Bases de Datos NoSQL: Escalabilidad y alta disponibilidad a través de patrones de diseño

Ing. Matías Javier Antiñanco

Director: Mg. Javier Bazzocco
Codirector: Dra. Silvia Gordillo

Trabajo Final Integrador para obtener el grado de Especialista en Ingeniería de Software

Facultad de Informática
Universidad Nacional de La Plata

- Diciembre de 2013 –

Indice General

Indice de Ilustraciones

Indice de Tablas

Tabla de Abreviaciones

Abreviación	Significado
ACID	Atomicity, consistency, isolation, durability
API	Application programming interface
BASE	Basically Available, Soft-state, Eventual consistency
BI	Business intelligence
DHT	Distributed hash table
EAV	Entity-Atribute-Value
FOSS	Free and open source-software
GFS	Google File system
HA	High availability
LRU	Least recently used
NOSQL	Not only SQL
RDBMS	Relational database management system
RYOW	Read Your Own Writes
SPOF	Single point of failure

NOTA En el presente documento se encontrará una mezcla de términos en lenguaje castellano y términos en lenguaje Ingles, se hace esta convención debido a que algunos términos no tienen una traducción exacta y reemplazarlos por palabras del castellano podría introducir algún tipo de confusión.

1. Introducción

1.1 Contexto

El término NoSQL fue utilizado por primera vez en 1998 por Carlo Strozzi para referirse a una base de datos open-source que omitía el uso de SQL, pero sí seguía el modelo relacional. Dicha base de datos utilizaba como interfaz de usuario lenguaje shell de UNIX. Este uso del término NoSQL no tiene relación con el concepto actual, referido al movimiento NoSQL. [1]

El término fue re-introducido por Eric Evans, empleado de Rackspace, cuando Johan Oskarsson de Last.fm pregunto en IRC cual sería un buen nombre para la primera reunión de bases de datos distribuidas de código abierto que estaba organizando. Según palabras del mismo Eric Evans *"fue una de las 3 o 4 sugerencias que arroje en un lapso de 45 segundos sin pensar"* [2]. El nombre intentaba recoger el número creciente de bases de datos no relacionales y distribuidas que no garantizaban ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), consideradas atributos claves en las RDBMS clásicas.

A partir de entonces, el término "NoSQL" ha sido ampliamente discutido, *"...muchos de los avocados al NoSQL, especialmente en la blogósfera entendían el término y el movimiento como una total negación de RDBMS y proclamaban el fin de dichos sistemas"* [3]. Otros, como Michael Stonebraker reclaman que no debería tener relación con SQL y debería ser renombrado a un término como NoACID: *"..."NOSQL" en realidad significa "No disco" o "No ACID" o "No threading"..."* [4]. Finalmente, una tercer postura, critica el término "NoSQL", indicando que no se explicita lo que abarca el movimiento, sino lo que excluye: *"NoSQL es más fácilmente definido por lo que excluye: SQL, joins..."* [5].

En el paper "The end of an Architectural Area" [6], Michael Stonebraker et. al. sostienen que los pilares de los RDBMS fueron diseñados hace más de 25 años cuando las características del hardware, requerimientos del usuario y mercado, eran muy diferentes. Plantea que las raíces de dichas bases de datos provienen de System R de IBM, y que éste responde a una arquitectura de la época donde las características diferían notablemente de las actuales: almacenamiento orientado a disco y estructuras indexadas, multithreading para disminuir la latencia, mecanismos de control de concurrencia basados en bloqueos, y recuperación basada en logs, entre otros. En dicho trabajo se analizan y critican dichas características, contrastándolas contra productos de distinta índole y utilizando como marco de referencia el benchmark TPC-C. Concluye su estudio indicando varios aspectos que deberían ser considerados y alienta a un total rediseño de las bases de datos.

Bajo la premisa de un total rediseño de los RDBMS, las bases de datos NoSQL consideran fundamental trabajar sobre tres pilares de las aplicaciones web distribuidas: Consistencia, Alta Disponibilidad y Escalabilidad.

