se puede usar una variable que haya dentro del nodo. Una variable está restringida a una sola declaración, ya que pueden tener un significado distinto, o ninguno, en otra declaración.

Sintaxis de relaciones

Cypher usa un par de guiones para representar una relación sin direccion, “--”. Las relaciones direccionales tienen una cabeza de flecha en uno de los extremos, “<--”, “-->”. Una expresion entre *brackets* “[...]” puede usarse para

añadir detalles. Esto puede incluir variables, propiedades, y tipos de información.

La sintaxis y semántica que se encuentra entre los *brackets* de una relación son muy similares a la que se encuentra entre los paréntesis de un nodo. Una variable puede definirse para utilizarse en cualquier punto de la declaración.

Sintaxis de patrones

Combinando la sintaxis de nodos y relaciones, se pueden expresar los patrones. De la misma forma en que se representan las propiedades de las relaciones como listas de **llave/valor** envueltas en "{}", las propiedades pueden usarse para guardar información y/o restringir patrones.

Variables de patrones

Para incrementar modularidad y reducir la repetición, *Cypher* permite asignar patrones a variables. Esto permite a los caminos equivalentes ser analizados, usados en otra expresión, etc.

Existen funciones para acceder a los detalles de*l path: nodes (path), relationships (path) y length (path).*

Cláusulas

Las declaraciones de *Cypher* suelen tener múltiples cláusulas; cada una realiza una tarea específica, por ejemplo:

Crear y emparejar patrones en el grafo.

Filtrar, proyectar, organizar o paginar resultados.

Componer declaraciones parciales.

Combinando cláusulas de *Cypher*, se puede componer declaraciones más complejas que expresar lo que se quiere consultar o crear.

PATRONES EN PRÁCTICA

Creación de datos

Para añadir datos, se emplean los patrones que ya se conocen. Al aportar patrones, se puede especificar qué estructura de grafo, etiquetas, y propiedades se desea hacer parte del nuevo grafo.

La cláusula más sencilla para crear grafos es *CREATE*, la cual creará los patrones que sean especificados. Por ejemplo: *CREATE (:MOVIE {TITLE:"THE MATRIX",RELEASE:1997}).*

Si se ejecuta esta declaración, *Cypher* devolverá el número de cambios; en este caso, sería añadir un nodo, una etiqueta y dos propiedades, siendo el nodo la totalidad del elemento que se ha creado, la etiqueta será *MOVIE* y las propiedades serán *TITLE* y *RELEASE*.

En caso de que se quiera devolver lo que se acaba de crear, se añade la cláusula *RETURN*, que hace referencia a la variable que se ha asignado a los elementos que se tienen en patrones; simplemente se añade la declaración de *RETURN* después de la declaración de *CREATE* de la siguiente forma:

CREATE…

RETURN "variable"

En esta ocasion, *Cypher* también devolverá que se ha creado un registro.

Si se quiere crear más de un elemento se dispone de dos opciones: bien se puede utilizar varias declaraciones de *CREATE* o utilizar una única declaración y separar los elementos por comas.

Se puede utilizar la declaración *CREATE* para crear nodos ya relacionados, utilizando la nomenclatura ya vista anteriormente de flechas, se añade la relación entre los guiones que forman la flecha, de la siguiente forma:

NODO-RELACIÓN->NODO

La relación tendría una estructura como la siguiente:

-[r:ACTED_IN { roles: ["Forrest"]}]->

Donde "r" sería la relación, que en este caso es "actuó en". Para tener una imagen más completa de la relación, se añadirán los nodos que está uniendo:

(a:Person { name:"Tom Hanks"})-[r:ACTED_IN { roles: ["Forrest"]}]->(m:Movie {title:"Forrest Gump"})

En la mayoría de los casos, se suele conectar estructuras ya existentes, lo cual requiere de conocimientos sobre cómo encontrar patrones ya existentes en el grafo.

Emparejamiento de patrones

La búsqueda de patrones que encajan se realiza con la declaración *MATCH*, a la que se le pasa alguno de los patrones de los que ya han sido utilizados para describir lo que se está buscando, de forma similar a como funcionaría una consulta.

Una declaración *MATCH* buscará por los patrones que se especifiquen y devolverá una línea por patrón que encaje.

Siguiendo con los ejemplos anteriores, se pueden buscar los nodos que lleven la etiqueta *MOVIE*. La declaración quedaría de la siguiente forma:

MATCH (m:MOVIE)

RETURN m

También puede realizarse una declaración *MATCH* más precisa:

MARCH (m:MOVIE {TITLE:"THE MATRIX"})

RETURN m

En estos casos, sólo se está aportando la información suficiente para encontrar los nodos, no son necesarias todas las propiedades. En la mayoría de los casos, existen propiedades clave como *in ID* que ayudan a realizar búsquedas precisas.

La verdadera utilidad de esta declaración radica en la búsqueda de conexiones; realizando búsquedas de relaciones se pueden encontrar las conexiones de un tipo concreto que tiene un nodo con otros.

Es importante saber que es posible acceder a los nodos que ya se conocen desde cualquier punto a través de la notación *dot* con *IDENTIFICADOR.PROPIEDAD*.

Acoplamiento de estructuras

Mediante el uso de las estructuras *CREATE y MATCH,* se puede acoplar estructuras a un grafo. Esto permite extender un grafo con nueva información. Primero se declara *MATCH* con las conexiones existentes, después, se declara *CREATE* para acoplar los nuevos nodos con relaciones. Un ejemplo de esto sería:

MATCH (m:MOVIE {TITLE:"THE MATRIX"})

CREATE (p:PERSON {NAME:"KEANU REEVES"})

CRETE (p)-[r:ACTED_IN{ROLES:['NEO']}]->(m)

RETURN p,r,m

Es importante recordar que se pueden asignar variables tanto a los nodos como a las relaciones para usarlos posteriormente, independientemente de si han sido (o no) emparejados.

