PEC3. Sistemas distribuidos. Kepa Sarasola Bengoetxea

1. Consistent hashing

Antes de entrara a explicar que es y cómo funciona el Consistent hashing, es conveniente aclarar ciertos conceptos que participan de esta metodología.

Hash Tables

Una tabla hash es una estructura de datos que asocia llaves con valores (key-value), su principal función es la de facilitar la búsqueda de elementos en un entorno de datos, ya que permite el acceso a los elementos de datos a partir de una clave generada usando un algoritmo o una función llamada *hash*. De esta manera transforma la clave del dato en un *hash* identificativo del dato, este *hash* identifica el dato dentro de la tabla con lo que su posterior búsqueda se facilita.

La forma de implementar esta operación es pidiendo la llave y el valor, para con ambos elementos hacer la inserción del dato:

1. Para almacenar un elemento en la tabla de *hash* se ha de convertir su valor aplicando una función.
2. El resultado de la función ha de mapearse al espacio de direcciones del vector mediante la función módulo, así se obtiene un índice válido.
3. El elemento se almacena en la posición de la tabla obtenida.
4. Posiblemente puede haber colisiones entre los elementos debido a que ya existe un elemento en la posición de la tabla obtenida. Una solución utilizada para evitar esto es la utilización de una lista (*bucket*), que contiene todos los objetos que comparten un índice determinado

Aunque las búsquedas dentro de los *bucket* o depositos son lineales, una tabla *hash* del tamaño adecuado debe de tener una cantidad manejable, más bien pequeña de objetos por *bucket*, para que el acceso a los datos pueda ser casi inmediata.

Escalado horizontal, hash distribuido (distributed hash table)

El escalado horizontal o *hash* distribuido responde a la necesidad de dividir una tabla *hash* en varias partes alojadas en diferentes servidores, con el objetivo de que se puedan manejar tablas de *hash* de tamaños grandes, aplicando un escalado horizontal.

La aplicación del *hash* distribuido incorpora un nuevo problema al planteamiento del *Hash Table*, ya que de alguna manera hay que incorporar en el dato la localización del servidor en el que está almacenado. La manera más simple de hacer esto es calculando el módulo del número de servidores e incorporando este dato al *hash* alojado en la *tabla hash*. Para la recuperación de un dato, el cliente primero calcula el *hash* aplicando el módulo de N (número del servidor) localizando en que servidor se encuentra este.

Es importante subrayar que para que el uso del *hash* distribuido se realice de manera correcta es necesario que la función de *hash* utilizada para la distribución de claves debe de ser la misma en todos los clientes.

El problema de este sistema de *hash* distribuido se da cuando alguno de los servidores falla o se incorpora un nuevo servidor al sistema, es decir cuando el número de servidores

cambia, ya que como hemos comentado anteriormente el módulo del número de servidores del sistema es utilizado para localizar en que servidor están las tablas que contienen el dato. Esto significa que un cambio en el número de servidores disponibles en el sistema supone un cambio de las claves de todos los datos del sistema. Lo cual conllevaría que las claves no se encontrarían, se generarían errores y que posiblemente se tengan que rehacer los datos originales.

Consistent hashing

El *Consistent Hashing* se plantea para dar una solución al problema planteado en el caso de las tablas de *hash* distribuidas, ya que se ve la necesidad de un sistema distribuido que no dependa directamente del número de servidores disponibles. Este sistema fue descrito por primera vez por David Karger en el MIT en 1997 como un método de distribuir solicitudes de información entre un número indeterminado de servidores.

Este es un esquema de *hash* distribuido que opera independientemente del número de servidores u objetos en una tabla de hash distribuida asignándoles a estos una posición en un circulo abstracto o anillo de hash.

Uso de Chord

Chord es un protocolo de superposición de enrutamiento que ubica nodos y objetos dentro de una red. Cada nodo tiene un ID asociado a su posición lógica en una estructura circular cuyo valor puede estar entre 0º y 360º, además cada nodo tiene una clave que se obtiene a partir de una función *hash*. De esta manera los objetos pueden informar en que nodos se encuentran ubicados.

