Tema 3. Modelado de bases de datos y técnicas de gestión de Big Data

viu Universidad Internacional de Valencia

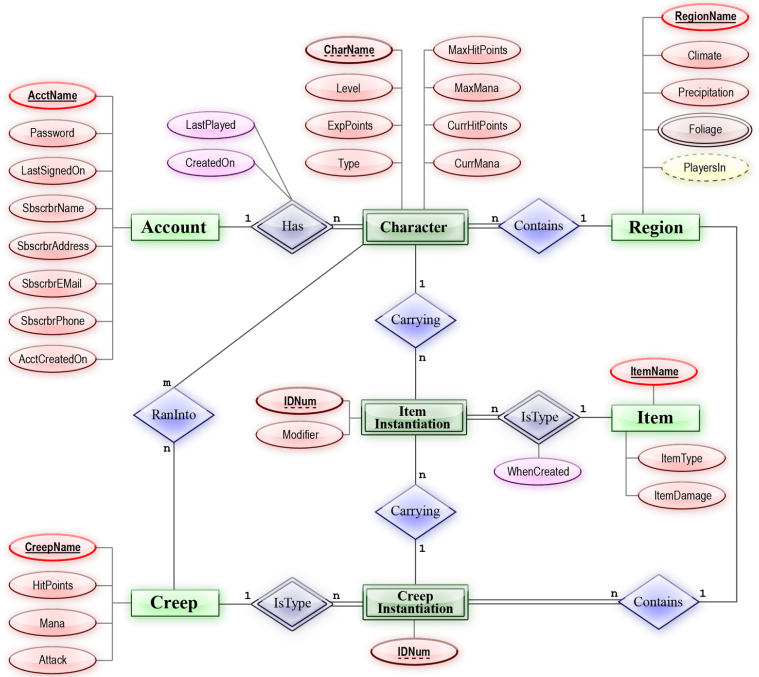
31.10.24

¿Cómo diseñar una BBDD?

- Evitar el diseño directo.
- Seguir metodología.
- Identificar todos los niveles:
  - Conceptual
  - Lógico
  - Físico
- Estos 3 modelos serán usados para modelar en varias BBDD, desde relacionales hasta NoSQL

Modelo conceptual

- Representan el dominio de la aplicación a un alto nivel, utilizando términos y conceptos familiares para los usuarios de la aplicación, ignorando los aspectos de nivel lógico y físico
- Formados por 3 elementos principales:
  - Entidad
  - Atributo
  - Relación



Obtención del modelo conceptual

- Entender modelo a representar, es decir, el problema.
- Identificar entidades, atributes y relaciones entre atributos.

- Ejemplo de un caso de estudio:
  - Tenemos que representar un modelo conceptual de una empresa que tiene que enviar productos a sus clientes en pedidos

1. Cada pedido tiene un número, y se almacenará el cliente que lo solicita y la fecha en la que se realiza
2. De cada producto se quiere registrar un código de producto, su descripción, su precio y sus existencias
3. Para los clientes se quiere registrar su nombre, su DNI y su dirección
4. Un pedido incluirá una serie de productos cada cual con su cantidad

Entidades del problema

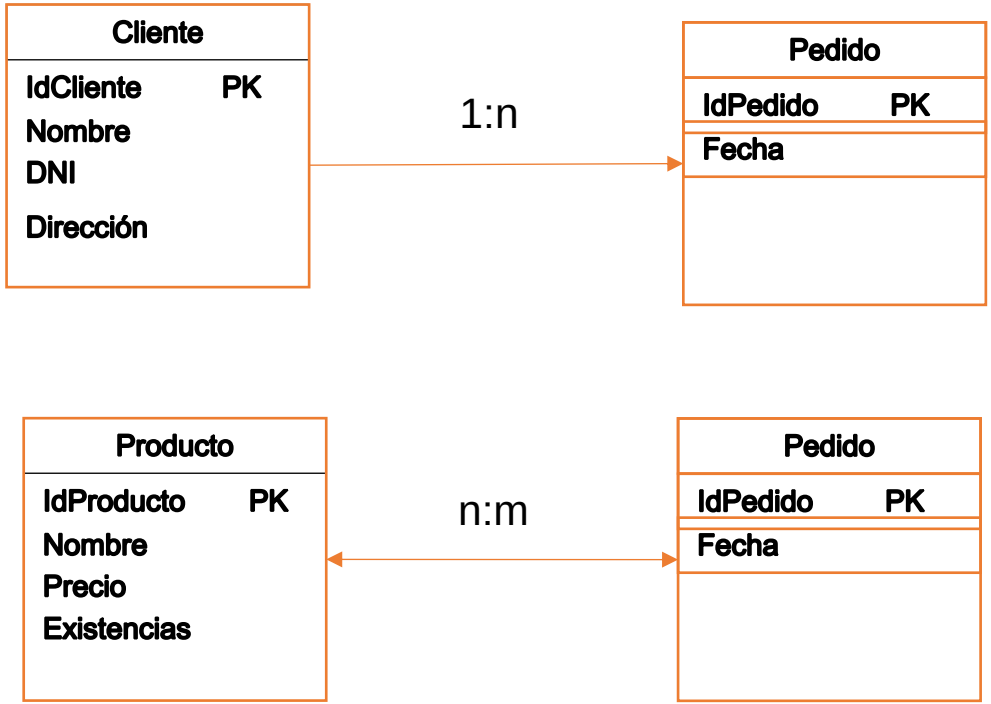
Producto	
IdProducto	PK
Nombre	
Precio	
Existencias	

Pedido	
IdPedido	PK
Fecha	

Cliente	
IdCliente	PK
Nombre	
DNI	
Dirección	

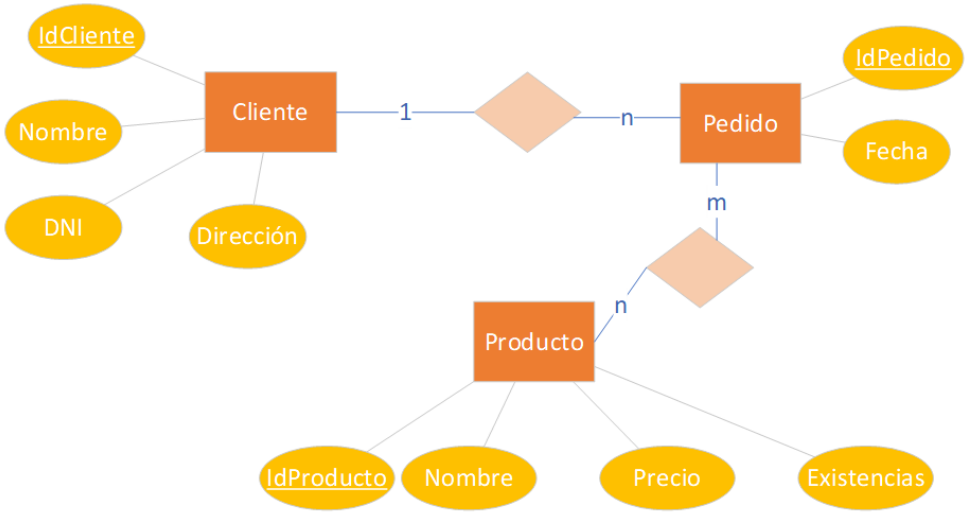
Relaciones

- Tres tipos de relaciones:
  - 1:1
  - 1:n
  - n:m
- “Un cliente puede realizar muchos pedidos, pero un pedido sólo está asociado a un cliente” -> Relación 1:n
- “un producto puede aparecer en muchos pedidos, y un pedido puede contener muchos productos”-> Relación n:m