Es recomendable que, pese a poder incluir en una única declaración *CREATE* el nodo y la relación, para mayor fiabilidad se dividan en una declaración *CREATE* cada una.

Otro elemento que puede dificultar esta tarea es que, al combinar los elementos *MATCH* y *CREATE,* se obliga a los elementos *CREATE* que vengan seguidos de la declaración a ejecutarse una vez por cada registro. En muchos casos, este es el comportamiento que se está buscando, pero si no es el caso, es importante mover las declaraciones de *CREATE* antes de las declaraciones

de *MATCH*. Esto también puede solucionarse mediante el uso de la declaración *MERGE* que se presentará a continuación.

Completar patrones

Siempre que se obtengan datos de sistemas externos o no se esté seguro de si la información ya existe en el grafo de que se dispone, es necesario expresar una operación de actualización repetible que no altere los elementos ya existentes. En *Cypher*, *MERGE* tiene esta función.

MERGE actúa como una combinación de *MATCH* y *CREATE* que comprueba los datos ya existentes antes de crear nuevos. Con *MERGE* se define un patrón para ser creado o encontrado. Normalmente, al igual que con *MATCH,* sólo se incluye la propiedad clave por la que buscar en el patrón del que se dispone, y, al igual que las declaraciones que se han visto anteriormente, permite añadir propiedades con *ON CREATE*.

```
MERGE (m:MOVIE {TITLE:"MATRIX"})
ON CREATE SET m.RELEASED = 1999
RETURN m
```

En este caso se recibirá bien el nodo ya existente o, en caso de que no existiese, un nodo recién creado.

Una cláusula *MERGE* sin variables previamente asignadas creará o emparejará el patrón al completo. No produce una mezcla parcial de emparejamiento y creación en el patrón. Para lograr este resultado de mezcla, es necesario definir variables para las partes que no deberían verse afectadas.

En resumen, *MERGE* es una herramienta para asegurar que no se crea información o estructuras duplicadas. Esto supone el coste de tener que revisar los datos existentes antes. Especialmente en grafos de gran tamaño pueden ser costoso escanear una gran colección de datos con la misma etiqueta en busca de una propiedad en concreto. Se puede remediar en cierta forma con la creación de índices adicionales o condiciones, que se verán más adelante. Cuando existe certeza de que los datos que se están introduciendo no forman parte de la Base de Datos, es mejor usar *CREATE* en lugar de *MERGE*.

MERGE también puede garantizar que no se generen relaciones duplicadas. Para esto, es necesario proporcionar ambos nodos de la relación.

OBTENIENDO LOS RECURSOS CORRECTOS

Filtrado de resultado

La cláusula con la que se realizan los filtrados de información, similar a *SQL*, es *WHERE*. Esta cláusula permite utilizar expresiones *booleanas* combinadas con los operadores *AND*, *OR*, *XOR*, y *NOT*. A estas expresiones *booleanas* se las determina como *predicados* y las simples son las comparaciones, en concreto las de igualdad. Por ejemplo:

```
MATCH (m: MOVIE)
WHERE m.TITLE = "THE MATRIX"
RETURN m
```

Esta consulta podría simplificarse incluyendo la condición en el patrón:

```
MATCH (m: MOVIE {TITLE: "THE MATRIX"})
RETURN m
```

Otras opciones son las comparaciones numéricas, emparejando expresiones regulares y comprobando la existencia de valores dentro de una lista:

```
MATCH (p:PERSON)-[r:ACTED_IN]->(m:MOVIE)
WHERE p.NAME =~ "K.+" OR m.RELEASED > 2000 OR "NEO" IN
r.ROLES
RETURN p,r,m
```

Los patrones pueden usarse como *predicados. MATCH* expande el número y forma de patrones emparejados, un patrón con *predicado* restringe la colección de datos actual del resultado. Sólo permite los caminos o *paths* que satisfagan el patrón dado. Como es de esperar, el uso de *NOT* sólo permite los caminos que no cumplan el patrón especificado.

```
MATCH (p:PERSON)-[:ACTED_IN]->(m)
WHERE NOT (p)-[:DIRECTED]->()
RETURN p,m
```

En el ejemplo anterior se obtendrán los actores que hayan actuado pero nunca hayan dirigido una película.

Existen formas más avanzadas de filtrar resultados, como las *list predicates*, que se expondrán más adelante.

Devolver resultados

Ya se ha mostrado que la cláusula *RETURN* puede devolver nodos, relaciones y caminos a través de las variables. No obstante, esta cláusula puede devolver cualquier expresión. Para aprovechar esto, primero se debe asimilar qué es una expresión en *Cypher*.

Las expresiones más sencillas son los valores literales, como números, *strings*, *arrays* y los mapas. Se puede acceder a las propiedades individuales de un nodo, relaciones o mapas con *dot syntax,* por ejemplo, *n.NAME*. Elementos individuales o partes de *arrays* pueden extraerse con subíndices como *NAME[0]* o *MOVIES[1..-1].* Cada evaluación de función también es una expresión, como es el caso de *length (array)*.

Los *predicados* que se usan en las cláusulas *WHERE* cuentan como expresiones *booleanas*.

Las expresiones simples pueden ser compuestas y concatenadas para formar expresiones más complejas.

Por defecto, la expresión en sí misma se usará como etiqueta para la columna; en muchos casos, es mejor usar un alias para facilitar la lectura. Esto se hace forma similar a *SQL*:

expresión AS alias

El alias puede entonces ser usado para referirse a la columna.