La Consistencia refiere a un contrato entre un almacenamiento de datos y procesos, en el cual el almacenamiento de datos especifica precisamente cuales son los resultados de las operaciones de lectura y escritura ante la presencia de concurrencia [7]. Se garantiza que a partir de un cambio en el estado de un sistema, todo cliente que accediera posteriormente, obtendrá el nuevo valor del estado: "Cada request recibe la respuesta correcta".

La Disponibilidad es la cualidad de un sistema para mantenerse operativo principalmente ante contingencias [8]: "Cada request eventualmente recibe una respuesta". La Alta Disponibilidad (HA) desde la visión de Eric Brewer [9] implica mantener el servicio funcionando ante:

- Caídas y Fallas de discos
- Actualizaciones de Bases de datos
- Actualizaciones de Software
- Actualizaciones de Sistema Operativo
- Cortes de energía
- Cortes en la red
- Movimiento físico del equipo

La Escalabilidad indica la habilidad de un sistema para reaccionar y adaptarse sin perder calidad, o bien manejar el crecimiento continuo de trabajo de manera fluida, o bien estar preparado para hacerse más grande sin perder calidad en los servicios ofrecidos [10].

Existe un cuarto requerimiento que se conoce como "Tolerancia a fallos o particiones": La Tolerancia a fallos refiere a que un sistema continúa funcionando a pesar de pérdidas en los mensajes por particiones [11].

Si bien es deseable contar con éstas características, no es posible hacerlo plenamente y en forma simultánea. Es necesario renunciar o al menos sacrificar parcialmente una para obtener las otras, tal como se menciona en el teorema de CAP.

1.2 Descripción del Problema

El teorema de CAP fue formulado por Eric Brewer y alega que en un sistema distribuido con datos compartidos se deben elegir, como máximo, dos de las tres

características: Consistencia, Alta Disponibilidad y Tolerancia a particiones. Brewer indica que se debe optar entre las propiedades ACID y las BASE (Básicamente disponible, estado flexible y eventualmente consistente), y propone como criterio de decisión lo siguiente: Si el sistema o parte del sistema debe ser consistente y tolerante a fallos (es el caso de los RDBMS), entonces se requiere optar por las propiedades ACID, mientras que si se pueden permitir inconsistencias con el fin de favorecer la disponibilidad y tolerancia a particiones (es el caso de Dynamo de Amazon [12]), entonces bien pueden aplicarse las propiedades BASE. Una tercera alternativa (utilizada por BigTable de Google [13]) consiste en no optar entre propiedades ACID y BASE, sino en mantener la consistencia y disponibilidad a costas de no ser completamente operativos en presencia de particiones de red. [9]

1.3 Propuesta de Solución

Existe una variedad de técnicas y patrones de diseño orientados a optimizar los requerimientos no funcionales anteriormente citados, incluso algunos de éstos aplican a más de uno. A los fines prácticos de éste estudio dividiremos las técnicas y patrones de diseño en:

- Técnicas y patrones de diseño orientadas a mantener la "Consistencia": Aplicaremos el enfoque de prioridad a los requerimientos de "Tolerancia a particiones" y "Alta Disponibilidad", favoreciendo la "Escalabilidad", para ello, se considerará la consistencia eventual. Algunas técnicas que pueden utilizarse para mantener el control de versiones sobre los elementos de datos son: el uso de TimeStamps, Bloqueo optimista, vector clock y Gossip, y almacenamiento multiversión, entre otros.

- Técnicas y patrones de diseño orientados a la maximización de la "Escalabilidad": Cuando el volumen de datos o capacidad de procesamiento son excedidos en una máquina, es necesario pensar en un esquema distribuido y para ello deben aplicarse técnicas de particionamiento. Dependiendo del tamaño del sistema y el dinamismo de la topología, pueden aplicarse distintas técnicas, entre ellas: Memory Caches, Clustering, separación de lecturas y escrituras, Sharding, Consistent Hashing y Membresía.