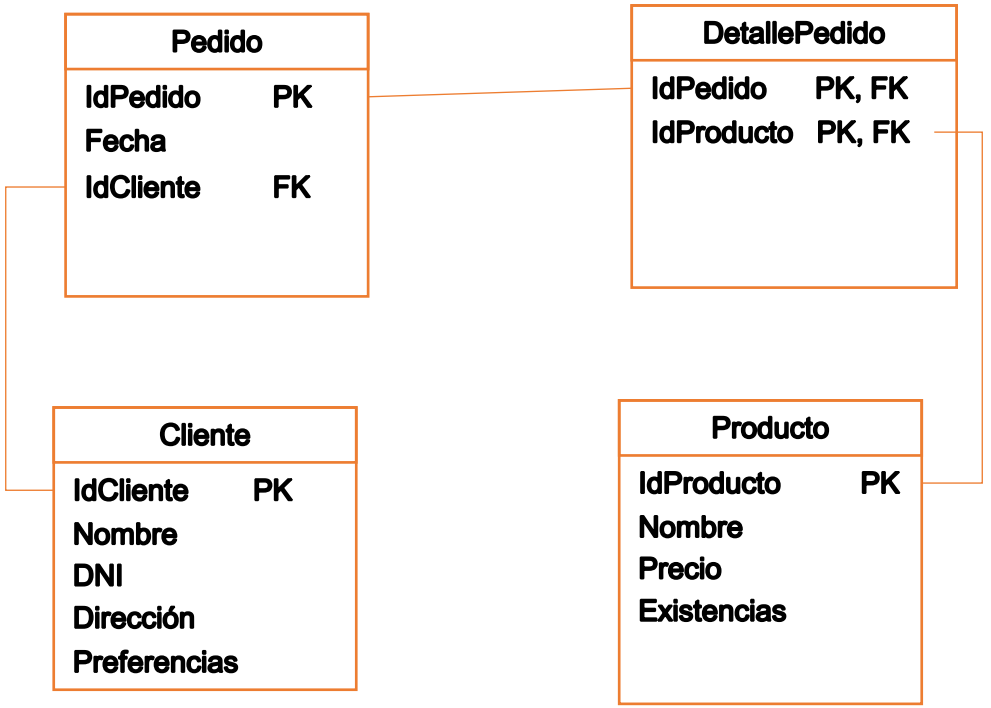


Representación

- **Rectángulos representan entidades.**
- **Elipses representan atributos.**
- **Rombos representan relaciones.**
- **Líneas representan relaciones entre los atributos y las entidades o entre las relaciones y las entidades**
- **Las claves primarias se representan con subrayado.**
- **Doble elipse representa un atributo de tipo conjunto.**

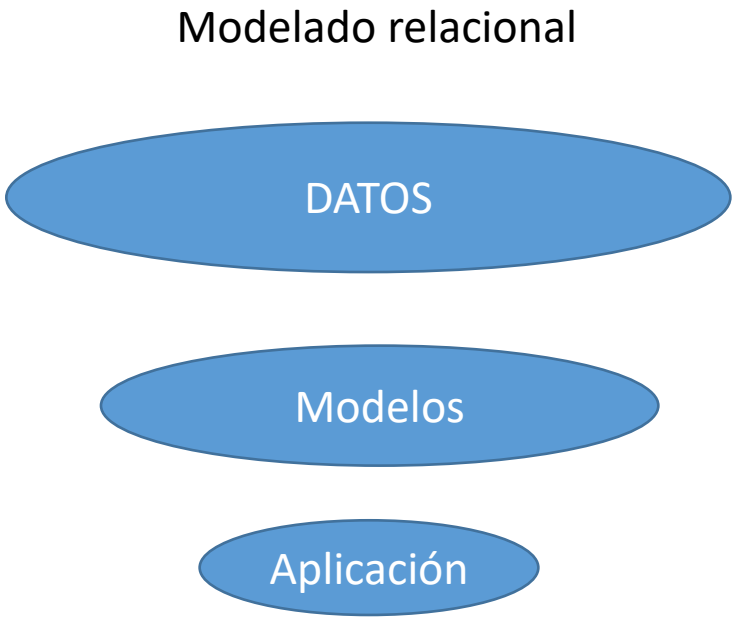


- El modelo lógico sería similar a una base de datos relacional tradicional
- Las entidades serán tablas.
- Los atributos serán columnas.
- Para las relaciones se hace lo siguiente:
  - **Cardinalidad 1:n: Se deberá implementar una clave ajena de la clave primaria de la entidad con cardinalidad 1 en la entidad con cardinalidad n.**
  - **Cardinalidad n:m: Se deberá crear una tabla intermedia. En esta tabla serán tanto clave primaria como ajena las claves primarias de ambas entidades relacionadas.**

