ISTEA

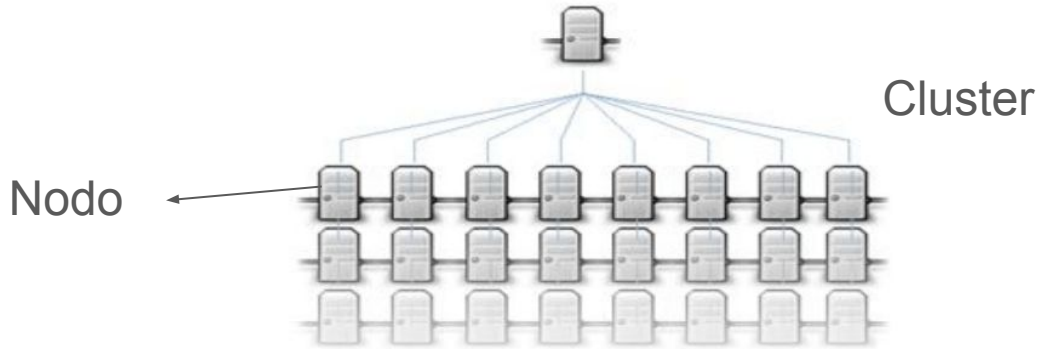
Bases de Datos No Relacionales

- Replicación

- Sistemas distribuidos

Cuando nos encontramos frente a sistemas que administran grandes volúmenes de información y que tienen alta interacción con diferentes y múltiples usuarios concurrentes, no podemos pensar que un ambiente monolítico o centralizado soportará esto de manera eficiente, debemos pensar en un **sistema distribuido.**

Este será grupo de servidores independientes interconectados a través de una red dedicada que trabajarán como un único recurso de procesamiento/almacenamiento.



- Características de los sistemas distribuidos

Concurrencia: Al estar compuestos por múltiples nodos (servidores) permiten que la carga de trabajo pueda distribuirse entre los diferentes nodos.

Transparencia: Un sistema distribuido ideal es transparente para los usuarios y las aplicaciones, lo que significa que estos no deben ser conscientes de la distribución de los componentes del sistema. El sistema debe presentarse como si fuera un único sistema monolítico.

No hay dependencia de los componentes: Un sistema distribuido debe ser capaz de seguir funcionando incluso si algunos de sus componentes fallan. Esto se logra mediante la **replicación** de datos y servicios, y el uso de mecanismos de tolerancia a fallos.

- Características de los sistemas distribuidos

Escalabilidad: Deben ser escalables, lo que significa que deben poder adaptarse a un aumento o disminución de la carga de trabajo sin afectar a su rendimiento.

Heterogeneidad: Pueden estar compuestos por nodos con diferentes tipos de hardware y software. Esto requiere el uso de mecanismos para gestionar la heterogeneidad.

Distribución geográfica: Los nodos de un sistema distribuido pueden estar en diferentes ubicaciones geográficas. Esto requiere el uso de mecanismos para gestionar la red y la comunicación entre **nodos remotos**.

- Características de los sistemas distribuidos

Seguridad: Los sistemas distribuidos son más vulnerables a los ataques que los sistemas monolíticos. Es importante implementar medidas de seguridad adecuadas para proteger el sistema y sus datos.

Gestión de la complejidad: Al ser más complejos de gestionar que los sistemas monolíticos se debe contar con herramientas y mecanismos adecuados para gestionar la configuración, el mantenimiento y la monitorización del sistema.

- **Beneficios de los sistemas distribuidos:**

Mayor escalabilidad y rendimiento: Los sistemas distribuidos pueden ***escalarse horizontalmente*** añadiendo más nodos al sistema, lo que permite aumentar la capacidad de procesamiento y almacenamiento.

Mayor disponibilidad: Los sistemas distribuidos son más ***tolerantes a fallos***, ya que si un nodo falla, los demás nodos pueden seguir funcionando.

Mejor rendimiento: La carga de trabajo puede distribuirse entre los diferentes nodos del sistema, lo que puede mejorar el rendimiento general.

Acceso a recursos compartidos: Los sistemas distribuidos permiten a los usuarios acceder a recursos compartidos, como datos y servicios, desde cualquier lugar de la red.

Flexibilidad: Los sistemas distribuidos son más flexibles y adaptables que los sistemas monolíticos, ya que pueden modificarse fácilmente para satisfacer nuevas necesidades.