También, de forma similar a *SQL***,** si se quiere mostrar los resultados únicos y no los duplicados, se puede utilizar *DISTINCT* después de *RETURN.*

Agregar información

En muchos casos, puede resultar útil agregar o agrupar los datos encontrados mientras se recorren los patrones del grafo. En *Cypher*, la agregación se hace en el *RETURN* al computar los resultados finales. Muchas funciones comunes de agregación que se hallan en *SQL* pueden ser empleadas en *Cypher*, como es el caso de *count, sum, avg, min,* y *max*, entre otras.

Es importante tener en cuenta que, durante la agregación, los valores *null* se saltan.

La agregación afecta qué datos se mantienen visibles en la ordenación o consultas posteriores.

Ordenar y paginar

Es común usar *count(x)* para ordenar y paginar después de la agregación.

Para ordenar, se utiliza la expresión *ORDER BY expresión [ASC|DESC],* no se puede ordenar por elementos que no estén en la cláusula *RETURN*.

La paginación se realiza con *SKIP {}* y *LIMIT{}*

Colección de valores agregados

Para coleccionar datos agregados se usa la función *collect()*, la cual colecciona todos los valores agregados en una lista. Es especialmente útil para las estructuras padre-hijo, en las cuales una entidad relevante se devuelve en una

columna con toda la información pertinente asociada en una lista creada con *collect()*. Lo que evita repetir la información del padre por cada registro hijo.

COMPOSICIÓN DE DECLARACIONES EXTENSAS

Unión

UNION [ALL] se usa para combinar los resultados de dos declaraciones que tienen la misma estructura de resultado.

With

En *Cypher* es posible encadenar fragmentos de declaración, de forma similar a una tubería de información. Cada fragmento trabaja con la salida del anterior, y su resultado es que alimenta al siguiente. Sólo las columnas declaradas con la cláusula *WITH* están disponibles para las siguientes partes de la consulta.

La cláusula *WITH* se usa para combinar las partes individuales y declara qué partes fluyen de una parte a la otra. Es similar a *RETURN*, salvo que la consulta no finaliza, sino que se prepara para la siguiente parte. Las expresiones que se pueden usar en *RETURN* también pueden ser utilizadas en *WITH,* la única diferencia es que todas las columnas deben llevar un alias.

DEFINICIÓN DE UN ESQUEMA

Índices

La principal razón para usar índices es marcar el punto inicial de un grafo. Una vez se dispone de este, puede recorrerse usando las estructuras del grafo para obtener alto rendimiento en la Base de Datos.

Los índices pueden añadirse en cualquier momento, pero añadir un índice cuando ya hay datos cargados en la Base de Datos supondrá una mayor carga de trabajo.

La expresión para crear un índice es *CREATE INDEX ON* y se emplea de la siguiente forma:

```
CREATE INDEX ON: ACTOR(NAME    )
```

Una vez creado el índice, no es necesario realizar ninguna acción adicional al consultar datos, ya que se usará el índice más apropiado de forma automática.

Índices compuestos son índices de múltiples propiedades para todos los nodos que tienen una determinada etiqueta.

Limitaciones

Las limitaciones o *constraints* se usan para asegurar que los datos se adhieren a las reglas del dominio. Por ejemplo, para definir qué valores dentro de los nodos que comparten una determinada etiqueta deben de ser únicos, como establecer que todos los nodos que tengan la etiqueta de *ACTOR* y la propiedad de *NAME* deben de tener un valor *NAME* único entre todos los nodos con etiqueta *ACTOR*.

La sintaxis de una limitación es la siguiente:

```
CREATE CONSTRAINT ON  (movie:MOVIE) ASSERT movie.title IS UNIQUE
```

Añadir una limitación supone añadir un índice a la propiedad sobre la que se esté estableciendo la limitación, en caso de que se elimine la limitación también se eliminará el índice. Si se desea mantener ese índice, tendrá que ser creada aparte.

Es posible añadir una limitación en una Base de Datos que ya esté poblada, pero será necesario que los datos que hay en ella ya cumplan la condición de la limitación.

Se puede comprobar qué limitaciones existen ya dentro de una Base de Datos con el proceso *db.constraints*.

IMPORTACIÓN DE DATOS

Cypher permite la importación de datos mediante el uso de ficheros *CSV*. Es recomendable que, antes de realizar una importación, se creen los índices y limitaciones que se quiera aplicar a los datos que van a ser importados. Esto también protege a la Base de Datos de importar datos que no sean válidos, ya que, si los datos del fichero *CSV* no cumplen las condiciones impuestas por las limitaciones, la importación fallará.

Es recomendable que los datos que se importan en fichero *CSV* tengan *Ids*. Aunque estos *IDs* sean una propiedad temporal, permiten relacionar más fácilmente los nodos cuando se suben varios *CSV*; al indexar los *Ids*, acciones como *MATCH* serán mucho más rápidas.

Al usar *MERGE* o *MATCH* en una importación, es importante asegurarse de que existe un índice o una limitación de valor único en la propiedad en la que se está realizando la operación, para asegurar que se realice de la forma más eficiente posible.

La carga de ficheros *CSV* se realiza con *LOAD CSV*. Un ejemplo sería:

```
LOAD CSV WITH HEADERS FROM “file:...” AS alias
```

CASOS DE USO

DETECCIÓN DE FRAUDE

Las redes de fraude tienen mecanismos que con el análisis de datos lineal no son detectables, pero, con el análisis escalable de múltiples relaciones entre datos, éstos pueden ser detectados.

RECOMENDACIONES EN TIEMPO REAL Y REDES SOCIALES

Las Bases de Datos de grafos permiten conectar personas con intereses. Con esta información, las empresas pueden ajustar sus servicios a sus clientes.

CENTRO DE GESTIÓN DE DATOS

Las Bases de Datos de grafos son perfectas para controlar Bases de Datos que crecen con rapidez.

SISTEMAS DE GESTIÓN DE DATOS MAESTRO

Permite generar una centralización de recursos que hace posible mantener protocolos y formatos constantes.