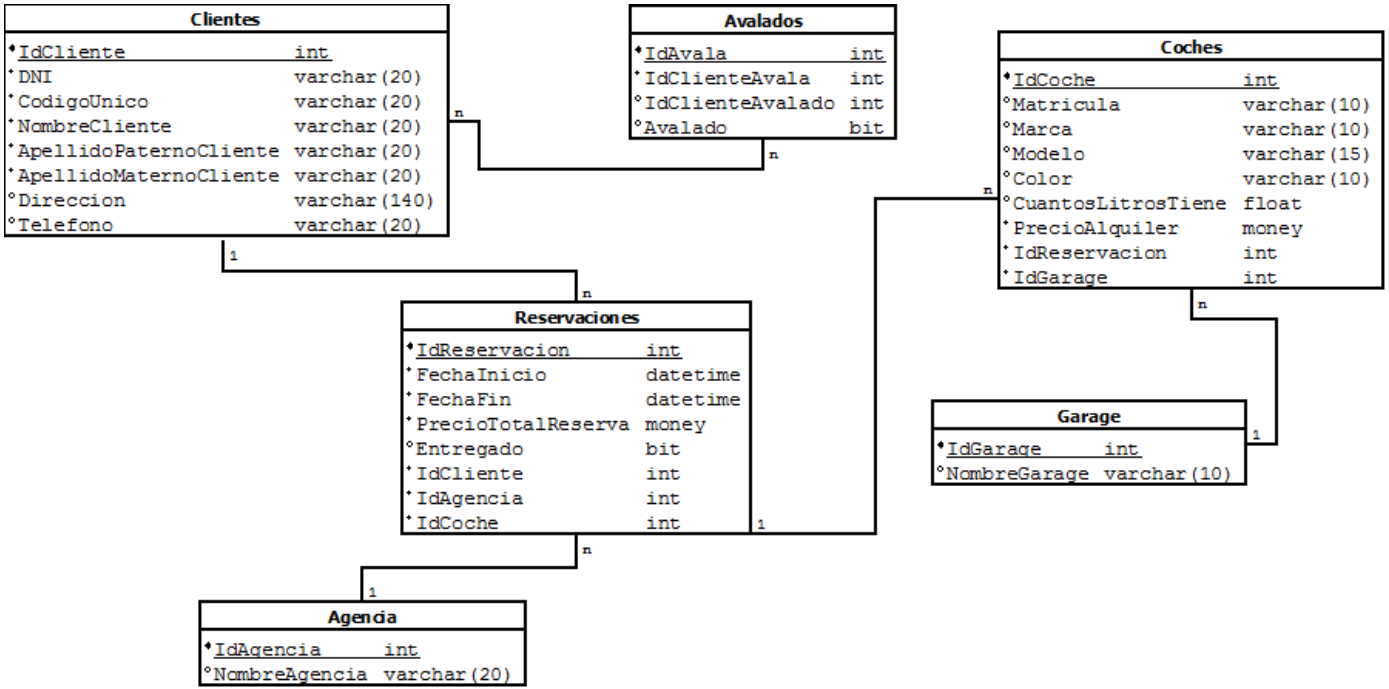


Modelado en una BBDD relacional

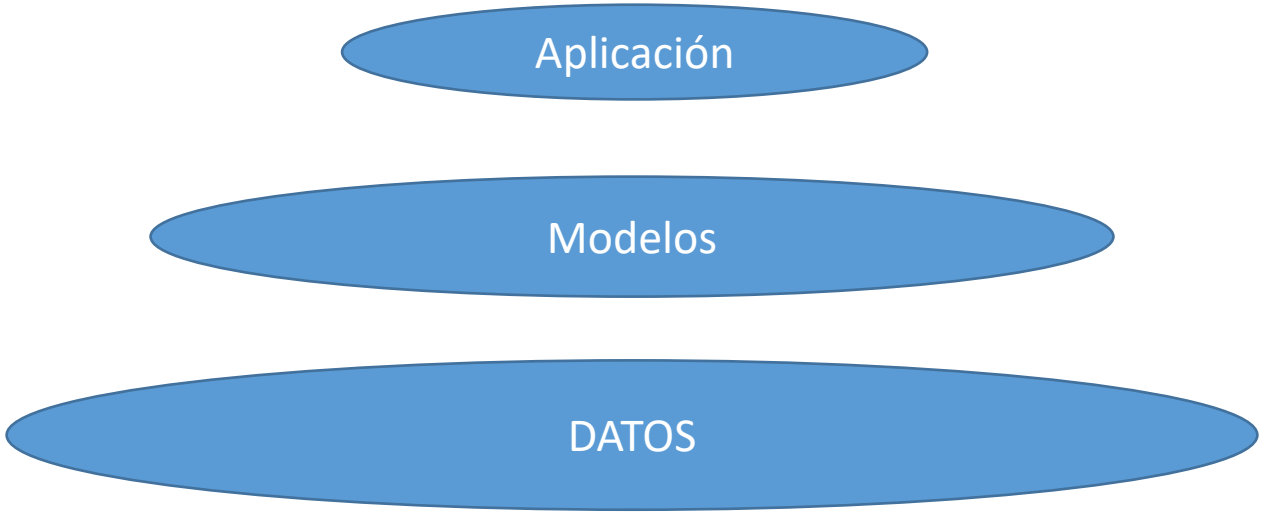
- **En una base de datos relacional modelamos nuestros datos basándonos en los datos.**
- **Lo más común es tener una base de datos normalizada y en cada una de las tablas la única duplicidad de datos serán las claves ajenas que nos servirán para realizar JOINs entre tablas.**
- **Esto en varias bases de datos NoSQL no tendría sentido como por ejemplo las orientadas a documentos y las orientadas a columnas**



Ejemplo de base de datos relacional

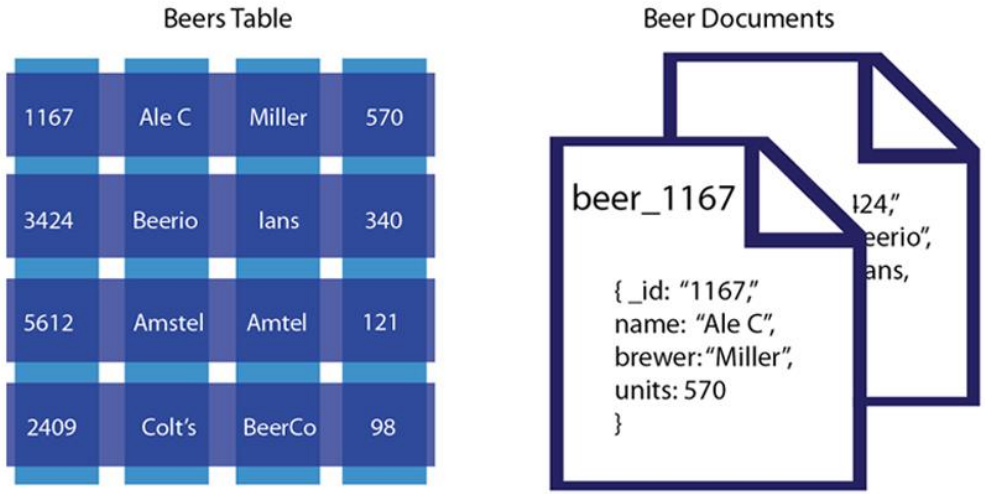


Modelado en documentos y orientada a columnas



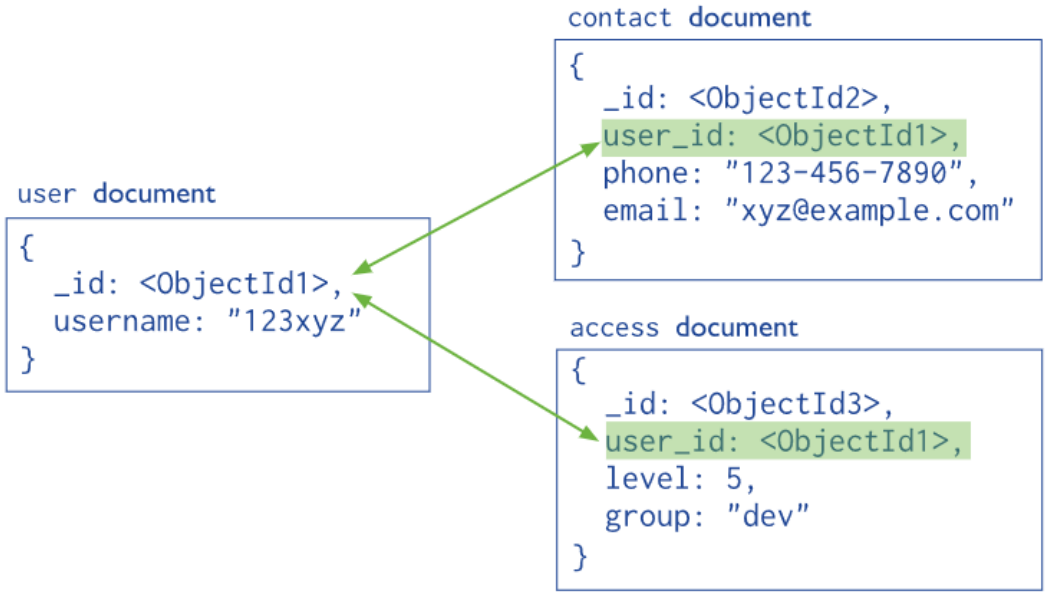
Primeros pasos

- Este tipo de bases de datos pretenden mejorar el rendimiento aplicando la anidación de datos. Cuando hay una consulta que requiere una jerarquía de objetos anidados, esta jerarquía puede ser retornada accediendo a una sola localización
- Para definir el modelo lógico tenemos que tomar dos decisiones:
    1. Identificar cuantas jerarquías se necesitan almacenar en la base de datos.
    2. Determinar cuántos elementos cada jerarquía contendrá.



Identificación jerarquías

1. Análisis de la consulta. Identificar entidades del FROM. Cada una de estas entidades deberá ser documento principal
2. Crear contenido de cada documento
3. Añadir atributos de la entidad principal
4. Identificar relaciones de la entidad principal
    a) Cardinalidad relación
    b) Entidad de la relación
    c) Si entidad en consulta, entonces creamos subdocumento
    d) Repetir desde 1 para cada entidad identificada en c.