Básicamente se trata de asignar un punto en el borde de un circulo, es decir, un punto que tendrá un angulo entre 0º y 360º, a los *hashes* que componen la tabla de *hash*. En el mismo circulo, también en el borde se colocan los servidores, asignándoles también una posición al igual que a los datos.

Dado que en este momento tenemos tanto las claves para los objetos como para los servidores en el mismo circulo, podemos definir una regla para asociar las claves con los servidores, definiendo así que cada clave de objeto pertenecerá al servidor cuya clave esté más cercana en el sentido contrario a las agujas del reloj.

¿Que ocurriría si un servidor deja de funcionar o se añade un nuevo servidor?

En caso de que un servidor cause baja, en primer lugar se eliminarían las referencias al servidor del anillo de *hash*, esto supone que las claves de objeto que antes estaban a la derecha de este servidor pasarán a estar en alguno de los servidores que tiene el sistema y por lo tanto se realojarán dentro del anillo, ocupando nuevas posiciones. Pero que ocurrirá con las claves de objeto que estaban en los servidores que se mantienen y que no han fallado? Pues no ocurriría nada, ya que estas claves de objeto mantienen sus posición con respecto a los servidores dentro del círculo.

Por supuesto algo similar ocurre si añadimos un nuevo servidor al sistema, este primero se situará en un punto dentro del circulo o anillo *hash*, las claves de datos que se asignen a este nuevo servidor tendrán que ser modificados para que se realice esta asignación, pero el resto de las claves de valor no se modificarán.

Casos de uso

Como ya hemos comentado hasta ahora este sistema intenta cubrir unas necesidades de tratamiento de una gran cantidad de datos con una alta disponibilidad de estos. Vemos ahora tres casos en los que se usa la tecnología Consistent Hashing:

- Caches distribuidas (Memcached, Redis, etc.) se utilizan para acelerar las aplicaciones que requieren de consultas a bases de datos, almacenando los resultados de estas consultas para poder devolver la información requerida de manera más rápida.
- Bases de datos NoSQL (Cassandra, Riak)
- Esta técnología tambien está presente en muchos servicios de Amazon como S3, a traves de Amazon Dynamo.

2. Acoplamiento temporal y espacial, 4 combinaciones:

En términos de sistemas distribuidos, el concepto de comunicación indirecta se aplica cada vez más a los paradigmas de comunicación. La comunicación indirecta se define como la comunicación entre entidades en un sistema distribuido a través de un intermediario, sin que exista un acoplamiento directo entre el emisor/es y el receptor/es. La literatura se refiere a dos propiedades clave derivadas del uso de un intermediario:

- Desacoplamiento de espacio, en el que el remitente no conoce o necesita conocer la identidad del receptor/es y viceversa. Debido a este desacoplamiento, el sistema tiene mucha libertad para cambiar de participantes en las comunicaciones, ya que tanto los emisores como los receptores se pueden reemplazar, actualizar o replicar.
- Desacoplamiento de tiempo, en el que el emisor/es y el receptor no necesitan existir en el mismo momento para comunicarse. Este sistema es muy usado en sistemas el el que existe una alta tasa de volatilidad.

La comunicación indirecta se utiliza en sistemas distribuidos donde se prevean cambios, como entornos móviles donde pueden darse multitud de conexiones y desconexiones a una red. La siguiente tabla, obtenida del libro de referencia de la asignatura, nos da una información resumida:

	Acoplamiento Temporal	Desacoplamiento Temporal
Acoplamiento espacial	**Propiedades:** Comunicación dirigida hacia un receptor o receptores conocidos que deben de existir en ese momento.	**Propiedades:** Comunicación dirigida hacia un receptor o receptores determinados. Tanto el emisor como el receptor pueden tener tiempos de vida independientes.
Desacoplamiento espacial	**Propiedades:** el remitente no necesita conocer la identidad del receptor/es, ni este del remitente, pero deben existir en el momento de la comunicación.	**Propiedades:** el remitente no necesita saber la identidad del receptor/es. Remitente y receptor/es pueden tener tiempos de vida independientes.