HDF ALMACENAMIENTO DE DATOS

INTRODUCCIÓN

HDFS es un software del proyecto *Apache Software Foundation* y un subproyecto del proyecto *Apache Hadoop. HDFS* son las siglas de *Hadoop Distributed File System*. *Hadoop* se presenta como una solución para almacenar grandes cantidades de datos, cantidades que se manejan con sistemas de Big Data, terabytes y petabytes. *Hadoop* utiliza *HDFS* como su sistema de almacenamiento.

HDFS permite conectar nodos (máquinas) que se encuentran dentro de un clúster en el que se distribuyen datos, lo cual permite acceder a los datos como si se tratase de un sistema de ficheros único, ejecutando los comandos y las aplicaciones con *MapReduce*, modelo de procesado que será presentado más adelante.

CARACTERÍSTICAS

HDFS tiene muchas similitudes con otros sistemas de ficheros distribuidos, pero las diferencias son la clave de su utilidad para sistemas de Big Data. *HDFS* tiene un modelo de *write-once-*

read-many, el cual relaja los requisitos del control de concurrencia, simplifica la coherencia de los datos y permite acceso de alto rendimiento.

Otro atributo único de *HDFS* es que es más eficiente al localizar la lógica de procesamiento cerca de los datos en lugar de mover los datos al espacio de la aplicación.

HDFS restringe rigurosamente la escritura de forma que sólo puede escribirse un elemento a la vez. Se adjuntan bytes al final del flujo, de forma que los flujos siempre se guardan en orden de llegada.

HDFS se presenta como un sistema de almacenado que promete las siguientes características:

Tolerancia a fallos con recuperaciones automáticas tras la rápida detección de errores.

Acceso a datos a través de *streaming MapReduce*.

Coherencia de datos simple y robusta.

Procesamientos lógicos cerca de los datos en lugar de acercar los datos al procesamiento lógico.

Portabilidad a lo largo de sistemas operativos y hardware heterogéneo.

Escalabilidad para guardar y procesar de forma fiable grandes cantidades de datos.

Economía, al distribuir los datos y procesos a lo largo de un clúster de máquinas.

Eficiencia al distribuir la lógica de procesos y los datos en paralelo en nodos donde se realiza el almacenamiento.

Fiabilidad, al mantener de forma automática múltiples copias de los datos y relanzando la lógica de procesos en caso de fallo.

ARQUITECTURA

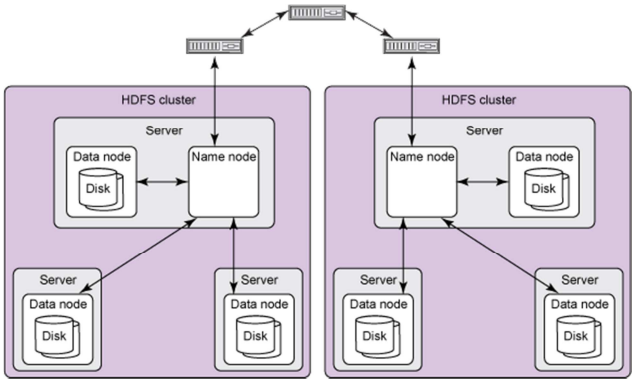
HDFS está compuesta de clústeres interconectados donde residen los directorios de archivos. Un clúster de *HDFS* consiste en un único nodo, llamado *NameNode*, que maneja el dominio de sistema de archivos y regula el acceso a archivos por el cliente. A demás de *NameNode*, están los nodos de datos o *DataNodes*, que guardan los datos en forma de bloques en los ficheros.

NAMENODE Y DATANODES

En *HDFS* el *NameNode* se encarga del manejo del sistema de archivos y las operaciones del dominio, como abrir, operar, cerrar y renombrar los directorios de archivos. Este nodo también maneja los bloques de datos y *DataNodes*, que se ocupan de las peticiones de lectura y escritura de los clientes de *HDFS*. Los *DataNodes* crean, borran y replican los bloques de datos de acuerdo con las instrucciones de *NameNode*.

Esta estructura de *NameNode* y *DataNode* se repite en cada clúster de *HDFS*, como se puede ver en la siguiente figura:

Ilustración 17. *NameNode* y *DateNode* en clústeres de *HDFS*



Fuente: *IBM Developer*

RELACIÓN ENTRE *NAMENODE* Y *DATANODES*

Los *NameNodes* y *DataNodes* son componentes de software diseñados para correr de forma independiente en máquinas distintas, en un ecosistema de sistemas operativos heterogéneos.

HDFS está diseñado usando Java; por lo tanto, cualquier máquina que soporte Java puede correr *HDFS*. Una instalación de clúster típica tiene una máquina dedicada para correr como *NameNode* y puede que en la misma máquina también se instale un *DataNode*, las demás máquinas son *DataNodes*.

> **Protocolo de comunicación:** todas las comunicaciones que se realizan en *HDFS* se construyen en el protocolo *TCP/IP*. Los clientes *HDFS* se conectan a un puerto abierto de protocolo de control de transmisión (*TCP*) en el *NameNode*. Tras esto, se comunican con el *NameNode* usando un protocolo basado en RPC (*Remote Procedure Call*). Los *DataNodes* se comunican con el *NameNode* usando un protocolo propietario basado en bloques.

Los *DataNodes* están en un bucle continuo en el que mandan al *NameNode* instrucciones. Un *NameNode* no puede conectarse directamente a los *DataNodes*; simplemente devuelve valores de funciones invocadas por un *DataNode*. Cada *DataNode* mantiene abiertos *sockets* del servidor para que el código cliente u otros *DataNodes* puedan leer o escribir datos. El puerto *host* para este *socket* del servidor es conocido por el *NameNode*, este provee información a los clientes.