- **Aplicaciones de los sistemas distribuidos:**

Internet: Internet es el ejemplo más grande y complejo de un sistema distribuido.

Redes sociales: Las redes sociales, como Facebook y Twitter, son sistemas distribuidos que permiten a los usuarios compartir información y comunicarse entre sí.

Comercio electrónico: Los sistemas de comercio electrónico, como Amazon y eBay, son sistemas distribuidos que permiten a los usuarios comprar y vender productos en línea.

Computación en la nube: Es un modelo de servicio que ofrece acceso a recursos informáticos, como almacenamiento y procesamiento, a través de Internet. Son típicamente sistemas distribuidos.

Sistemas de gestión empresariales: Por ejemplo los sistemas ERP (Enterprise Resource Planning) y CRM (Customer Relationship Management), pueden ser sistemas distribuidos que permiten a las empresas automatizar su fuerza de venta y marketing.

La necesidad de poder trabajar en forma eficiente en ambientes distribuidos, es decir basados en clusters, fue uno de los primeros requerimientos que justificaron la aparición del movimiento NoSQL.

Los datos ahora debían poder gestionarse de manera eficiente en estos ambientes.

Una solución: **REPLICACIÓN**

Tipos de replicación

- **Replicación Maestro-Esclavo (Master/Slave)**
- **Replicación Esclavo-Maestro-Esclavo (Slave-Master-Slave)**
- **Replicación entre pares (peer to peer)**

Otras soluciones:

- Particionamiento (Sharding)
- Combinación entre Replicación y Particionamiento

Mediante la **REPLICACIÓN** es decir, **copias múltiples de los datos en diferentes servidores de bases de datos**,se protege a una base de datos de la caída de algún servidor.

Las copias proporcionan **redundancia** y en consecuencia **incrementan la disponibilidad** de los datos.

Se definen estrategias o *planes de recuperación ante desastres* (DPR), para garantizar esta disponibilidad, incluyendo fallas relacionadas con el hardware, las interrupciones de servicio de red, o incluso problemas aún mayores de infraestructura tecnológica.

Tipos de **REPLICACIÓN**

Replicación Maestro-Esclavo (Master/Slave)

- Un nodo es designado como maestro, o primario.
- El nodo maestro adquiere la autoridad de ser fuente de los datos y es generalmente responsable de procesar cualquier actualización de los mismos.
- Los restantes nodos son esclavos, o secundarios.
- La replicación sincroniza los nodos esclavos desde el maestro.
- La lectura de datos puede realizarse desde el nodo maestro o desde cualquier nodo esclavo.

Tipos de **REPLICACIÓN**

Replicación Maestro-Esclavo (Master/Slave) : Tener en cuenta este esquema cuando…

- Se realizan ***pocas actualizaciones en los datos pero muchas lecturas.*** Esto debido a que se puede escalar horizontalmente de manera fácil para gestionar más lecturas en diferentes nodos. E incluso ante una falla en el maestro los esclavos siguen procesando lecturas, pudiendo esperar a que se recupere el maestro para procesar escrituras o, se puede transformar un esclavo en maestro.

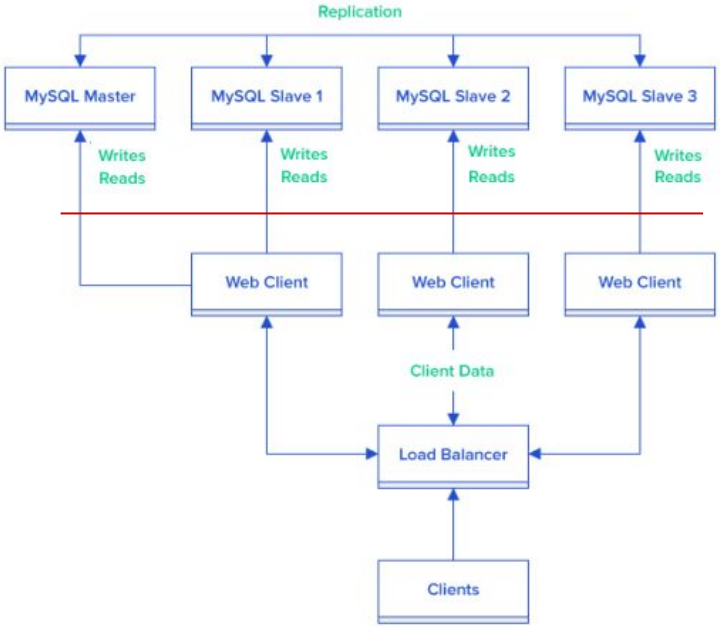
- Entonces no es un esquema óptimo para datos que tienen un tráfico de escritura denso. Aunque tener lecturas en los esclavos alivia el maestro, conclusión decimos que **no sirve para un escalamiento de escrituras**. Podrían tambien existir problemas de consistencia de lectura.