Ejemplo aplicado

- Se requiere satisfacer la siguiente consulta

```
SELECT Cliente.Nombre, Cliente.Dni, Cliente.Direccion, Pedido.IdPedido,
Pedido.Fecha, Producto.Nombre, Producto.Precio
FROM Cliente JOIN Pedido JOIN Producto;
```

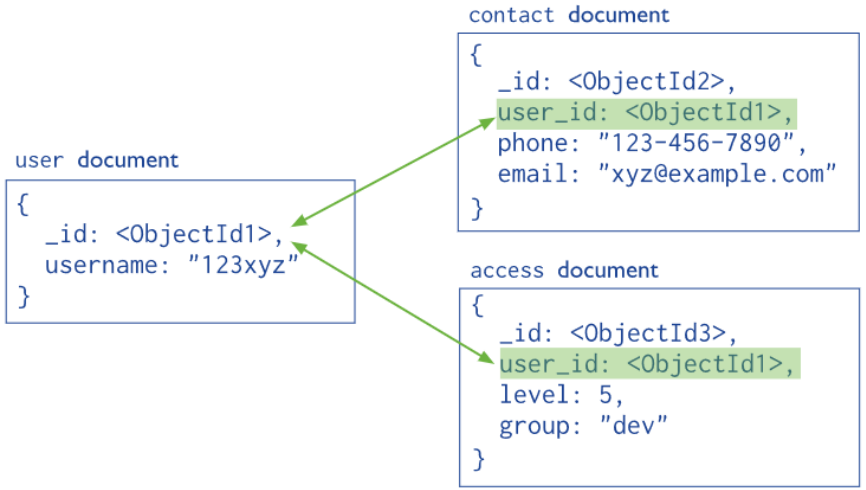
- Primero crearemos un documento basándose en la entidad Cliente.
- Los clientes tienen con pedido una relación de 1:n, por lo que se debe crear una colección de pedidos.
- Cada uno de estos pedidos será un subdocumento que actuará como documento principal.
- Los pedidos pueden tener varios productos. Por lo que dentro de cada pedido se deberán crear varios subdocumentos productos.

Ejemplo aplicado

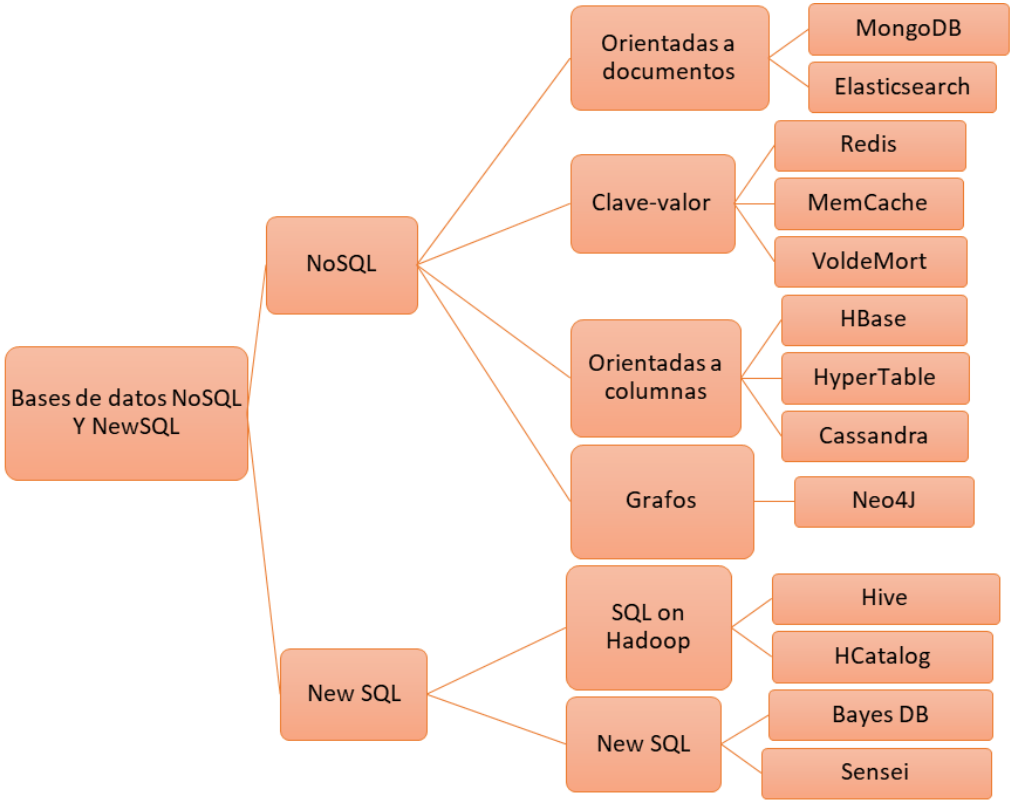
```
Documento Clientes {
   ID: “1”
   Nombre: “María Rodríguez”
   DNI: “73498334G”
   Dirección: “Calle Barcelona, 20, 4I, Ponferrada, León, España”,
            Pedidos [
              {IdPedido: 1
               Fecha: 10-11-2020
               Producto [
                              {IdProducto: 1,
                              Nombre: Peras,
                              Precio: 3},
                              {IdProducto: 2,
                              Nombre: Manzanas,
                              Precio: 2}
               ]
   ]
```

Alternativa modelado

- Otra posibilidad es la de normalizar el modelo a costa de sacrificar rendimiento
- En vez de crear subdocumentos se definirán referencias a la clave primaria de otros documentos
- Cuando usar la normalización:
  - Cuando la desnormalización no provea de ventajas con respecto al rendimiento en las lecturas.
  - Relación varias a varias (n:m) complejas de definir de manera desnormalizada.
  - Para modelar data sets jerárquicos



Soluciones comerciales



- Base de datos NoSQL más popular en 2023, quinta en general*
- Permite búsqueda de información por campos, consulta de rangos o expresiones regulares.
- El retorno puede ser un documento o funciones Javascript.
- Cualquier campo puede ser indexado

*https://db-engines.com/en/ranking

- Consultas se realizan a través del Shell

MongoDB: db.productos.find({})

*SQL: SELECT * FROM productos*

- En este caso queremos obtener los elementos de la colección “productos”

```
⌥⌘1                    ./mongosh
$ mongosh > const nameRegex = /max/i
undefined
$ mongosh > db.users.find({name: nameRegex}, {_id: 0, name: 1})
[
  { name: 'Maximo Heathcote' },
  { name: 'Maximus Borer' },
  { name: 'Maximillian Walker Jr.' },
  { name: 'Maxwell Williamson' }
]
$ mongosh > db.users.fnd()
TypeError: db.users.fnd is not a function
$ mongosh > db.users.find({age: {$gt
db.users.find({age: {$gt   db.users.find({age: {$gte

$ mongosh > db.users.find({age: {$gt
```

Para especificar más la consulta usando el equivalente a un WHERE en SQL, se usa la siguiente sintaxis:

{ <field1>: <value1>, ... }

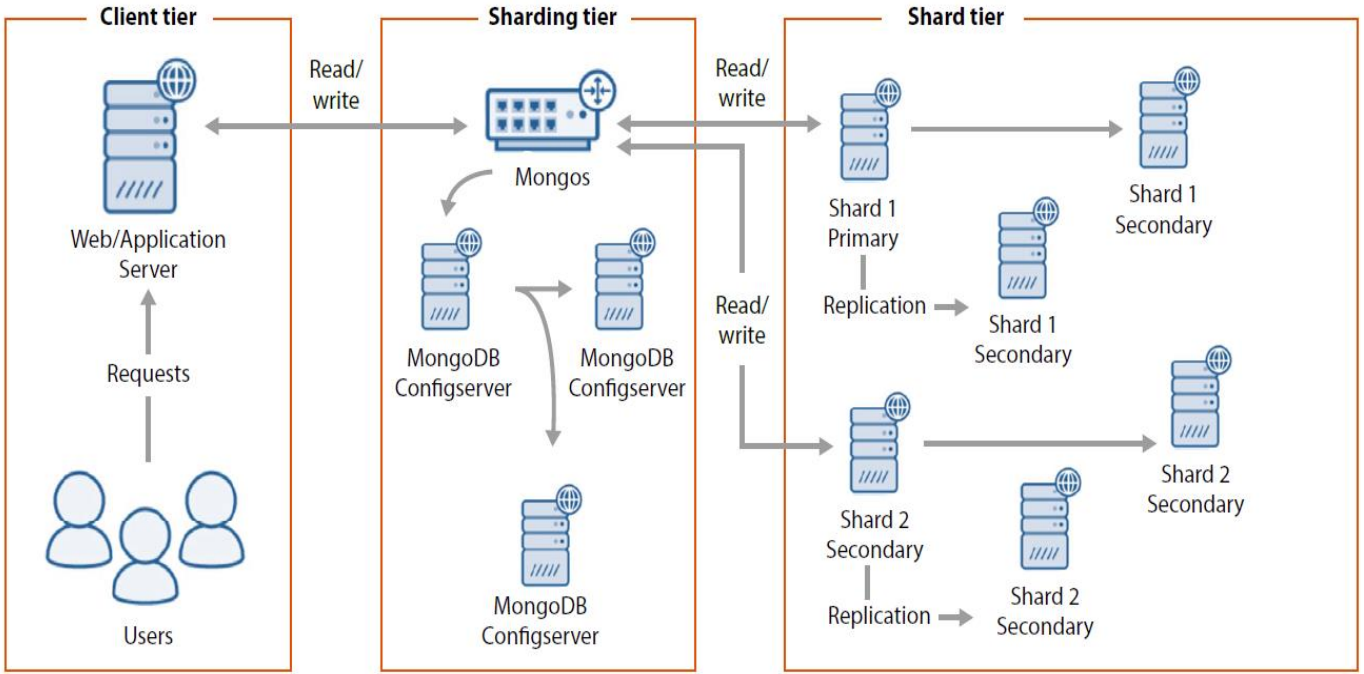
Un ejemplo usando el ejemplo de la base de datos pedidos sería:

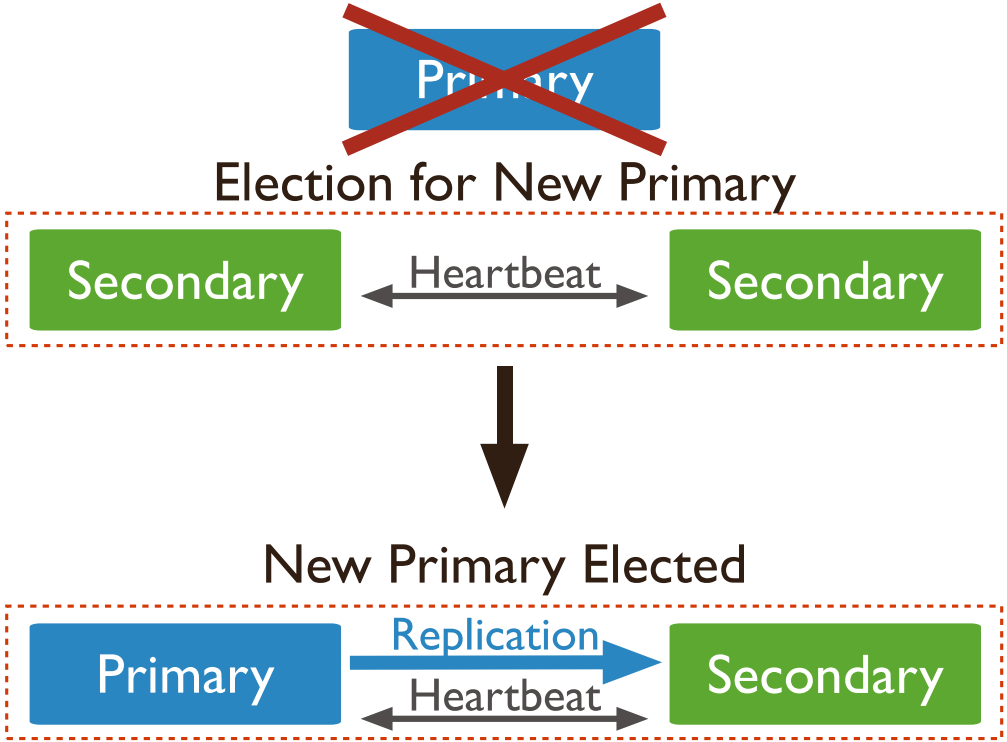
db.productos.find({ price: 10 })

Que sería el equivalente a la consulta SQL

SELECT * FROM productos WHERE price = 10;

MongoDB: Sharding





https://www.mongodb.com/docs/manual/replication/

WiredTiger Storage Engine

- Control de concurrencia a nivel de documento para las operaciones de escritura
- Control de concurrencia optimista
- Al comienzo de una operación, WiredTiger proporciona un *snapshot* de los datos afectados por la operación
- Al escribir en disco, WiredTiger escribe todos los datos de un *snapshot* en disco de una manera consistente con respecto al total de archivos
- Checkpoints

In-Memory Storage Engine

- Los conserva en la memoria para que la latencia de los datos sea más predecible.
- Usa concurrencia a nivel de documento para las operaciones de escritura.
- Los datos no son persistentes
- Destinado a datos de aplicaciones y datos de sistema como usuarios, permisos, índices, etc

- MongoDB escoge y cachea el plan de consultas más eficiente posible con los índices disponibles
- Esta evaluación se realiza basándose en el número de unidades de trabajo (*works*) que participan en el plan de ejecución de la consulta
- Existe una entrada de caché de plan asociados que se usa para mejorar el rendimiento de consultas futuras con la misma forma. Esta entrada puede tener los siguientes estados:
    - Missing: No existe todavía un plan asociado.
    - Inactive: Esta entrada es un marcador de posición. Es decir, el planificador ha calculado el coste de la consulta y ha introducido el marcador, pero aún no se ha asignado un plan. Se iniciará un proceso para elegir el plan ganador.
    - Active: En la entrada se encuentra el plan ganador. Esta entrada será usada para crear planes de consulta.

Apache Cassandra

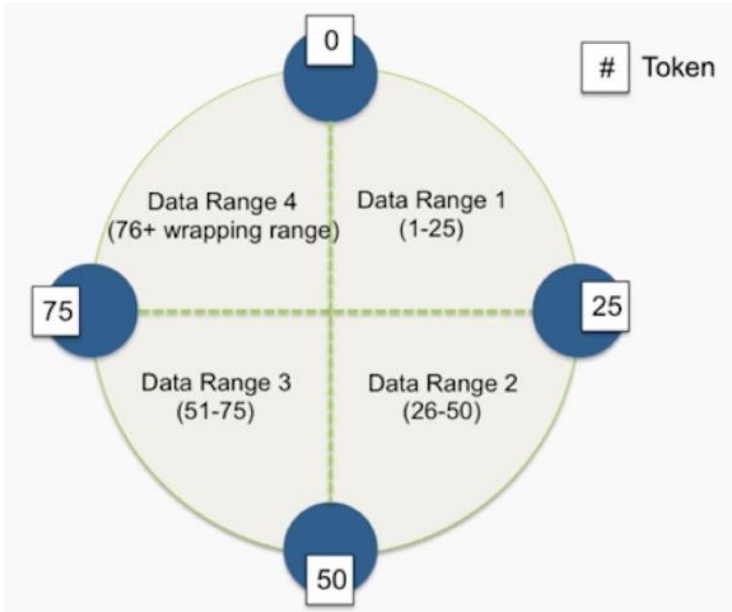
- **Base de datos NoSQL orientada a columnas.**
- **Lanzada inicialmente en 2008 por Avinash Lakshman y Prashant Malik**
- **Ofrece distribución de datos entre diferentes nodos que componen el cluster.**
- **Utilización del lenguaje CQL (Cassandra Query Language) para realizar transacciones.**