En *NameNode* mantiene y administra los cambios en el dominio del sistema de archivos.

DOMINIO DEL SISTEMA DE ARCHIVOS

HDFS soporta una organización jerárquica tradicional en la que un usuario o aplicación puede crear directorios o guardar ficheros en él. La jerarquía del dominio es similar a las de otros sistemas de archivos; se puede crear, renombrar, mover y borrar ficheros.

HDFS soporta sistemas de archivos de terceros como *CloudStore* o *Amazon S3*.

INTERFACES

HDFS es accesible de varias formas, ya que presenta una API nativa de Java con una capa nativa de *C* para la API de Java, además de ser posible el uso de un navegador para acceder a los archivos en *HDFS*.

Existen numerosas aplicaciones que pueden usarse para acceder a *HDFS.* Algunas de estas son las siguientes:

FileSystem* (FS) *shell: interfaz de línea de comandos.

DFSAdmin: colección de comandos que pueden usarse para administrar un clúster de *HDFS.*

***File System Consistency Check* (FSCK):** subcomando de la aplicación *Hadoop*, útil para detectar inconsistencias en los ficheros, aunque no se puede utilizar para corregir las inconsistencias.

NameNodes* y *DataNodes: servidores web de *HDFS* que permiten a los administradores comprobar el estado del clúster.

MANEJO DE DATOS EN *HDFS*

REPLICACIÓN

HDFS replica los bloques de archivos de cara a la tolerancia a errores. Una aplicación puede especificar el número de réplicas de un archivo en el momento de creación, este número puede cambiarse en cualquier momento. El *NameNode* es el encargado de las decisiones relacionadas con la replicación de bloques.

Conciencia de *Rack*: normalmente, los clústeres de *HDFS* son colocados a lo largo de múltiples instalaciones. El tráfico de la red entre nodos a lo largo de la misma instalación es más eficiente que entre diversas instalaciones. Un *NameNode* trata de poner réplicas de un bloque en múltiples instalaciones para mejorar la tolerancia a errores. No obstante, *HDFS* permite a los administradores decidir a qué instalación pertenece un nodo. De esta forma, cada nodo conoce el *ID* de su *rack*.

HDFS utiliza un modelo de localización de réplicas inteligente para aumentar rendimiento y fiabilidad. La optimización de la localización de réplicas es una facultad

única de *HDFS* que otros sistemas de archivos no realizan, y es facilitada por el conocimiento del *ID* del *rack* por parte de los nodos que facilita un uso eficiente del ancho de banda.

Los ecosistemas grandes de *HDFS* normalmente operan a lo largo de múltiples instalaciones de máquinas. La comunicación entre dos nodos localizados en distintas instalaciones, como ya se ha expuesto anteriormente, es más lenta que si se encontrasen en la misma instalación. Por esto, el *NameNode* trata de optimizar las comunicaciones entre los *DataNodes*, identificando la localización de estos mediante el *ID* de su *rack*.

ORGANIZACIÓN

Una de las principales metas de *HDFS* es soportar archivos de gran tamaño. El tamaño típico de un bloque de *HDFS* es 64MB. Por tanto, cada archivo de *HDFS* consiste en uno o más bloques de 64MB. *HDFS* intenta colocar cada bloque en *DataNodes* separados.

Proceso de creación de archivos

La manipulación de archivos en *HDFS* es similar a los procesos utilizados en otros sistemas de archivos. Como *HDFS* es multi-máquina, aparece como un único disco, todo el código que manipula archivos en *HDFS* utiliza subclases del objeto *org.apache.hadoop.fs.FileSysten*.

Este es el aspecto del proceso creación de un archivo en *HDFS*:

```
byte[] fileData = retrieveFileDataFromSomewhere();
String filePath = retrieveFilePathStringFromSomewhere();
Configuration config = new Configuration();  // assumes to automatically load
        // hadoop-default.xml and hadoop-site.xml
org.apache.hadoop.fs.FileSystem hdfs
org.apache.hadoop.fs.FileSystem.get(config);
org.apache.hadoop.fs.Path path = new org.apache.hadoop.fs.Path(filePath);
org.apache.hadoop.fs.FSDataOutputStream outputStream = hdfs.create(path);
outputStream.write(fileData, 0, fileData.length);
```

Staging para realizar *Commit*

Cuando un cliente crea un archivo en *HDFS*, primero mete los datos en caché en un archivo local. Tras esto, se redirigen las escrituras subsecuentes a un archivo local temporal. Cuando el archivo local ha acumulado bastantes datos como para llenar un bloque *HDFS,* el cliente reporta esto al *NameNode*, el cual convierte el archivo a un *DataNode* permanente. El cliente, entonces, cierra el

archivo temporal y envía los datos restantes a un *DataNode* recién creado. El *NameNode* entonces hace *commit* del *DataNode* al disco.

Tubería de replicación

Cuando un cliente acumula un bloque completo de datos de usuario, extrae una lista de *DataNodes* que contiene la réplica de ese bloque del *NameNode*. El cliente envía todo el bloque de datos al primer *DataNode* especificado en la lista de replicación. Este proceso de tubería se repite hasta que el factor de replicación ha sido satisfecho.

MAPREDUCE

MapReduce es un modelo de programación e implementación asociada para procesar y generar colecciones de Big Data con un algoritmo paralelo, distribuido en un clúster.

MapReduce, así como *HDFS,* forman parte del ecosistema de *Hadoop*. Su funcionamiento consiste en dividir el procesamiento de datos en dos fases: la fase de *Map* y la fase de *Reduce*.

Map: es la primera parte del procesamiento, donde se especifica toda la lógica, reglas y código.

Reduce: es la segunda fase, en la que se especifican procesamientos ligeros como agregación.