Tipos de **REPLICACIÓN**

Replicación Slave-Master-Slave

- Todas las réplicas tienen el mismo peso, todos los nodos aceptan escrituras. La pérdida de algún nodo no evita el acceso a los datos. Es fácil agregar nodos que mejoren la performance y distribuyan la carga.
- Cuando se escribe en un nodo esclavo se realiza la copia al nodo primario y del primario se replica al resto de los esclavos.

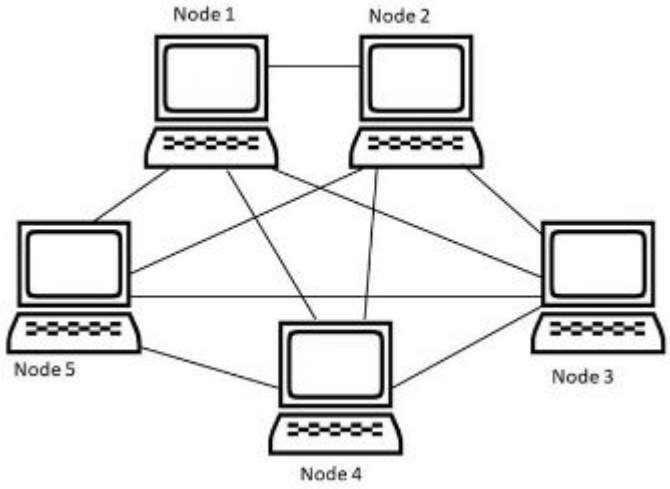
Actualización y consultas de datos en todos los nodos



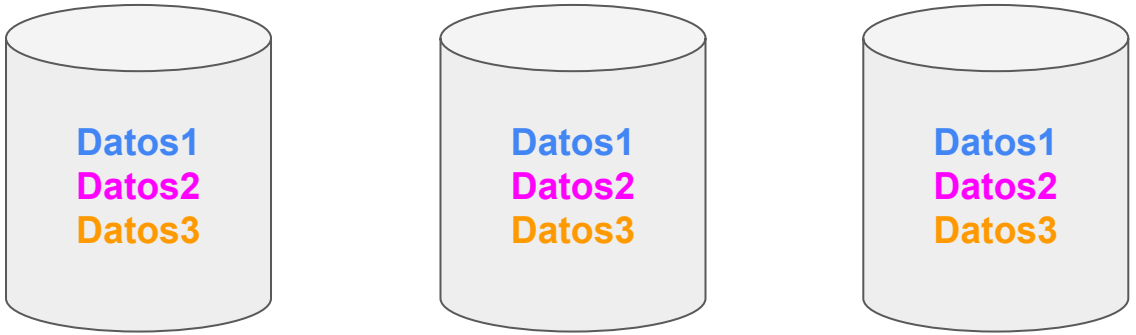
Tipos de **REPLICACIÓN**

Replicación entre pares (Peer to Peer o P2P)

- Todas las réplicas tienen el mismo peso, todas aceptan escrituras. La pérdida de algún nodo no evita el acceso a los datos.
- Es fácil agregar nodos que mejoren la performance y distribuyan la carga.
- *El principal problema es la consistencia*, ya que cuando se escribe en diferentes lugares se corre el riesgo de que dos clientes intenten actualizar el mismo dato a la vez, lo que se denomina conflicto escritura-escritura. Pueden existir problemas de consistencia de lectura.



Entonces…



Más allá del tipo de **REPLICACIÓN** que se implemente, lo que tendremos serán varias copias de los datos en los distintos servidores del cluster.

Sin embargo…

Cuando hablamos de **SHARDING** se distribuyen distintas partes del conjunto de datos en diferentes servidores que se denominan **particiones**.