- Técnicas y patrones de diseño orientados a la "Alta Disponibilidad": Contar con este requerimiento implica no solo realizar replicación de datos, sino también disponer de mecanismos que permitan la detección y recuperación ante fallas eventuales y permanentes, así pueden citarse las siguientes técnicas: AntiEntropia a través de árboles Merkle, Sloppy Quorum, Hinted handoff, entre otros.

1.4 Objetivos

Este trabajo presentará un catálogo de técnicas y patrones de diseño aplicados actualmente en bases de datos NoSQL. Las técnicas presentadas favorecen la

“Escalabilidad” y la “Alta Disponibilidad” sobre la “Consistencia”, considerando que la “Consistencia Eventual” es aceptable bajo determinadas condiciones.

1.5 Estructura

El enfoque propuesto consistirá en una presentación del estado del arte de las bases de datos NoSQL, una exposición de los conceptos claves relacionados y una posterior exhibición del conjunto de técnicas y patrones de diseño orientados a la consistencia eventual, escalabilidad y alta disponibilidad.

Para tal fin,

- Se describirán brevemente las características principales de las bases de datos NoSQL, cuales son los factores que motivaron su aparición, sus diferencias con sus pares relacionales y una taxonomía de las mismas.
- Se presentará el teorema CAP y se contrastarán las propiedades ACID contra las BASE.
- Se introducirán las problemáticas que motivan las técnicas y patrones de diseño a describir.
- Se presentarán técnicas y patrones de diseños que solucionen las problemáticas.
- Finalmente, se concluirá con un análisis integrador, y se indicarán otros temas de investigación pertinente

1.6 Limitaciones

Se excluye del presente trabajo los siguientes puntos:

- Comparación entre bases de datos NoSQL
- Análisis exhaustivo de las características de los diferentes almacenamientos NoSQL: Se nombran algunos productos solo con el objetivo de mostrar referentes en la taxonomía.
- Demostración de teorema de CAP
- Detalles sobre implementación de las técnicas en los diferentes productos

2. Bases de datos NOSQL

2.1 Introducción

En el mercado actual de las aplicaciones web empresariales, los sistemas de gestión de bases de datos relacionales (RDBMS) son el medio de almacenamiento predominante. El modelo relacional ideado por Ted Codds en los años 70 aún sigue vigente, *"...ningún sistema ha tenido un completo rediseño desde su concepción."* [6]. Sin embargo, la idea de que un mismo modelo pueda adaptarse a todos los requerimientos (en inglés "one-size-fits-all") ha sido cuestionada, dando lugar a un abanico de bases de datos alternativas. Este movimiento es lo que se conoce como NoSQL (Notonly SQL).

Existe una larga discusión sobre si las ventajas y beneficios que ofrecen las bases de datos NoSQL son reales o solo pueden ser alcanzadas bajo situaciones ideales o de laboratorio [14], y si estas bases de datos son una propuesta seria o solo una moda influenciada por grandes empresas de internet bien conocidas como Facebook, LinkedIn, Amazon.com, entre otros.

Una fuerte crítica puede encontrarse en el paper de Oracle "Debunking the nosql hype": *"...con tantas opciones introducidas regularmente, las bases de datos NoSQL están empezando a parecerse a una heladería que te atrae con el nuevo sabor del mes. Sin embargo, no deberías atarte demasiado a nuevos sabores, porque podrían no estar disponible por mucho tiempo..."* [14]. Es para tener en cuenta que 4 meses después de la presentación de dicho paper, en la conferencia Oracle OpenWorld de San Francisco, Oracle a través de Thomas Kurian, vicepresidente ejecutivo de desarrollo de productos, presentó su alternativa NoSQL, que estaría basada en la base de datos open-source BerkeleyDB y soportada por un nuevo hardware conocido como Oracle Big Data Applicance [15].

Desde la visión de los adeptos a los RDBMS podemos mencionar las siguientes críticas a las bases de datos NoSQL:

- **No hay un claro líder:** El mercado de NoSQL está muy fragmentado, lo cual es un problema para el open-source porque se requiere una gran cantidad de desarrolladores para tener éxito [14].