En *HDFS Map* realiza lecturas del sistema y escrituras en un archivo local fuera de *HDFS*. Estos datos son leídos por la función *Reduce* y la salida es escrita por *HDFS*.

CAPÍTULO 3. PUESTA EN PRÁCTICA DEL TEMARIO

PRÁCTICA 1

ENUNCIADO

El enunciado de la práctica describe un proyecto de migración de una Base de Datos relacional convencional a tecnologías *NoSQL*. En este enunciado, se aportará la estructura inicial de la Base de Datos en un modelo entidad-relación y se solicitará el rediseño de esta estructura para funcionar con *MongoDB*.

A lo largo de la práctica, se realizarán distintos apartados -algunos de carácter más teórico- en los que se cuestionarán las decisiones de diseño y las razones para tomarlas.

Se añadirá un apartado opcional de cara a ofrecer mayor complejidad y opción a una mayor nota.

APARTADOS

La práctica estará dividida en varios apartados, algunos teóricos y otros prácticos con el fin de evaluar, afianzar y potenciar los conocimientos sobre estas tecnologías en un entorno más práctico.

Estos apartados son los siguientes:

1. Rediseño de la Base de Datos para ser usado en *MongoDB*.
2. Justificación del diseño escogido.
3. ¿Es el caso presentado óptimo para el uso de este tipo de tecnologías?
4. Usando los programas facilitados, genere una colección de datos para su inserción en la Base de Datos. Si lo considera necesario, modifique estos programas para que los datos generados estén en un formato más apto para su inserción en la Base de Datos.
5. ¿En qué casos seria óptimo usar una segunda Base de Datos *Redis*?
6. ¿Cómo diseñaría esta Base de Datos?
7. Realice e integre su diseño de Base de Datos *Redis* con el diseño de la Base de Datos *MongoDB*. (Opcional)

OBJETIVO DE LA PRÁCTICA

El objetivo de la práctica es ofrecer al alumno una comparación directa entre el funcionamiento de Bases de Datos relacionales, con las que ya están familiarizados, y las Bases de Datos no relacionales. Se busca que el alumno sea capaz de diseñar una estructura para una Base de Datos no relacional, además de comprender y razonar los motivos del diseño, así como

comprender el razonamiento que existe tras la elección de un modelo de Bases de Datos no relacionales frente a los modelos convencionales.

MATERIAL NECESARIO

Para la realización de esta práctica, será necesario:

NetBeans IDE 8.2 o superior

MongoDB Atlas 4.0 o superior

Oracle Java Standart Edition 8.0/*OpenJDK8.0* o superior

Redis 5.0 o superior

CRONOGRAMA

En el caso de esta práctica, ya que requiere de conocimientos previos en *MongoDB*, el enunciado debería facilitarse a los alumnos a la vez que se comienza a ver estos contenidos. Debido a la limitación temporal del curso, la parte de la práctica que hace referencia al último tema del bloque se deja como parte opcional.

PRÁCTICA 2

ENUNCIADO

En el enunciado de la segunda práctica, se expone una oferta de proyecto para la cual se tendrán que diseñar prototipos en distintas tecnologías de Bases de Datos no relacionales, de cara a presentar la oferta más completa.

A lo largo de la práctica se intercalan preguntas de carácter teórico con el objetivo de razonar, de forma crítica, las mejores soluciones y aplicaciones de las tecnologías en función de la situación dada.

APARTADOS

Tras la descripción del problema en el enunciado de la práctica, se presentarán una serie de cuestiones a resolver por el alumno, a saber:

1. ¿Se podría emplear una Base de Datos *NoSQL* en este escenario, de forma que reporte un beneficio tangible respecto al uso de una Base de Datos relacional más convencional?
2. Elabore un diseño de Base de Datos como solución al *Request For Proposal (RFP)* incluyendo un prototipo de esta solución en la tecnología que crea conveniente. Si cree que existen varias posibles soluciones, inclúyalas.

3. Justifique su elección y diseño.
4. ¿Sería posible el uso de *HDFS* como sistema de almacenamiento para la solución a esta *RFP*?, ¿Cómo se compararía esta alternativa a otras más convencionales?, ¿Cómo realizaría la implementación de esta herramienta?

OBJETIVO DE LA PRÁCTICA

El objetivo de esta práctica es la familiarización con diversas tecnologías, así como los distintos diseños y razonamientos que hay detrás de cada solución, y las características que pueden hacer que una determinada tecnología se adapte mejor a una determinada aplicación.

MATERIAL NECESARIO

Para el desarrollo de esta práctica será necesario:

Python 2.7 o superior.

Apache Cassandra 3.11 o superior.

Neo4J Enterprise 3.5.3 o superior

Apache Hadoop 3.1.2 o superior.

CRONOGRAMA

El enunciado de esta práctica debería facilitarse a los alumnos tan pronto como comience el segundo bloque del temario, haciendo posible que los alumnos realicen avances en las distintas tecnologías que se emplearán en la práctica, a medida que estos conocimientos son expuestos de forma teórica.

CAPÍTULO 4. CONCLUSIONES

OBJETIVOS CUMPLIDOS

En el desarrollo de este trabajo se busca un afianzamiento de competencias ya adquiridas con anterioridad, como el desarrollo de documentación o la búsqueda de información de forma eficiente. Nuevas competencias, como es el conocimiento del funcionamiento y estructura de diversos tipos de Bases de Datos no relacionales.

La capacidad de analizar y diseñar estructuras de datos coherentes y relacionarlas con el tipo de Base de Datos más adecuado para el caso de uso en cuestión, las diferencias entre los distintos tipos de Bases de Datos, casos de uso, aplicaciones, ventajas y desventajas.

El enfoque de las prácticas ha sido el de presentar escenarios realistas que, pese a su simplicidad, permitan un entorno inmersivo que aporte bases para la realización de razonamientos durante la toma de decisiones en el diseño de la Base de Datos.