SHARDING

- Cada partición es una base de datos independiente pero que, colectivamente forman parte de una única base de datos lógica.
- La carga se intenta balancear en forma homogénea entre todos los servidores disponibles.
- Es necesario asegurar que los datos que se acceden en conjunto están presentes también en forma conjunta en el mismo nodo. Pero… *no debemos olvidar que la distribución debe darse de forma tal que la carga de los nodos sea pareja*, para lograr una buena performance, mejorando velocidad de lectura y escritura.
- Muchas BD NoSQL ofrecen **Autoparticionamiento**, el mismo motor elige la estrategia de particionamiento, pero también podría realizarse dentro de la aplicación.

SHARDING (cont.)

- Permite escalar horizontalmente en aplicaciones que hacen uso intensivo de escrituras y lecturas, sin embargo no aporta demasiado a lograr capacidad de resistencia ante fallas.
  La ventaja parcial es que la falla solo la sufren los usuarios que estaban accediendo a un nodo que falló.
- El uso y la estrategia de particionamiento es una decisión que debe tomarse desde el momento del diseño del modelo de datos y desde el desarrollo, ya que una decisión de estrategia errónea es muy costosa de revertir.

Una última opción: Combinar!

- **La replicación y el particionamiento son estrategias que pueden combinarse.**
- Tendremos múltiples maestros, pero cada ítem de datos estará en solo un maestro.
- Un nodo podría ser maestro para determinados datos y esclavo para otros.
- Esta estrategia es bastante usada en bases de datos column-family.
- La combinación de ambas aumenta la tolerancia a fallas

- **Resumiendo…**

  El escalamiento horizontal es necesario en sistemas distribuidos, puede implementarse mediante Replica (varias opciones), Sharding o la combinación de ambas.

¿Qué soporta MONGODB?

AMBAS→ Réplica y Sharding

En MongoDB

- La replicación es de tipo *Master-Slave* y se realiza a través de estructuras denominadas **Replica Set.**
- Un Replica Set es un grupo de procesos de mongo que mantienen el mismo conjunto de datos. Proveen redundancia y alta disponibilidad.
- Un nodo (proceso mongod) se define como ***Primario*** y recibe todas las operaciones de escritura.
- Sólo puede haber un único nodo primario, por lo tanto se proporciona una consistencia estricta en la escritura.
- Los nodos secundarios replican el ***log de operación*** (oplog) del nodo primario y aplican las operaciones en sus datos *asincrónicamente*.
- Si el nodo primario no está disponible, se elige uno nuevo por ***votación*** entre los secundarios.

Cómo lo hacemos…

- Creamos un servidor levantando una instancia ***mongod***

mongod --replSet rs --port 27058 --dbpath c:\data*instancia1*…..

rs→ es el nombre que recibe el Réplica Set (RS)

Luego por ejemplo creamos 2 nuevas instancias de mongod

mongod --replSet rs --port 27059 --dbpath c:\data*instancia2*…..

mongod --replSet rs --port 27060 --dbpath c:\data*instancia3*…..

Cada instancia deberá tener su carpeta de datos asociada,
por eso en el ejemplo si ejecutamos tres instancias en una misma máquina -para simular un cluster-
crearemos antes las carpetas:
C:\data\instancia1, C:\data\instancia2, C:\data\instancia3

Luego configuramos el ReplicaSet por ejemplo de la siguiente manera:

1. mongo --port 27058 **cfg**={*_id*:"rs",

   members: [{*_id*:0, *host*:"127.0.0.1:27058“, *priority*: 2},

   {*_id*:1, *host*:"127.0.0.1:27059"},

   {*_id*:2, *host*:"127.0.0.1:27060“, *arbiterOnly*: true}]

   }

2. rs.initiate(cfg)

También se podría crear un RS con un solo nodo y luego agregar más nodos con la misma instrucción *cfg y el nuevo detalle en al array members o, utilizando rs.**add**(“127.0.0.1:27059”), rs.**add**(“127.0.0.1:27060”) ….*

Para que tome efecto la nueva configuración se ejecuta ***rs.reconfig(cfg).***

Con rs.conf() veremos la configuración del Replica Set.

Veamos algunos conceptos más en detalle:

- Al definir un nuevo miembro del réplica set, existen varias opciones a setear:

... members: [
{ **_id: <int>,**
host: <string>,
arbiterOnly: <boolean> ,
buildIndexes: <boolean>,
hidden: <boolean>,
priority: <number>,
tags: <document>,
slaveDelay: <int>,
votes: <number>}, ...
], ...