Apache Cassandra

- **Al encontrarse la información distribuida y replicada nos proporciona alta disponibilidad.**
- **Es escalable horizontalmente.**
- **No tiene SPOF (puntos únicos de fallo). Es tecnología peer-to-peer**
- **El rendimiento que nos podemos esperar con un determinado hardware es predecible.**

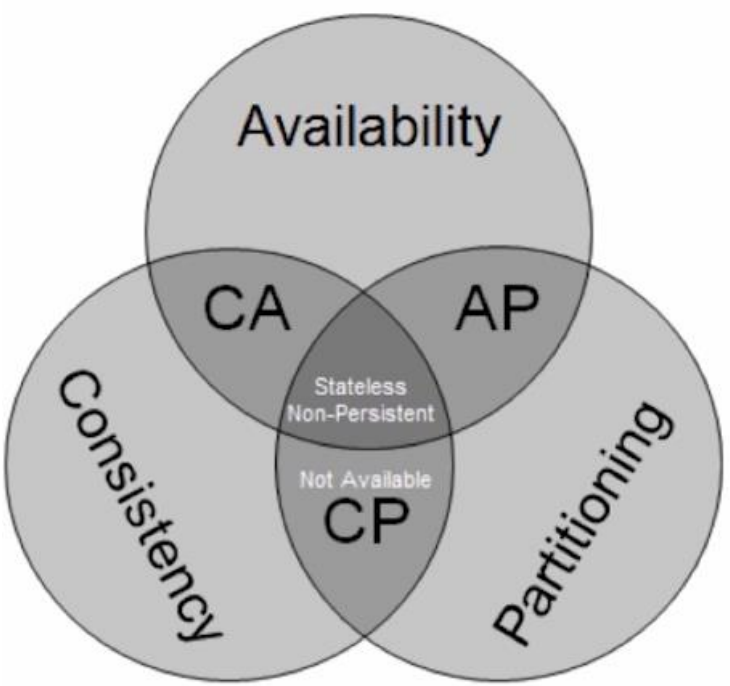
Apache Cassandra

- **No hay maestros y esclavos ni bases de datos de réplica.**
- **Los datos están distribuidos a lo largo del anillo.**
- **Todos los nodos tienen datos y pueden contestar a preguntas.**
- **La localización de los datos se determina por la partition key.**



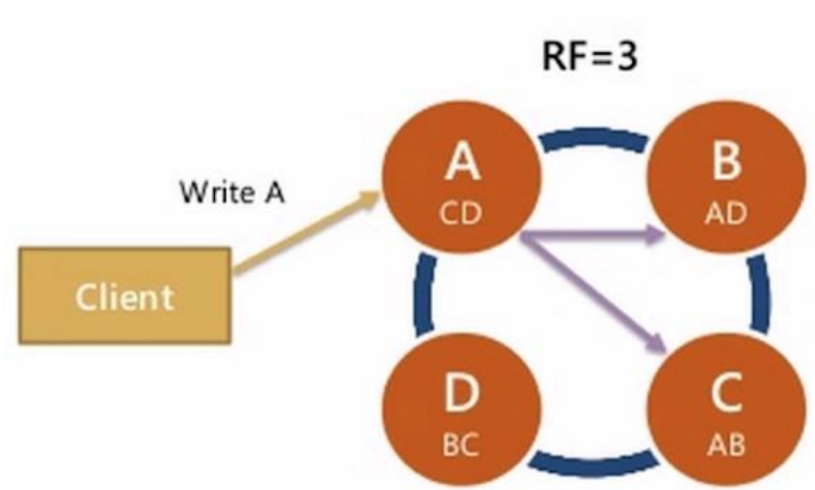
Apache Cassandra

- **Es imposible tener consistencia y alta disponibilidad en un entorno particionado.**
- **La latencia entre los diferentes nodos también hace impráctica la consistencia**
- **Cassandra elige la disponibilidad y la tolerancia a la partición sobre la consistencia.**



Apache Cassandra

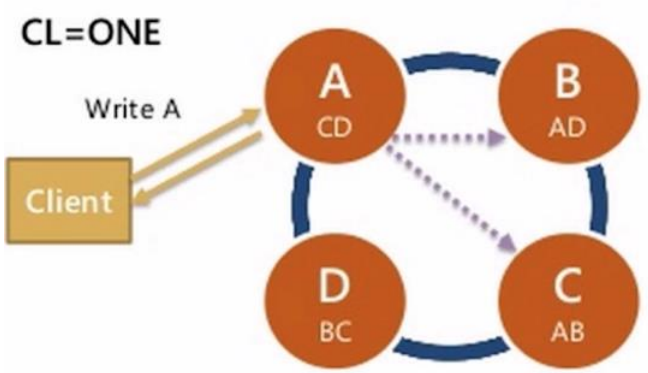
- **Los datos se replican de forma automática.**
- **El desarrollador elige el número de servidores en los que se replicarán los datos.**
- **Los datos se replican a cada réplica que le corresponda.**
- **Si una máquina se cae, los datos almacenados en esa máquina son replicados a otras del cluster.**

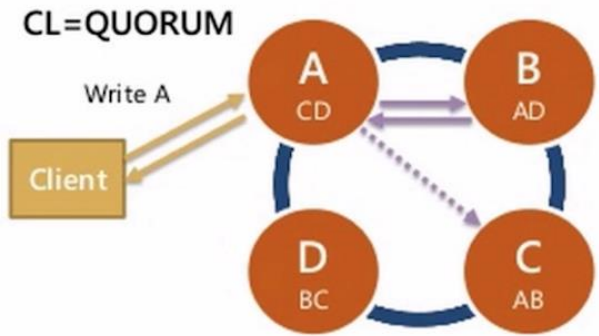


Apache Cassandra

Niveles de consistencia:

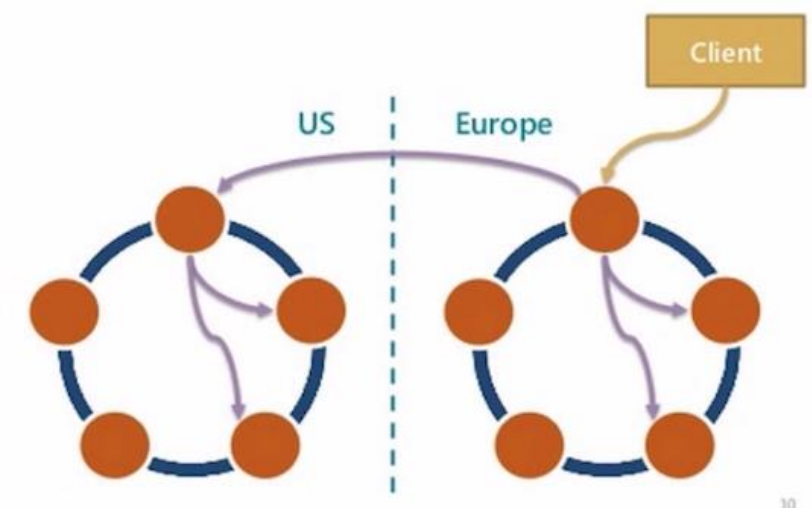
- **Cuantas replicas necesito consultar para garantizar que una transacción es correcta.**
- **Tipos:**
  - **One: Solo una réplica**
  - **Quorum: Mayoría de réplicas. Por ejemplo si tenemos 3 réplicas, la mayoría sería 2.**
  - **All: Todas las réplicas.**
- **Cuantos más nodos necesitemos, más tiempo tardaremos para ejecutar la operación.**





Apache Cassandra

- **Podemos tener varios datacenters. Normalmente escribiremos en un solo datacenter y automáticamente se replicará a otros.**
- **Podemos configurar la replicación de cada keyspace por separado.**
- **Los datacenters pueden ser tanto físicos como lógicos.**



Apache Cassandra

Más elementos técnicas:

- **Se inserta información a través de un nodo que actuará de coordinador.**
- **Sstables: Donde se guarda la información en disco.**
- **Las sstables nos permiten realizar copias de seguridad rápidamente.**

Níveles de compactación.