DIFICULTADES A TENER EN CUENTA EN EL PROCESO

Al tratarse de un curso orientado a alumnos de Formación Profesional y de corta duración, es fácil toparse con carencias de conocimientos que pueden limitar la efectividad del material desarrollado. Un buen conocimiento de las Bases de Datos relacionales es necesario como punto de partida y comparación a la hora de intentar comprender las Bases de Datos no relacionales, los problemas que tratan de solucionar y los retos tecnológicos que presenta el Big Data en general.

POSIBLES MEJORAS Y TRABAJOS FUTUROS

Las características de las Bases de Datos no relacionales son muy extensas y, pese a que la comprensión de la tecnología Big Data y de las Bases de Datos relacionales más convencionales ayudan a entender el contexto y los razonamientos que existen detrás de estas tecnologías, el tiempo que se dedica a este curso sigue siendo muy corto como para obtener un nivel de comprensión funcional de este tipo de Bases de Datos. La mejora más obvia que existe a este trabajo, por lo tanto, es la ampliación del tiempo del curso.

De contar con más tiempo, el enfoque del curso debería contener más peso práctico, ya que esa suele ser la metodología que permite una mejor retención del conocimiento, además de aportar la posibilidad de enfrentarse con los problemas y bloqueos técnicos más comunes a la hora de poner en práctica estas tecnologías. La limitación del tiempo supone que haya que poner un mayor énfasis en la parte teórica, pues sería inviable comenzar con materia práctica en un área que es desconocida.

BIBLIOGRAFÍA

LIBROS

(1) Carlson, J. L. (2013). *Redis in Action*. Manning: USA.

(2) Harrison, G. (2015). *New Generation Databases: NoSQL, NewSQL and Big Data.* Apress: New York.

(3) McCreary, D. y A. Kelly (2013). *Making sense of NoSQL. A guide for Managers and the Rest of Us.* Manning Shelter Island: USA.

(4) Sullivan, D. (2015). *NoSQL for Mere Mortals*. Pearson Education: USA.

ARTÍCULOS ACADÉMICOS

(1) Chen, JK y Lee WZ (2019). An Introduction of NoSQL Databases based on their categories and application industries. *Algorithms*, nº 20-106. DOI: 10.3390/a12050106

MANUALES Y GUÍAS

(1) A Brief Introduction to *Apache Cassandra*. Curso Udemy 2018-2019.

(2) An Introduction to the Hadoop Distributed File System. Disponible en: http://IBM.com/developersworks

(3) Documentación Oficial de *MongoDB*. Disponible en: https://docs.mongodb.com/

(4) Documentación Oficial de *Redis*. Disponible en: https://redis.io/documentation

(5) Documentación Oficial de *Neo4J.* Disponible en: https://neo4j.com/docs/

OTROS DATOS DE JUSTIFICACIÓN DEL CURSO DE FORMACIÓN

(1) Informe del Mercado de Trabajo de los Jóvenes. Estatal. Datos 2017 (2018). Observatorio de las Ocupaciones. Disponible en: http://www.sepe.es/contenidos/observatorio/mercado_trabajo/3067-1.pdf

(2) Información sobre Formación Profesional en la Comunidad de Madrid. Disponible en: www.comunidad.madrid/servicios/educacion/formacion-profesional

(3) Ley Orgánica 6/2001, de 21 de diciembre, de Universidades. BOE nº 307, de 24 de diciembre de 2001. BOE-A-2001-24515. Disponible en: https://www.boe.es/buscar/pdf/2001/BOE-A-2001-24515-consolidado.pdf

PÁGINAS WEB

(1) https://www.boe.es/diario_boe/txt.php?id=BOE-A-2014-2360

(2) https://www.boe.es/diario_boe/txt.php?id=BOE-A-2014-5591

(3) https://www.boe.es/buscar/doc.php?id=BOE-A-2008-819

(4) https://www.boe.es/buscar/doc.php?id=BOE-A-2009-18355

(5) https://www.boe.es/buscar/doc.php?id=BOE-A-2010-8067

(6) https://www.boe.es/buscar/doc.php?id=BOE-A-2010-9269

(7) www.comunidad.madrid/servicios/educacion/formacion-profesional

ANEXO 1

SERVICIOS E INFORMACIÓN SOBRE FORMACIÓN PROFESIONAL DE LA COMUNIDAD DE MADRID

(*www.comunidad.madrid/servicios/educacion/formacion-profesional*)

Según la página de información de la Comunidad de Madrid, existen tres modalidades graduales de formación profesional (FP) en el marco de "Informática y Comunicaciones". A continuación, se detallan las modalidades de FP y las propuestas formativas oficiales para cada una de estas:

1. **Formación Profesional Básica**
   - 1.1. Título Profesional Básico en Informática y Comunicaciones
     - Listado de competencias completas:
       - ✓ Realizar operaciones auxiliares de montaje de equipos microinformáticos
       - ✓ Realizar operaciones auxiliares de mantenimiento de sistemas microinformáticos
       - ✓ Realizar operaciones auxiliares con tecnologías de la información y la comunicación
       - ✓ Realizar operaciones de ensamblado en el montaje de equipos eléctricos y electrónicos
       - ✓ Realizar operaciones de conexionado en el montaje de equipos eléctricos y electrónicos
       - ✓ Realizar operaciones auxiliares en el mantenimiento de equipos eléctricos y electrónicos
   - 1.2. Título Profesional Básico en Informática de Oficina
     - Listado de cualificaciones profesionales completas:
       - ✓ Realizar operaciones auxiliares de montaje de equipos microinformáticos
       - ✓ Realizar operaciones auxiliares de mantenimiento de sistemas microinformáticos
       - ✓ Realizar operaciones auxiliares con tecnologías de la información y la comunicación
     - Listado de cualificaciones profesionales "incompletas":
       - ✓ Realizar operaciones básicas de tratamiento de datos y textos, y confección de documentación
       - ✓ Realizar operaciones auxiliares de reproducción y archivo en soporte convencional o informático
2. **Formación Profesional o Ciclo Formativo de Grado Medio**
   - 2.1. Técnico en Sistemas Microinformáticos y Redes