Veamos algunos conceptos más en detalle:

OPCION : arbiterOnly

- Es opcional de tipo booleano (true/false).
- El nodo árbitro puede votar en una elección de PRIMARY, pero no puede ser votado como PRIMARY.
- El nodo Arbiter no posee datos, sólo sirve para aportar un voto válido en una elección de PRIMARY dando **quórum**.
- Permite al replica set tener un número impar de miembros (recomendado).
- No tiene la sobrecarga (overhead) de un miembro Secondary que debe replicar los datos.
- Sólo de debe añadir un árbitro a replica sets con números pares de miembros.
- Si añade un árbitro a un conjunto con un número impar de miembros, en el Replica Set podrían haber de elecciones empatadas.

¿Qué es el quórum?

Es la **cantidad mínima de nodos que deben estar de acuerdo** para que una operación se considere válida y pueda completarse. Este concepto es fundamental para garantizar la **consistencia de datos** y la **tolerancia a fallos**. El nro. suele ser la mitad de nodos más uno. Por ejemplo en un cluster de tres nodos, el qurum será 2, en uno de 5 será 3…

Veamos algunos conceptos mas en detalle:

OPCION : priority

- Este atributo indica el grado de elegibilidad de un miembro del replica set para ser PRIMARY.
- Es un atributo opcional de tipo numérico con un rango entre 0 y 1000. El valor default es 1.
- Conviene especificar un valor más alto a los miembros más elegibes para ser PRIMARY.
- Por el contrario, conviene especificar un valor bajo para hacer a un miembro menos elegible.
- Las prioridades sólo son usadas para compararse con otros miembros del mismo replica set.
- Al cambiar el balance de prioridades de un replica set, se disparará una elección de primary.
- Si a un miembro se le asigna prioridad 0, el mismo nunca podrá ser primario.

  Un nodo secundario con prioridad 0, nunca podrá ser PRIMARY, ni votado como PRIMARIO. Igualmente mantiene una copia del conjunto de los datos, acepta operaciones de lectura y vota en las elecciones de primarios.
  Esta opción es particularmente usada en Implementaciones con Multi-data centers.

Veamos algunos conceptos mas en detalle:

OPCION : priority 0

Un nodo secundario con prioridad 0, nunca podrá ser votado como PRIMARIO. Igualmente mantiene una copia del conjunto de los datos, acepta operaciones de lectura y vota en las elecciones de primarios.

- Esta opción es particularmente usada en Implementaciones con Multi-data centers, donde las escrituras están más cercanas a un DC1, por lo que la latencia de escritura en el nodo secondary del DC2 no es la apropiada.
- Nodos con Prioridad 0 como Nodos Standby: cuando no es posible agregar nuevos miembros en un tiempo razonable, un nodo standby mantiene una copia actual de los datos y podría reemplazar a un miembro no disponible.
- Nodos con Prioridad 0 como Backup Server o Reporting Server: se configuran nodos con prioridad 0, para ser nodos en los que se realiza el backup del replica Set ó se destinan para servidor de reportes.

Veamos algunos conceptos mas en detalle:

OPCION : Hidden

- Este atributo en TRUE hace que un miembro del ReplicaSet sea invisible para las aplicaciones clientes.
- Los nodos con Hidden en TRUE deben tener configurado priority en 0, ya que no pueden ser PRIMARY.
- Votan en elecciones.
- Lo usual es usar nodos Hidden para tareas dedicadas como reporting o backups.
- No se tendrán en cuenta para la configuración de “*Read Preferences*”.
- Los miembros Slave *delayed*, deben ser miembros ocultos.

Veamos algunos conceptos mas en detalle:

OPCION : Slave Delayed

- Los nodos con esta propiedad seteada contienen una copia del conjunto de datos del replica set. Sin embargo, la copia que tienen refleja un **retraso en la actualización de los datos**.
  Por ejemplo, si la hora actual es 10.00AM y el NODO tiene un delay de una hora, este nodo delay tendrá operaciones anteriores a las 09:00AM.
- Un miembro Delayed debe ser un miembro de Prioridad 0 y además convendría que sea un miembro Hidden para que prevenir lecturas atrasadas desde una aplicación cliente.
- Vota en una elección.
- El tiempo de Delay debe ser igual o mayor que su ventana de mantenimiento y debe ser menor que la capacidad del **oplog**.