- **Cada una de las bases de datos NoSQL posee su propia interfaz no standard:** La adaptación a una base de datos NoSQL requiere una inversión significativa para poder ser utilizada. Debido a la especialización, una compañía podría tener que instalar más de una de estas bases de datos [16]. Por ello, algunos describen el mercado NoSQL como monopolísticamente competitivo (bajas barreras para entrar y salir, muchos pequeños proveedores con productos técnicamente heterogéneos y diferenciados, y un mercado inconsistente con las condiciones para la perfecta competencia), donde las empresas NoSQL están condenadas a obtener cero ganancias económicas a largo plazo [17].

- **El concepto "one-size-fits-all" ha sido criticado por FOSS y pequeñas startups porque ellos no pueden hacerlo:** El mantra NoSQL de "utilizar la herramienta correcta" resulta en muchas herramientas frágiles de propósito especializado, ninguna de las cuales es suficientemente completa para proveer una total funcionalidad [17].

- **Escalabilidad no tan simple:** Una de las características más difundida de las bases de datos NoSQL es su capacidad de escalar horizontalmente. Esta es promovida como una manera de manejar el crecimiento impredecible o exponencial de las necesidades de negocio, pero con frecuencia es más fácil decirlo que hacerlo, tal como lo demostraron los problemas de sharding que generaron el apagado en el Foursquare [18].

- **Se requiere una reestructuración de los modelos de desarrollo de aplicaciones:** Utilizar una base datos NoSQL típicamente implica usar un modelo de desarrollo de aplicaciones diferente a la tradicional arquitectura de 3 capas. Por lo tanto, una aplicación existente de 3 capas no puede ser simplemente convertida para bases de datos NoSQL, debe ser reescrita, sin mencionar que no es fácil reestructurar los sistemas para que no ejecuten consultas con join o no poder confiar en el modelo de consistencia read-after-write [14].

- **Con frecuencia se simplifica el teorema de CAP a *"Elige 2 de 3: Consistencia, disponibilidad, tolerancia a particiones":*** El uso del algoritmo de CAP para justificar la consistencia parcial en vistas de favorecer la tolerancia a particiones, ha generado un debate en la comunidad científica. Se alega que el teorema de CAP alimenta la filosofía de desarrollo de aplicaciones antes mencionadas, alentando la inconsistencia y "disculpas" [19].

Po otro lado, la elección sobre cuando utilizar consistencia parcial y cuando utilizar consistencia total no siempre esta tan claro, y en ciertos casos de consistencia parcial no se observan beneficios reales [20]. Pareciera que el uso de una arquitectura con consistencia parcial es realmente para sistemas que no pueden tolerar segundos de inactividad y están dispuestos a afrontar el problema de construir tolerancia a las inconsistencias en la aplicación con tal de evitar ésta inactividad [21].

- **Modelos de datos sin esquema podría ser una mala decisión de diseño:** Si bien los modelos de datos sin esquema son flexibles desde el punto de vista del diseñador, son difíciles para consultar sus datos. El modelo Entidad-Atributo-Valor (EAV) funciona bien para consultas clave-valor, pero para consultas de rango o más complejas (por ejemplo para reportes), éste esquema es complicado y lento [22]. EAV solo tiene sentido para esquemas que cambian frecuentemente.

  En los modelos de datos sin esquema, el manejo de los datos es delegado a la capa de aplicación. Dado que la aplicación necesita conocer que información almacena y como, a medida que evolucionan los datos la aplicación debe ser capaz de manejar todos los diferentes formatos [14].

- **Tú no eres Google:** Las bases de datos NoSQL han sido promocionadas por grandes compañías de internet, lo cual le ha agregado credibilidad. Estas empresas estaban motivadas por la posibilidad de contar con un software de base de datos que este bajo su control total, para ello contrataron a los mejores desarrolladores para la implementación y soporte de sus bases de datos. ¿Se encuentra tu compañía preparada para realizar dicha inversión en un área que no es tu competencia? ¿Está dispuesto a apostar su proyecto en vías de contribuir al open-source? Recuerda que tú no eres Google [14].