- Listado de cualificaciones profesionales completas:
  - ✓ Instalar y configurar el software base en sistemas microinformáticos
  - ✓ Instalar, configurar y verificar los elementos de la red local según procedimientos establecidos.
  - ✓ Instalar, configurar y mantener paquetes informáticos de propósito general y aplicaciones específicas.
  - ✓ Facilitar al usuario la utilización de paquetes informáticos de propósito general y aplicaciones específicas.
  - ✓ Montar equipos microinformáticos
  - ✓ Instalar y configurar el software base en sistemas microinformáticos.
  - ✓ Reparar y ampliar equipamiento microinformático.
  - ✓ Instalar, configurar y verificar los elementos de la red local según procedimientos preestablecidos.
  - ✓ Monitorizar los procesos de comunicaciones de la red local.
  - ✓ Realizar los procesos de conexión entre redes privadas y redes públicas.
  - ✓ Instalar y configurar el software base en sistemas microinformáticos.
  - ✓ Mantener y regular el subsistema físico en sistemas informáticos.
  - ✓ Ejecutar procedimientos de administración y mantenimiento en el software base de aplicación del cliente.
  - ✓ Mantener la seguridad de los subsistemas físicos y lógicos en sistemas informáticos.

3. Formación Profesional o Ciclo Formativo de Grado Superior

3.1. Técnico Superior en Administración de Sistemas Informáticos en Red

- Listado de cualificaciones profesionales completas:
  - ✓ Administrar los dispositivos hardware del sistema.
  - ✓ Instalar, configurar y administrar el software de base y de aplicación del sistema.
  - ✓ Asegurar equipos informáticos.
  - ✓ Instalar, configurar y administrar el software para gestionar un entorno web.
  - ✓ Instalar, configurar y administrar servicios de mensajería electrónica.
  - ✓ Instalar, configurar y administrar servicios de transferencia de archivos y multimedia.
  - ✓ Gestionar servicios en el sistema informático.
  - ✓ Configurar y explotar sistemas informáticos.
  - ✓ Configurar y gestionar un sistema gestor de bases de datos.
  - ✓ Configurar y gestionar la base de datos.
- Listado de cualificaciones profesionales “incompletas”:
  - ✓ Implementar, verificar y documentar aplicaciones web en entornos internet, intranet y extranet.

3.2. Técnico Superior en Desarrollo de Aplicaciones Multiplataforma

- Listado de cualificaciones profesionales completas:
  - ✓ Configurar y explotar sistemas informáticos.
  - ✓ Programar bases de datos relacionales.
  - ✓ Desarrollar componentes software en lenguajes de programación estructurada.
  - ✓ Desarrollar componentes software en lenguajes de programación orientados a objetos.
- Listado de cualificaciones profesionales "incompletas":
  - ✓ Instalar y configurar sistemas de planificación de recursos empresariales y de gestión de relaciones con clientes.
  - ✓ Crear elementos software para la gestión del sistema y sus recursos.

3.3. Técnico Superior en Desarrollo de Aplicaciones Web

- Listado de cualificaciones profesionales completas:
  - ✓ Desarrollar elementos software en el entorno cliente.
  - ✓ Desarrollar elementos software en el entorno servidor.
  - ✓ Implementar, verificar y documentar aplicaciones web en entornos internet, intranet y extranet.
- Listado de cualificaciones profesionales "incompletas":
  - ✓ Configurar y explotar sistemas informáticos tanto en lenguajes estructurados de aplicaciones de gestión como en lenguajes orientados a objetos.
  - ✓ Programar bases de datos relacionales tanto en lenguajes estructurados de aplicaciones de gestión como en lenguajes orientados a objetos.

ANEXO 2

BOLETÍN OFICIAL DEL ESTADO - BOE

Ley Orgánica 6/2001, de 21 de diciembre, de Universidades (selección de apartados aplicables a la justificación de este trabajo)

Título Preliminar (De las funciones y autonomía de las Universidades)

Artículo 1, apartado 2:

Son funciones de la Universidad al servicio de la sociedad:

a) La creación, desarrollo, transmisión y crítica de la ciencia, de la técnica y de la cultura.
b) La preparación para el ejercicio de actividades profesionales que exijan la aplicación de conocimientos y métodos científicos y para la creación artística.
c) La difusión, la valorización y la transferencia del conocimiento al servicio de la cultura, de la calidad de la vida, y del desarrollo económico.

d) La difusión del conocimiento y la cultura a través de la extensión universitaria y la formación a lo largo de toda la vida.

Título II (De la estructura de las Universidades) **/ Capítulo I** (De las universidades públicas), **Artículo 8, apartado 1**:

1. Las escuelas y facultades son los centros encargados de la organización de las enseñanzas y de los procesos académicos, administrativos y de gestión conducentes a la obtención de títulos de grado. Podrán impartir también enseñanzas conducentes a la obtención de otros títulos, así como llevar a cabo aquellas otras funciones que determine la universidad.

Artículo 9 (Departamentos)

1. Los departamentos son las unidades de docencia e investigación encargadas de coordinar las enseñanzas de uno o varios ámbitos del conocimiento en uno o varios centros, de acuerdo con la programación docente de la universidad, de apoyar las actividades e iniciativas docentes e investigadoras del profesorado, y de ejercer aquellas otras funciones que sean determinadas por los estatutos.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITÉCNICA
SUPERIOR

Universidad
de Alcalá