¿Qué es el oplog?

Es un log donde se registran todas las operaciones de escritura que se van haciendo. El método de replicación de MongoDB es asincrónico: los slaves o secondaries son los responsables de mantenerse actualizados. Consultan periódicamente el oplog de otro miembro (Por default el Primary) para saber si hay nuevas operaciones de escritura que deban realizar. El oplog se encuentra en la colección “oplog.rs” en la db “local”, (es la única db que no se replica a otros miembros.)

Veamos algunos conceptos mas en detalle:

OPCION : votes

- Es el número de votos que aporta un miembro cuando compite en la elección del PRIMARY.
- El número de votos por cada miembro del Replica Set puede ser 1 ó 0.
- Un réplica set permite tener hasta 50 miembros (version 3.0 en adelante), pero sólo 7 pueden ser miembros con Voto.
- Si tuviésemos un replicaset con más de 7 miembros, se deberá configurar votes en 0 para el resto de lo miembros que no votarán.
- Los miembros con votes = 0 pueden ser votados como PRIMARY, pero esto no los habilita a votar en elecciones.

Veamos algunos conceptos más en detalle:

OPCION : buildIndexes

- Indica si la instancia mongod debe crear índices o no en un miembro. El valor por default es TRUE.
- Sólo se puede configurar este valor cuando se agrega un NUEVO miembro al ReplicaSet, es decir NO se puede modificar el atributo buildIndexes de un miembro YA existente en el RS.
- No configurar este atributo en FALSO para un miembro que recibirá consultas de clientes.
- Al agregarlos se debe configurar priority en 0, sino mongodb retornará un error y como recomendación poner el nodo Hidden, para que las aplicaciones clientes no lo vean.

El atributo buildIndexes configurado en FALSE es utilizado en: para realizar backups utilizando mongodump, así ese miembro no recibe consultas y no invierte tiempo en la creación y mantenimiento de índices sobrecargando el sistema. Igualmente si se configura en FALSE, los nodos secundarios crearán índices para el campo _id, para facilitar las operaciones requeridas por la replicación.

Funcionamiento…

El Primary del Replica Set es el único nodo que puede recibir escrituras, por lo que es necesario que, en caso que este deje de estar disponible, pueda elegirse otro **miembro apto** y transformarlo en Primary mediante votación.
Si bien el Primary, por default es el único miembro del que se puede leer, esto puede ser cambiado especificándolo en la Read Preference.

Para que un Primary continúe siéndolo, deberá poder comunicarse al menos con la mayoría de los nodos (la mitad + 1 de los miembros)

Cuando un Secondary detecta que el Replica Set no tiene un Primary y cumple las condiciones para serlo, llama a elección postulándose como Primary. Si algún otro miembro votante conoce un secondary más apto para ser Primary (que esté más actualizado en cuanto a las operaciones del oplog o que tenga mayor prioridad), anulará la votación. En caso de que consiga una mayoría de votos positivos, el miembro pasará a ser Primary.

Funcionamiento… opciones:

La opción **writeConcern** determina cuando una operación de escritura es considerada exitosa. Puede especificarse para todo el Replica Set (cfg.settings…), para cada operación o para cada conexión.

{ w: , j: , wtimeout: }

Si **w** es un número, indica la cantidad de nodos que deben haber escrito el dato en memoria para que se confirme la escritura.

Naturalmente.. w:1 significará que se confirmará cuando el Primary grabe el dato en memoria,
w:2 implicará que, aparte del Primary, un Secondary deberá poseer el dato.

Funcionamiento… opciones:

La opción **writeConcern** si **w** no es un número, con ***majority*** la confirmación de escritura se dará cuando la mitad más uno de los nodos posean el dato. Este write concern garantiza que, con la escritura confirmada, si el Primary se cae y se elige uno nuevo, este poseerá el dato.
Con ***tag*** podemos tener un mayor control sobre cuándo recibir la confirmación de escritura. Por ejemplo, teniendo servidores de un Replica Set distribuidos entre 2 data centers, puedo recibir la confirmación cuando por lo menos un nodo de cada data center haya guardado el dato. Para esto al definir cada nodo le indico un tag.

j (journal) Confirma la escritura cuando el servidor escribe el journal en disco. Es decir que, aunque un servidor se caiga antes de que se escriba el documento en disco, sí se encontrará el registro de qué operación debe hacerse, por lo que no se perderá el dato

wtimeout : Forzará a la operación a retornar un error si no es confirmada antes del tiempo en milisegundos especificado. Este error no significa que el dato no se haya escrito en algún nodo, y tampoco implica hacer un rollback en los nodos en los que se haya escrito.