Ejemplos para estas cuatro combinaciones:

- **Acoplamiento espacial y temporal:** en el mundo no digital una reunión presencial o una llamada telefónica son buenos ejemplo de esta acoplamiento espacial y temporal, ya que es necesario que ambos agentes se conozcan y coincidan en el tiempo de realización de la comunicación. En el mundo digital una video llamada puede ser un ejemplo de esta situación, ya que también en este caso es necesario que tanto el emisor como el receptor sean conocidos y coincidan en el tiempo para que esa video llamada se pueda dar.

- **Desacoplamiento espacial y acoplamiento temporal:** un ejemplo de este sistema son las emisiones en directo desde cualquier canal de video como por ejemplo youtube.com o twitch.tv donde existe un emisor que realiza una emisión y en las que es necesario que el receptor se encuentre en el mismo instante temporal recibiendo la señal de video en este caso. Para realizar este tipo de comunicación no es necesario que el receptor y el emisor se conozcan de antemano, simplemente el emisor emite la señal y es receptor escoge una señal entre tantas de las que le ofrece el servicio *live* para recibirla.

- **Acoplamiento espacial y desacoplamiento temporal:** un ejemplo no digital de este sistema es el del correo ordinario, es decir una carta que tiene un emisor y un receptor que se conocen, pero el momento temporal en el que el emisor escribe la carta y el momento temporal en el que el receptor la lee, necesariamente no son los mismos. En cuanto al mundo digital, los sistemas de mensajería como WhatsApp o Telegram, utilizan este sistema, en el que los emisores y receptores han de conocerse necesariamente, pero no han de estar conectados en el mismo instante temporal para recibir el mensaje. Los mensajes se guardan en una cola que se descarga cuando el receptor se conecta al sistema.

- **Desacoplamiento espacial y temporal:** un foro de discusión público y online es un ejemplo en el que se cumple tanto el desacoplamiento espacial como el temporal, ya que cuando un emisor escribe el mensaje no tiene por que conocer a los receptores de este, es decir a los que van a interactuar con este y tampoco tiene porque coincidir temporalmente con ellos para que esta comunicación se de forma satisfactoria.

3. RPC vs API HTTP REST

¿Que es RPC?

RPC son las siglas de Remote Proceduce Call, (llamada a procedimiento remoto). Es una tecnología que regula la comunicación entre procesos, es decir, el intercambio de información entre procesos de sistema. Se define como un mecanismo síncrono que transfiere el flujo de control y los datos entre dos espacios de direcciones a través de una red de banda estrecha como llamada a proceso.

El proceso de comunicación con RPC consta del envío de parámetros y el retorno de un valor de función. A menudo se procesan muchas solicitudes en paralelo. RPC tiene como objetivo cumplir con el principio de transparencia es decir, las llamadas remotas deberían de implementarse de igual manera que las llamadas locales.

Actualmente las llamadas RPC se utilizan en muchos ámbitos, por ejemplo en los servicios web mediante el protocolo XML-RPC para la llamada a funciones remotas mediante HTTP. Otros campos de aplicación son los clústeres de ordenadores, las redes descentralizadas y las cadenas de bloques (por ejemplo las criptomonedas).

Un caso de uso especial se da en el mundo Linux que usa NFS como sistema de archivos en red, en este caso, se utiliza RPC entre cliente y servidor para montar el conjunto de archivos de un ordenador remoto en ordenador local.

Ventajas de RPC
El protocolo RPC gestiona la comunicación entre procesos de manera fiable y requiere un tiempo de procesamiento relativamente corto, facilitando así la programación de procesos de comunicación entre ordenadores remotos. Además permite una modularización coherente, ya que los procesos pueden distribuirse entre nodos del sistema, haciendo el sistema más eficiente. Otra ventaja es la excelente escalabilidad de las arquitecturas cliente-servidor implementadas con este sistema.