Por otro lado, desde la visión de los adeptos a las bases de datos NoSQL podemos mencionar las siguientes razones para desarrollar y utilizar éstos almacenamientos:

- **Evitar la complejidad innecesaria:** Los RDBMS proveen un conjunto amplio de características y obligan el cumplimiento de las propiedades ACID, sin embargo, para algunas aplicaciones éste set podría ser excesivo y el cumplimiento estricto de las propiedades ACID innecesario [3]. *"Las bases de datos relacionales te obligan a torcer tus datos para encajar en un RDBMS..."* [23].

- **Alto rendimiento:** En la década de los 80 las consultas a las bases de datos podían correr de noche como procesos batch, hoy día esto no es aceptable. Si bien algunas funciones analíticas pueden ejecutarse de noche, la evolución de la web requiere respuestas inmediatas a las consultas [24]. Las bases de datos NoSQL proveen un rendimiento mayor a las relacionales, incluso de hasta varios órdenes de magnitud: De acuerdo a una presentación realizada por los ingenieros Avinash Lakshman y Prashant Malik de Facebook, Cassandra puede escribir en un almacenamiento de datos más de 50 GB en solo 0.12 milisegundos, mientras que MySQL tardaría 300 milisegundos para la misma tarea [25].

- **Incremento del volumen de información no estructurada y empleo de hardware más económico:** En contraste con los RDBMS, la mayoría de las bases de datos NoSQL son diseñadas para poder escalar horizontalmente y no tener que confiar en hardware altamente disponible. Las máquinas pueden ser agregadas o quitadas sin el esfuerzo operacional que implica realizar sharding en soluciones de cluster de RDBMS: *"Oracle te diría que con el grado correcto de hardware, la configuración correcta de Oracle RAC (Real Application Clusters) y algún otro software mágico asociado, puedes alcanzar la misma escalabilidad. Pero, ¿a qué costo?"* [26].

- **Evitar el costoso mapeo objeto-relacional:** La mayoría de las bases de datos NoSQL son diseñadas para almacenar estructuras de datos más simples o más similares a las utilizadas en los lenguajes de programación orientados a objetos.

  De esta forma, se evita costosos mapeos objeto-relacional, beneficiando principalmente a aplicaciones de baja complejidad que difícilmente se benefician de un RDBMS [3].

- **El pensamiento "One-size-fits-all" estaba y sigue estando equivocado:** Existe un número creciente de escenarios que no pueden ser abarcados con un enfoque de base de datos tradicional. Si bien esta idea no es nueva, la búsqueda de alternativas a los tradicionales RDBMS ha sido impulsada por dos tendencias:

  – El continuo crecimiento de volúmenes de datos a ser almacenados

  – El crecimiento de la necesidad de procesar grandes cantidades de datos en cortos tiempos

"La verdad es que no necesitas ACID para las actualizaciones de estado de Facebook o tweets o comentarios, mientras que las capas de negocio y presentación puedan manejar robustamente datos inconsistentes. Obviamente no es lo ideal, pero aceptar pérdidas de datos o inconsistencias como una posibilidad, pueden llevar a una dramática flexibilidad…" [27].

- **El mito de particionar y distribuir un modelo de datos centralizado sin esfuerzo:** Los modelos de datos diseñados con una sola base de datos en mente, generalmente no pueden ser particionados y distribuidos fácilmente entre servidores de bases de datos: Cuando una base de datos empieza a crecer, al principio, se puede configurar replicación. A medida que la cantidad de datos sigue creciendo, se debe aplicar sharding con costos sistemas de administración o invertir una cantidad significativa de dinero en proveedores de DBMS para que operen la base de datos [3].