Funcionamiento… opciones:

La opción de query **readConcern** para replica sets determina qué datos se deben devolver para un query. Puede ser usado con los comandos: find, aggregate, distinct, count, etc.

Algunos niveles posibles son: local, available, linearizable y majority

- **local:** Devuelve los datos más recientes de la instancia a la que se le realiza la consulta, sin garantizar que haya sido escrita en una mayoría, *por esto los datos pueden llegar a ser rollbackeados*.
- **available**: Devuelve los datos más recientes de la instancia a la que se le realiza la consulta, sin garantizar que haya sido escrita en una mayoría, *por esto los datos pueden llegar a ser rollbackeados*. Para colecciones no shardeadas, el comportamiento de este read concern es idéntico a local.
- **majority:** El query devuelve los datos más recientes de la instancia que haya sido escrita en una mayoría de los miembros del Replica Set. *Esto significa que los datos son durables ante la caída de los nodos.*
- **linearizable:** El query devuelve datos que hayan sido escritos a una mayoría de los nodos previo a iniciar la operación de lectura. El query puede esperar a que escrituras que estén ejecutándose de forma concurrente terminen de propagarse a la mayoría de los nodos previo a devolver los resultados.

Funcionamiento… opciones:

La opción **Read preferences** indica de qué nodo se leerá el dato que estemos pidiendo. Las opcioens disponibles son las siguientes:

- **Primary** (Default): Lee únicamente del Primary. De no haber un Primary disponible, retornará error.
- **Primary Preferred**: Leerá preferentemente del Primary, pero si momentáneamente no hay ninguno leerá de un Secondary.
- **Secondary**: Leerá únicamente de un Secondary. De no haber Secondaries, devolverá un error.
- **Secondary Preferred**: Leerá preferentemente de un Secondary, pero si sólo está el Primary levantado, leerá de este.
- **Nearest**: Leerá del nodo que tenga menor latencia de red.

*Utilizando el **shell de mongo**, por default nos dejará leer únicamente del Primary, por lo que, al querer leer de un Secondary, deberemos ejecutar **rs.slaveOk()** indicando que aceptamos los datos del Secondary, sabiendo que puede no tener las últimas escrituras.*

¿Qué es el journal?

Para proveer durabilidad ante el caso de la caída repentina de una instancia, MongoDB usa archivos **journal** en el disco como un write ahead log, en los que, al llegar una escritura, se registrará la operación a realizarse antes de persistir los datos reales en disco.

La escritura en disco de los datos del journal se realiza con una frecuencia mucho mayor que la de los datos reales. Si la instancia falla antes o mientras se está escribiendo datos a disco, se tendrán en el journal las operaciones que se deberán realizar para mantener la integridad y poseer escrituras más recientes.

La estrategia de mongo llamada WiredTiger escribe el journal en disco en las siguientes condiciones:

- Cada 100 milisegundos
- Si la operación de escritura tiene j: true en el write concern

MongoDB usa un tamaño de journal de 100 MB, es decir se crea un nuevo archivo de journal aproximadamente cada 100 MB de datos. Al crear un nuevo archivo de journal, se escribe el anterior en el disco.

Diferencia entre oplog y journal

Oplog almacena transacciones de alto nivel que modifican la base de datos (las consultas no se almacenan, por ejemplo), es decir insertar un documento, actualizar, etc.

Oplog se mantiene en el primario y los secundarios chequearán periódicamente al maestro para obtener las operaciones recién realizadas (desde el último sondeo).

A veces, las operaciones se transforman antes de almacenarse en el oplog para que sean idempotentes (y se puedan aplicar de forma segura muchas veces).

El journal, por otro lado, se puede activar / desactivar en cualquier nodo (primario o secundario) dependiendo de la versión y es un registro de bajo nivel de una operación con el fin de permitir la recuperación ante bloqueos.
Se utilizan puntos de control que se crean cada 60 segundos.
En caso de que ocurra un bloqueo entre los puntos de control, se pueden perder algunos datos.