- **Cassandra almacena los datos en SSTables. A lo largo del tiempo Cassandra puede llegar a almacenar información de una fila en varias SSTables. Para mantener la salud de la BD, Cassandra periódicamente junta SSTables y descarta datos viejos. Esto se llama compactación.**
- **Estrategias de compactación:**
  - **LeveledCompactionStrategy (LCS): SSTables de un tamaño fijo relativamente pequeño (160Mb por defecto) que son agrupados por niveles. Cada nivel es 10 veces más grande que el anterior. En cada nivel se juntan filas clave en SSTables del siguiente nivel. Se recomienda para lecturas intensivas.**
  - **SizeTieredCompactionStrategy (STCS): Estrategia por defecto. Esta estrategia comprime cuando hay un número de SSTables de tamaño similar en disco que coincida con el valor establecido en min_threshold. Recomendado para escrituras intensivas.**
  - **TimeWindowCompactionStrategy: Junta STCS con el uso de ventanas de tiempo. En una ventana de tiempo junta todas las SSTables en una sola utilizando STCS. Al acabar la ventana del tiempo todas las SSTables son juntadas en una sola ventana. Se recomienda para series temporales y datos que tienen un tiempo de vida prefijado.**

Cacheado

- **Cassandra incluye cacheado integrado y distribuye los datos de caché a lo largo del cluster.**
- **Soluciona el problema del comienzo en frío al salvar cache en disco periódicamente. Cassandra lee el contenido del cache y distribuye los datos cuando se reinicia. El cluster no comienza con caché en frío.**
- **Partition key cache: Recomendable quitarlo, malos resultados en experimentación.**
- **Row cache: Solo activarlos cuando el número de lecturas sea mucho mayor que el número de escrituras. Al menos un 95%.**

```
Key Cache                : entries 26, size 2.35 KiB, capacity 100 MiB, 113 hits,
                           140 requests, 0.807 recent hit rate, 14400 save period in seconds
Row Cache                : entries 1, size 346 bytes, capacity 100 MiB, 1 hits,
                           2 requests, 0.500 recent hit rate, 0 save period in seconds
Counter Cache            : entries 0, size 0 bytes, capacity 50 MiB, 0 hits, 0 requests,
                           NaN recent hit rate, 7200 save period in seconds
Chunk Cache              : entries 33, size 2.06 MiB, capacity 480 MiB,
                           47 misses, 264 requests, 0.822 recent hit rate,
                           NaN microseconds miss latency
```

Configuración Cassandra

- **Debemos elegir la estrategia de replicación de nuestra base de datos, la cual determinará qué nodos son replicas:**
  - **SimpleStrategy: Determina el número de nodos en las que está copiada una fila. Si el valor de replication_factor es 3 entonces tres nodos diferentes tendrán esa fila. Sobrepone cualquier configuración que haya en los datacenters.**
  - **NetworkTopologyStrategy: Permite que haya configuraciones específicas por datacenters. Además intenta elegir replicas de un datacentes de diferentes racks. Si el número de racks es mayor o igual al factor de replicación del datacenter, entonces cada réplica estará en un rack diferente.**
  - **TransientReplication: Pemite configurar un subconjunto de replicas para que solo repliquen datos que no han sido reparados permitiendo desacoplar la redundacion de datos de la disponibilidad. Está en fase experimental, por lo que no es recomendado usarlo en entornos de producción No nos podemos olvidar tampoco del nivel de consistencia que habíamos visto antes.**

Storage-Attached Indexing

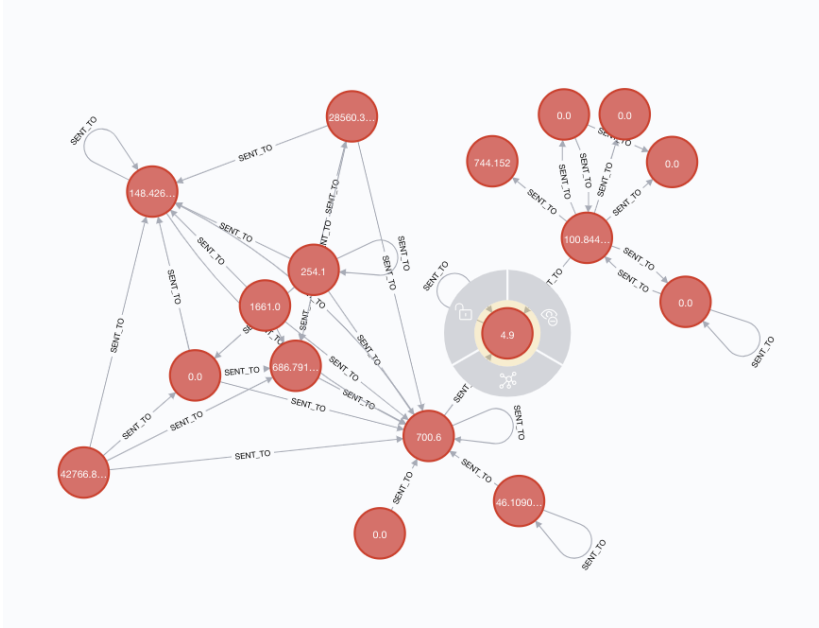
- **En la versión 5.0 se han introducido recientemente una forma más óptima de indexación en las columnas que no sean clave: el uso de Storage-Attached Indexing.**
- **Nos va a permitir realizar consultas sobre columnas no clave, sin las altas penalizaciones de rendimiento de anteriores versiones.**
- **A cuantos más datos, peor rendimiento ofrecen, por lo que en problemáticas big data podrían no ser una opción a considerer.**
- **Cuántos más índices en la consulta, peor rendimiento.**
- **Actualmente en la versión 1, algunas operaciones como comparaciones de String, operador OR y ordenamiento global se incluirán en la versión 2.**

- Nuevas funciones para CQL.
- Nuevas estrategias de compactación.
- Mejoras en las SSTables: Optimización de la memoria y el uso del almacenamiento.
- Búsqueda vectorial, que aprovecha las técnicas de indexación de almacenamiento adjunto e indexación densa para transformar la exploración y el análisis de datos.
- Dynamic Data Masking: Útil para no mostrar información sensible en las consultas aunque esté almacenado en la base de datos.

- Base de datos orientadas a grafos
- Alternativa recomendable para almacenar datos relacionados entre sí
- Esquema flexible

Estructuras

- **Nodos-Representan instancias de entidades como por ejemplo documentos, usuarios, recetas, etc. Pueden contener propiedades**
- **Relaciones-Existen entre los diferentes nodos. Se puede acceder a ellas independientes o a través de los nodos a los que están unidos. Las relaciones también pueden contener propiedades, de ahí el nombre de modelo gráfico de propiedades.**
- **Propiedades: tanto los nodos como las relaciones pueden tener propiedades. Las propiedades están definidas por pares clave-valor.**
- **Etiquetas: Pueden utilizarse para agrupar nodos similares**



Para inserción de datos y creaciones de estructuras se usa CREATE. Por ejemplo, para crear un cliente y un pedido que haya comprado dicho cliente se ejecutará una operación como la siguiente:

```
CREATE (cliente1:Cliente {nombre: ’Manuel’}),
(pedido1:Pedido { fecha:'2020-10-2'}),
(cliente1)-[:Ha_comprado]->( pedido1)
```

Para la consulta que satisfaga la pregunta “¿Qué pedidos ha comprado el cliente de nombre Manuel?” haríamos lo siguiente:

```
Match(c:Cliente{nombre:’Manuel’}) – [:Ha_comprado]-> (p:Pedido)
Return c.name, p.fecha
```