El protocolo RPC es excelente para la gestión de procedimientos o comandos

Inconvenientes de RPC
Su mayor inconveniente es que no existe un estándar unificado, existen diferentes implementaciones, la mayoría específica por cada empresa o proveedor y no suelen ser compatibles entre sí.

Los niveles de transferencia de los sistemas basados en RPC conllevan una perdida de velocidad, además del uso de recursos por parte de diferentes entornos, como en el caso del cliente-servidor vuelve más complejo el sistema.

Existen muchas causas que hacen el sistema susceptible a sufrir errores, como la división en diferentes instancias de procesamiento, retrasos o interrupciones o la ralentización del servidor.

También es una clara desventaja su complejidad, ya que abarca muchos y diferentes funciones, ya que está pensado para dar solución a casi cualquier necesidad.

API HTTP REST
REST viene de "REpresentational State Transfer", es un protocolo que se basa en realizar la transferencia del estado, es decir, se trata de un protocolo que no tiene estado, el cliente es quien se tiene que ocupar de enviar el estado al servidor en cada llamada, como por ejemplo un *token* para la autenticación en el sistema, etc.

REST es un enfoque con un estilo de operación muy restringido, en el que los clientes usan las operaciones GET, POST, PUT, DELETE, OPTIONS y PATCH para manipular recursos que son representados habitualmente en formatos XML o JSON.

La diferencia principal con el protocolo RPC es que se habla más de recursos que de procesos, en consecuencia, se trata de recursos que los clientes pueden consumir. Cuando se crea un nuevo recurso, se crea un nuevo punto de acceso mediante el cual puede ser accedido o actualizado mediante una URL. Como ya hemos dicho, el servidor

no mantiene el estado del cliente, por lo que ha de ser este quien mediante la URL indique cual es su estado.

En el contexto de la internet actual, podemos decir que es el protocolo más utilizado para el uso de los servicios web. Al igual que Amazon, otros servicios como por ejemplo Twiter o Facebook han impulsado de manera clara este protocolo

Ventajas
Las API REST son una opción excelente para modelar el dominio, es decir, usar recursos o entidades, haciendo que CRUD este disponible para todos sus datos.

El no mantener un estado es una desventaja clara sobre todo para el cliente, ya que se tiene que ocupar de enviar este en cada llamada, pero supone una gran ventaja para el lado del servidor ya que no tiene que guardar los datos de los clientes que acceden a él, ni ninguno de sus estados.

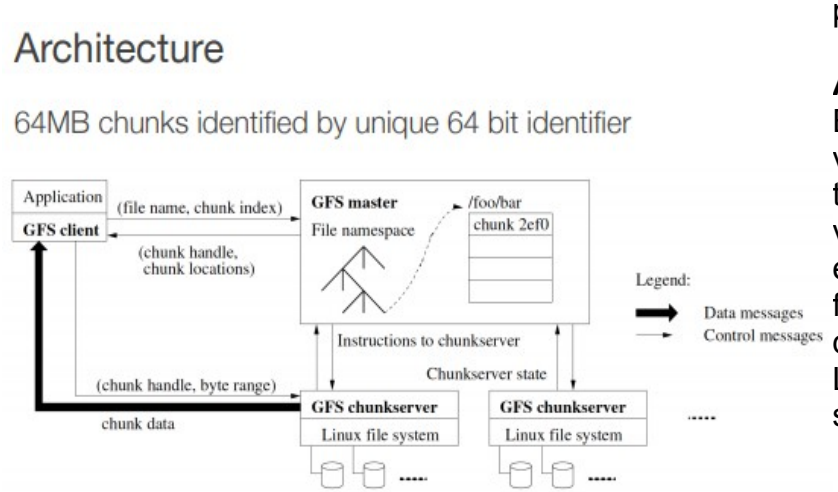
Otra ventaja es que la escalabilidad del sistema es muy grande ya que es mucho más sencillo trabajar con varios nodos al no tener que manejar estados. En esta arquitectura existe la separación cliente/servidor, es decir, los dos sistemas son independientes, tanto en cuanto a la tecnología como al lenguaje utilizado para su desarrollo.