  Por otro lado, si bien las abstracciones intentan ocultar a la aplicación aspectos relacionados a la distribución y particionamiento (a través de capas proxy que redirigen los requests a los correspondientes servidores), ésta abstracción no puede aislar a la aplicación de la realidad: La aplicación necesita conocer aspectos relacionados a latencia, fallas distribuidas, etc. de forma tal de contar con suficiente información para tomar la decisión correcta específica al contexto. Por ello, se sugiere diseñar el modelo de datos para que se adapte a un ambiente particionado incluso si inicialmente solo habrá un servidor de base de datos centralizado. Este enfoque ofrece la ventaja de evitar costoso cambios posteriores en la aplicación. [28]

- **RDBMS con MemCache vs. Sistemas construidos desde el inicio con escalabilidad en mente:** *"Sharding con MySQL para soportar las cargas de escritura, cache de objetos con memcached para manejar las cargas de lecturas y mucho código de integración es como solía hacerse, pero a medida que crecen los requerimientos de escalabilidad, éstas tecnologías se vuelven obsoletas"* [3].

- **Las necesidades de ayer vs. las de hoy:** Las necesidades relacionadas al almacenamiento de datos han cambiado considerablemente con el tiempo:

  En las décadas del 60 y 70 las bases de datos se diseñaban para ejecutarse en grandes y costosas máquinas aisladas. En cambio, hoy día, muchas grandes

compañías optan por utilizar hardware más económico con una probabilidad de fallo predecible, y en consecuencia, diseñan las aplicaciones para que manejen tales fallos que se consideran parte del "modo normal de operación" [3].

Los RDBMS son adecuados para datos relacionados rígidamente estructurados, permitiendo consultas dinámicas expresadas en un lenguaje sofisticado. Sin embargo, hoy día, particularmente en el sector web, los datos no son rígidamente estructurados ni se requieren consultas dinámicas, ya que la mayoría de las aplicaciones conocen de antemano la información que se puede requerir y por lo tanto es suficiente contar con consultas predefinidas en la base de datos y asignar valores a las variables dinámicamente [3].

Desde el punto de vista del diseño, las bases de datos relacionales fueron concebidas para instalaciones centralizadas y no distribuidas. Aunque se han introducido mejoras para alcanzar esta funcionalidad, las bases de datos relacionales acarrean falencias por no ser diseñadas con los conceptos de distribución en mente: por ejemplo, el mecanismo de sincronización con frecuencia no se implementa eficientemente y requiere costosos protocolos como commit en 2 o 3 fases. Otra dificultad se presenta al intentar hacer "transparente" a la aplicación la arquitectura de clusters de una base de datos relacionales, esto conlleva a las famosas falencias de la computación distribuida:

"Esencialmente cada persona cuando construye su primer aplicación distribuida realiza los siguientes ocho supuestos, que terminan siendo falsos y generan grandes problemas a largo plazo:

- *La red es confiable.*
- *La latencia es cero.*
- *El ancho de banda es infinito.*
- *La red es segura.*
- *La topología no cambia.*
- *Existe un solo administrador.*
- *El costo de transporte es cero.*
- *La red es homogénea."* [29]

2.2 Taxonomía

Debido a la heterogeneidad de las distintas soluciones NoSQL, existen varios criterios de clasificación. Una interesante y simplificada taxonomía sugiere utilizar la "evolución" de estos almacenamientos como referencia en la clasificación [30], bajo este criterio tenemos:

- **Almacenamiento clave-valor:** es un modelo simplista pero muy poderoso. Posee una pobre aplicabilidad para los casos que requieren procesamiento de rangos de claves.

- **Almacenamiento clave-valor ordenado:** Este modelo suple la limitación de procesamiento de un rango de claves y mejora significativamente las capacidades de agregación, sin embargo no provee ningún framework para el modelado de valores.

  Los almacenamientos clave-valor consisten en un mapa o diccionario (DHT) en el cual se puede almacenar y obtener valores a través de una clave. Este modelo favorece la escalabilidad sobre la consistencia, y omite/limita las funcionalidades analíticas y de consultas complejas ad-hoc (especialmente joins y operaciones de agregación). Si bien los almacenamientos por clave-valor han existido por largo tiempo, un gran número de éstos ha emergido influenciados por Dynamo de Amazon [3]. Otros representantes de este grupo son Proyecto Voldemort, Tokyo Cabinet/Tokyo Tyrant, Redis, Memcache y Scalaris.