- Las tareas de ejecución de una consulta se descomponen en operadores, cada uno implementado una unidad de trabajo específica
- **Plan de ejecución:**
  - Cada operador se representa como un nodo del árbol
    - Entrada: Cero o más filas
    - Salida: Cero o más filas
  - La salida de cada operador en la en la entrada del siguiente operador
- Los planes de ejecución son evaluados en los nodos hoja del árbol
  - Operadores scan y seek
    - Obtienen los datos directamente del motor de almacenamiento, incurriendo por lo tanto en consultas a la base de datos
  - Cualquier fila producida por lo nodos hoja es reconducida a sus nodos padre, que a su vez la reconduce a sus nodos padres y así una y otra vez hasta llegar al nodo raíz. El nodo raíz produce el resultado final de la consulta

- Evaluaciones de consultas lazy
- La mayoría de los operadores reconducen las filas de salida a sus operadores padres tan pronto como estos se producen.
- Operadores como los de ordenación y ordenamiento, necesitan procesar todas las filas antes de enviárselas a sus padres.
    - causan altos consumos de memoria y por lo tanto pueden afectar negativamente al rendimiento de las consultas

Novedades en 4.0

- Introducido en versiones recientes.
- Cuando se recomienda usar sharding en Neo4J:
  - Por razones legales en la que determinados datos se almacenarán en un shard específico
  - Datos que ya están separados en instancias pero que se pueden entrelazar en un grafo unificado.
  - Minimizar latencia en regiones almacenando los datos más relevantes de cada posición geográfica en dichas regiones.
  - Archivar datos en desuso.
  - Cuando el grafo se ha vuelto demasiado grande y es necesario particionarlo.
- Se usa una base de datos virtual que actúa como coordinador, la cual no almacena datos.
- Es muy relevante tener un buen esquema ya que será fundamental para la elección de cómo dividir los datos en shardings de forma adecuada.
- Similar a Cassandra, es también recomendable saber qué consultas serán ejecutadas contra la base de datos, aunque no totalmente necesario.

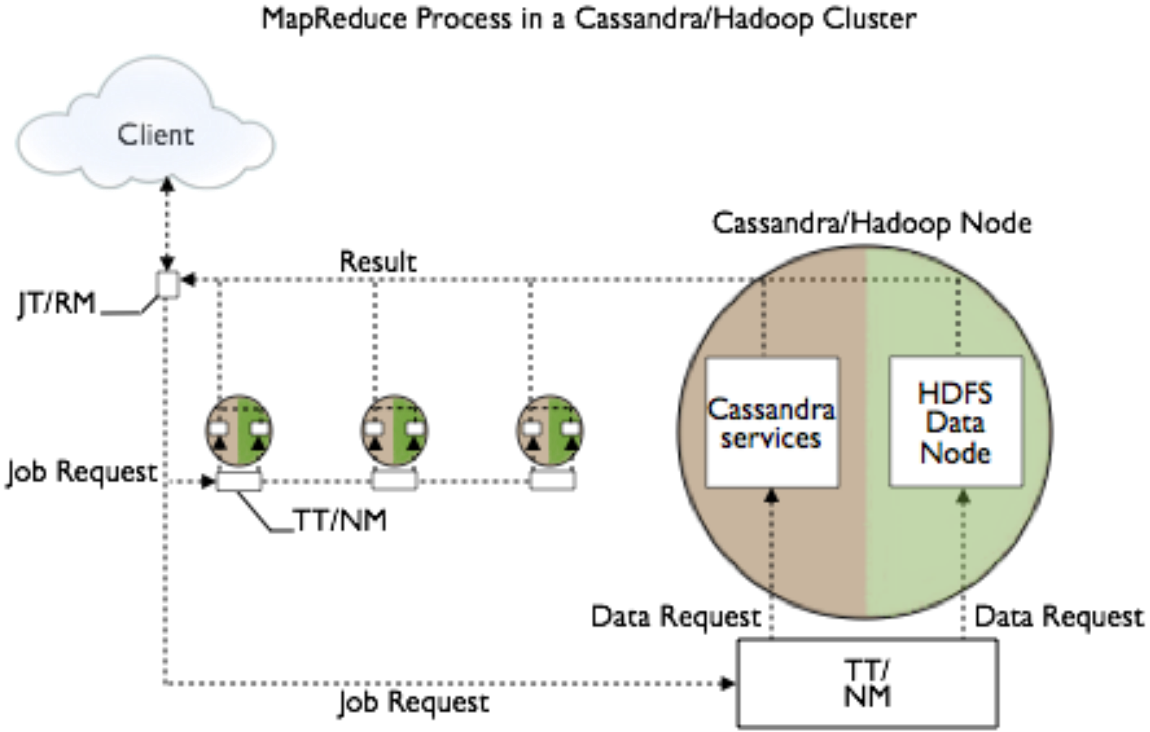
- Diferentes estrategias.
  - Time window: Buena para el análisis y sistemas basados en eventos como transacciones.
  - Logical domain entity: Bueno para entornos en la nube SaaS.
  - Por localización geográfica

- Clustering autónomo.
  - Más facilidad para la replicación de datos.
  - No automatiza sharding
- Consultas entre diferentes shards
- Mejora en la construcción de consultas
- Mejores opciones de optimización.
- Etc

Redis

- Base de datos clave-valor
- Guardan la información en memoria RAM o caché en vez de en disco
- Alta capacidad de lectura y escritura
- Sistema de réplica de una BBDD principal
- No orientada a almacenar Big Data por sí sola.
- Recientemente están haciendo avances para que Redis pueda ser usada como base de datos principal y abandonar el concepto de que solo se puede usar para caché.

Cassandra

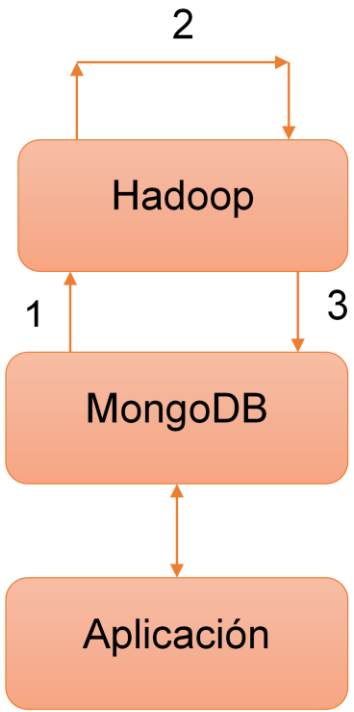


MongoDB: Agregación de lotes

1. Los datos se extraen de MongoDB y se procesan dentro de Hadoop a través de trabajos MapReduce. Los datos también pueden provenir de otros lugares que no sean MongoDB con el objetivo de tener una solución con múltiples fuentes de datos.

2. La salida que proporcionan estos trabajos MapReduce puede escribirse de nuevo en MongoDB para su consulta posterior y para cualquier análisis ad hoc.

3. Las aplicaciones construidas sobre MongoDB pueden, por tanto, utilizar la información de los análisis por lotes para presentarla al cliente final o para habilitar otras funciones posteriores.



MongoDB: Data Warehousing

Los datos pueden provenir de diferentes fuentes de información, pudiendo contar cada fuente con sus propios lenguajes de consultas y funcionalidad. Para reducir la complejidad en estos escenarios, Hadoop puede ser usados como data warehouse y actuar como un repositorio centralizado para estas fuentes. En este escenario se haría lo siguiente:

- Trabajos periódicos MapReduce cargan datos de MongoDB a Hadoop.
- Una vez que los datos de MongoDB y de otras fuentes esté en Hadoop, se puede hacer una consulta sobre todos los datos juntos.