Desventajas
REST esta pensado y basado en dar respuesta a un sistema CRUD, aunque en principio no tiene porque verse limitado a esto, si es cierto que su uso parece muy encaminado a ello. Si necesitamos de opciones más amplias, el protocolo REST puede quedarse corto.

Esta basado en una comunicación síncrona y que funciona exclusivamente sobre el protocolo HTTP. Además tiene el problema de tener que documentar la API para saber las URLs y los métodos y parámetros que están disponibles.

4. Describe Google File System y la Bigtable

Google File System (GFS, GooFS o GoogleFS), es un sistema de archivos distribuido propietario desarrollado por Google, que soporta toda su infraestructura informática de procesamiento de información en la nube. Está especialmente diseñado para proveer eficiencia, fiabilidad de acceso a datos usando sistemas masivos de clúster de procesamiento en paralelo.



Arquitectura
El clúster GFS se compone de varios nodos, divididos en dos tipos, un único nodo maestro y varios *chunkservers* (cuya función es la de almacenar fragmentos de ficheros). Normalmente cada uno de estos nodos está basado en Linux y ejecuta un proceso de servidor a nivel de usuario.

Los archivos se dividen en fragmentos de tamaño fijo, estas porciones o fragmentos son identificados por un *chunk* único e inmutable de 64 bits asignado por el nodo maestro en el momento de creación del fragmento, el nodo maestro guarda las asignaciones que va haciendo. El nodo maestro ademas de las referencias de los fragmentos se encarga de almacenar los metadatos del sistema de archivos, esto incluye el espacio de nombres, la información de control de acceso, la asignación de archivos a fragmentos y las ubicaciones actuales de los fragmentos.

Los servidores de *chunk* almacenan los fragmentos en discos locales como archivos de Linux. Cada fragmento es replicado al menos en otros tres servidores de *chunk* en la nube para tener un mínimo de redundancia, aunque puede haber fragmentos que por su alta demanda puedan ser replicados en un número mayor de servidores.

Los clientes interactúan con el maestro para las operaciones de metadatos que ya hemos definido, pero una vez obtenidos estos, la comunicación es directa con los *chunkservers,* lo cual evita que se formen cuellos de botella en el nodo maesrtro.

Tener un sólo nodo maestro simplifica enormemente el diseño y permite al nodo maestro tomar decisiones respecto de la replicación y colocación de los fragmentos de ficheros de una manera más inteligente. Sin embargo existe el riesgo de provocar un cuello de botella debido a las operaciones de lectura y escritura sobre estos ficheros, para ello se asegura que los clientes nunca realicen operaciones de lectura/escritura directamente sobre el nodo maestro.

Escalabilidad
Respecto a la escalabilidad del sistema, la división entre un nodo maestro y multitud de nodos *chunkserver* permite al sistema escalar de manera horizontal en el caso de los *chunkserver*, tanto el número de servidores como la replicación de los fragmentos de ficheros.

BigTable
Google BigTable es un mecanismo de almacenamiento de datos no relacional, distribuido y multidimensional construido sobre las tecnologías de almacenamiento patentadas de Google para la mayoría de sus aplicaciones. Esta construido sobre Google File System, Chubby Lock Service, y algunos otros servicios y programas de Google, y funciona sobre 'commodity hardware' (sencillos y baratos PCs con procesadores Intel).

Arquitectura
BigTable no se puede definir como una base de datos no relacional, se define mejor como un mapa ordenado y multidimensional, disperso, distribuido y persistente. BigTable asigna dos valores de cadena arbitrarios (clave de fila y clave de columna) y marcas de tiempo (por lo tanto realiza un mapeo tridimensional) en una matriz de bytes no interpretada.

Esta diseñado para escalar en el rango de petabytes en “cientos o miles de máquinas”, pudiendo asumir el sistema el aumento del número de nodos sin tener que reconfigurarlo.

Las tablas se dividen en varias tabletas (rango de filas de la tabla): los fragmentos de la tabla se dividen en grupos de filas para que cada tableta no tenga un tamaño muy grande. Cuando el tamaño de una tableta se incrementa de manera importante, el servidor puede

deshacerse de otras tabletas o dividir esta en otras tabletas que se alojarán en otros servidores.

Las claves de columna se agrupan en conjuntos llamados familias de columnas, que forman la unidad básica de control de acceso. Toda la información almacenada en una familia de columnas suele ser del mismo tipo. El control de acceso y la contabilidad de disco y memoria se realiza a nivel de familia de columnas.

Cada celda de una tabla puede contener varias versiones de los mismos datos, estas versiones están indexadas por marcas de tiempo, que se representan mediante números enteros de 64 bits. Las versiones se almacenan en orden decreciente de la marca de tiempo, de tal manera que las versiones más recientes se puedan leer primero.

El formato de archivos Google SSTable se utiliza internamente para almacenar datos de BigTable. Cada SSTable contiene una secuencia de bloques (normalmente de un tamaño de 64Kb) y un índice de bloque que se utiliza para localizar los bloques que contiene. De esta manera se puede realizar una búsqueda consultando el índice en una sola operación de disco. BigTable utiliza el sistema de ficheros distribuido de Google GFS para almacenar archivos de registro y datos.

BigTable también utiliza un servicio de bloqueo distribuido llamado Chubby, consistente en un sistema de cinco réplicas activas, una de las cuales una será el maestro y atenderá a las solicitudes. Chubby utiliza el algoritmo de Paxos para mantener sus replicas consistentes en prevención de fallos, asegurándose así que al menos un servidor maestro existe y está activo.

Escalabilidad
Bigtable es una tabla poco poblada que puede escalar hasta miles de millones de filas y miles de columnas, lo que te permite almacenar terabytes o, incluso, petabytes de datos. Se indexa solo un valor de cada fila; este es conocido como la clave de fila. Bigtable es ideal para almacenar cantidades grandes de datos con una sola clave y con una latencia muy baja. Admite una capacidad alta de procesamiento de lectura y escritura con baja latencia, y es la fuente de datos ideal para las operaciones de MapReduce.

En general, el rendimiento de un clúster se escala de forma lineal a medida que se le agregan nodos. Por ejemplo, un clúster SSD con 10 nodos, puede admitir hasta 100,000 filas por segundo para una carga de trabajo típica de solo lectura o de solo escritura.

Tiempo de aplicación del escalado
Después de aumentar la cantidad de nodos en un clúster para escalar verticalmente, pueden pasar hasta 20 minutos bajo carga antes de que veas una mejora significativa en el rendimiento del clúster. Cuando disminuyas la cantidad de nodos en un clúster a fin de reducir la escala, intenta no reducir el tamaño del clúster más de un 10% en un período de 10 minutos para minimizar los picos de latencia.

5. a) Arquitectura, propósito y características de Redis

Redis es un almacén de estructura de datos en memoria de código abierto que se utiliza con frecuencia para implementar cachés y bases de datos clave-valor (típicamente NoSQL). Su nombre proviene de las iniciales de Remote Dictinary Server, un tipo de servidor apto para su uso como memoria rápida de datos.

En un servidor Redis los datos no se guardan en el disco duro, sino en la memoria principal, esto permite que Redis funcione como memoria cache o como una unidad de memoria principal.

Cada entrada a la base de datos recibe asignada una clave cuyo objetivo es que se puedan localizar los datos fácilmente cuando se necesiten, la información está pues disponible de manera directa, sin que haya que realizar consultas relacionales.

La estructura de datos típica de Redis está formada por strings, tanto para las claves como para los valores, aunque para estos últimos admite también otro tipo de datos como hash, list, set, sorted set, bitmap, hyperLog y Stream.

Redis es una herramienta que permite acceder a los datos de una manera rápida, gracias a la replicación de los datos, la velocidad de lectura aumenta aún más ya que los datos pueden solicitarse desde diferentes instancias.

Arquitectura

Un clúster de Redis se compone de nodos idénticos, la arquitectura se compone de:

- **Una ruta de administración** que incluye el administrador del clúster, el proxy y la interfaz de usuario API REST. El administrador del clúster es el responsable de orquestar el clúster, colocar fragmentos de bases de datos en el resto de nodos del sistema y detectar y mitigar fallos.

- **Una ruta de acceso a datos** que se compone de accesos a nodos maestros y esclavos. Los clientes realizan operaciones de datos en los nodos maestros, mientras que estos mantienen los nodos esclavos replicándolos en memoria

Redis está escrito en código C optimizado, se encuentra basado en la estructura de *Tablas Hash* (estructura de datos tipo diccionario), siendo de este modo una estructura de valores clave en memoria de rápido acceso y de código abierto.

5. b) Mejora de la escalabilidad

Cada clúster de Redis puede contener varias bases de datos, que representan datos que pertenecen a una sola aplicación o servicio y está diseñado para escalar a cientos de bases de datos por clúster para proporcionar modelos multiusuario.

Redis se basa en la fragmentación (*sharding*) de la información (que se replica siguiendo el esquema maestro-esclavo) en los nodos del clúster. La fragmentación es transparente para las aplicaciones de Redis, cada base de datos se puede fragmentar en muchos o pocos fragmentos de Redis (el número es configurable) que pueden estar contenidos en uno o varios clústers.

Esta opción de fragmentación de la base de datos es la que permite a Redis escalar de una manera óptima, por ejemplo, en un clúster Redis que puede contener hasta 500 nodos (Redis v5.0.6 o superior) se puede optar por dar respaldo a 83 fragmentos (con un nodo principal y 5 réplicas por fragmento) o hasta 500 fragmentos (sin utilizar replicas).

La principal ventaja es que un fragmento se puede a su vez dividir en más fragmentos para escalar el rendimiento y mantener latencias muy bajas, reestructurando el sistema sin tiempos de inactividad.

5. c) Qué es el sharding y cómo lo usa Redis

El sharding es una técnica de fragmentación de la base de datos con el objetivo de optimizar su rendimiento y mejorar la escalabilidad de esta en un sistema distribuido.

Es un patrón de arquitectura de base de datos que divide una sola base de datos en tablas más pequeñas conocidas como *shards* (fragmentos), cada una almacenada en un nodo por separado. Cada partición de la base de datos se conoce como un fragmento lógico y su almacenamiento dentro de un nodo se conoce como fragmento físico.

Dividir una base de datos en varios fragmentos ayuda con la escalabilidad y la disponibilidad, además es una técnica repetible que permite seguir escalando horizontalmente la base de datos de manera indefinida. El *sharding* también mejora la disponibilidad de las bases de datos, ya que si una máquina falla, sólo ese fragmento pasará a ser inaccesible, el resto de fragmentos seguirán funcionando con normalidad.

En este punto conviene matizar, ya que aunque en principio puede parecer que el *sharding* es igual a particionar una base de datos, *sharding* se puede considerar un tipo especial de particionamiento. El particionamiento se define como cualquier división de una base de datos en partes distintas, que puede plantearse en modo horizontal (diferente número de filas) o vertical (todas las filas, pero dividiendo por subconjuntos de columnas).

Sharding es un caso especial de partición horizontal, es decir la tabla se trocea con todas las columnas y se van haciendo fragmentos que contienen rangos de filas.

Sharding en Redis
El *sharding* es una técnica esencial para mejorar la escalabilidad y la disponibilidad de las implementaciones Redis. En principio, en un cluster de Redis solamente se maneja un único fragmento que puede ser replicado horizontalmente agregando nodos hasta un total de cinco.

El sharding en Redis puede implementarse de varias maneras:

- **Sharding por parte del cliente:** los clientes seleccionan la instancia de Redis adecuada para leer o escribir una clve en particular
- **Sharding asistido por proxy:** se utiliza un poxy para manejar las solicitudes y enviar las solicitudes a la instancia de Redis adecuada
- **Enrutamiento de consultas:** la consulta se envía a una instancia aleatoria que asume la responsabilidad de redirigir al cliente a la instacia de Redis adecuada.