  En la siguiente ilustración puede observarse un almacenamiento clave-valor:

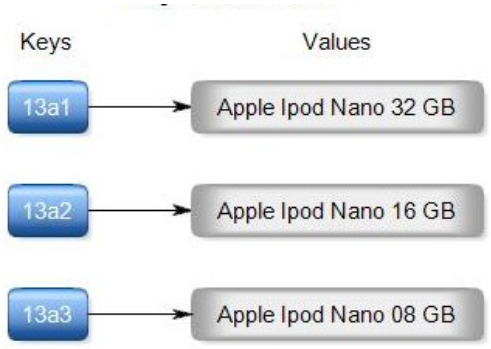

Ilustración 1: Almacenamiento Clave-Valor

Ilustración de http://nosql.rishabhagrawal.com/2012/07/types-of-nosql-databases.html

- **Bases de datos orientadas a columna:** Este modelo es representado por BigTable quien modela los valores como una terna de mapa de mapas de mapas (familias de columnas, columnas y versiones con timestamps).

El enfoque de almacenar y procesar datos por columna en lugar de fila tiene sus orígenes en Analytics y BI, donde se han construido aplicaciones de gran performance bajo una arquitectura de procesamiento paralelo shared-nothing. Algunos ejemplos en este campo son Sybase IQ y Vertica [31].

BigTable es descripto como un "mapa ordenado, multidimensional, persistente, distribuido y disperso" [13], aunque no tan purista, suele utilizarse como referente para las bases de datos orientadas a columna. Otros autores lo catalogan como un "almacenamiento de columna amplia", un "almacenamiento de registro extensible" o un "almacenamiento EAV" [3]. Algunos ejemplos basados en BigTable son HyperTable, HBase, Riak y Cassandra.

En la siguiente ilustración puede observarse una base de datos pura orientada a columna y su diferencia con una orientada a fila:

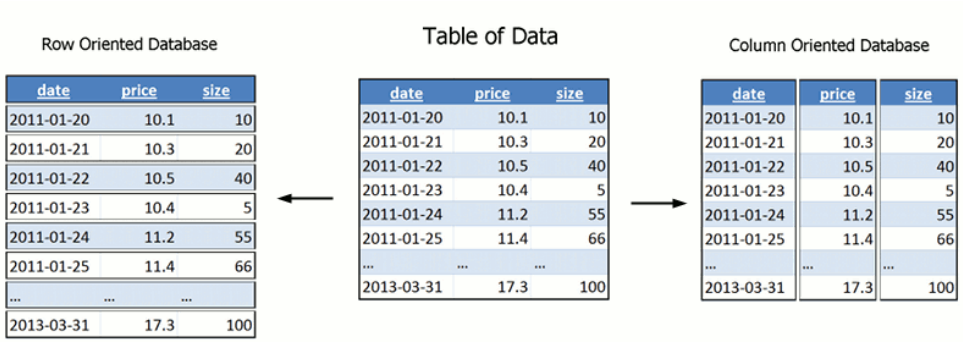

Ilustración 2: Base de datos orientada a columna vs. orientada a fila

Ilustración de http://www.timestored.com/kdb-guides/kdb-database-intro

En la siguiente ilustración puede observarse el modelo de columnas propuesto por BigTable y utilizado por productos como Cassandra:

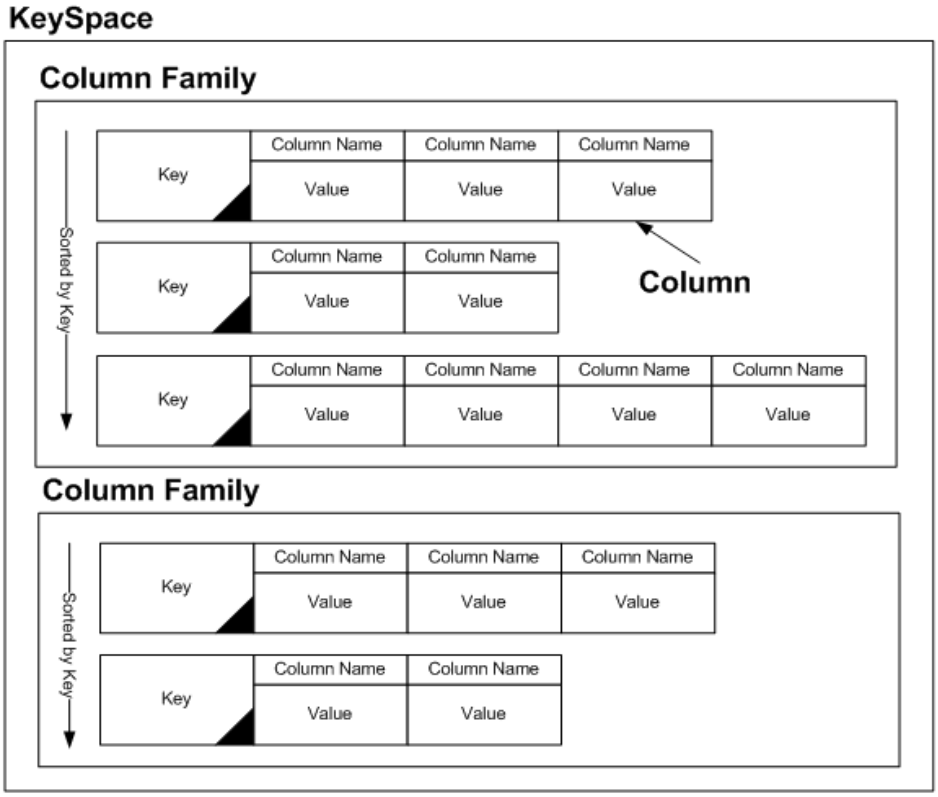

Ilustración 3: Modelo de datos de Cassandra

Ilustración de http://javamaster.wordpress.com/2010/03/22/apache-cassandra-quick-tour/

- **Base de datos documental:** Son consideradas descendientes de Lotus Notes ya que comparten muchos conceptos: documentos, vistas, distribución y replicación entre servidores y clientes. De hecho, los avocados a éstas bases de datos las llaman "Notes hecho correctamente" [3].
  Ofrecen dos mejoras significativas respecto al modelo de columnas, la primera es que permiten estructuras de datos más complejas, encapsulando los pares clave-valor en documentos; la segunda mejora es el sistema de indexado a través de árboles B.

  Dos representantes de este género son Apache CouchDB (utilizado en el proyecto Ubuntu One) y MongoDB (incluido en SourceForge.net, foursquare, The New York Times, bit.ly, etc.).

  En la siguiente ilustración puede observarse una base de datos documental y su diferencia con una relacional:

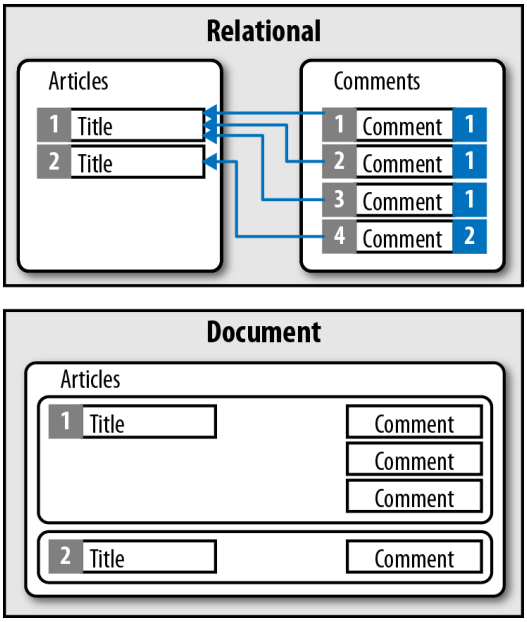

Ilustración 4: Base de datos documental vs. relacional

Ilustración de http://www.prabathsql.blogspot.com.ar/

- **Base de datos orientada a grafos:** Inspiradas en Euler y la teoría de grafos, permite contar con un modelo de negocio más complejo, con flexibilidad en las relaciones entre entidades [30].

  Estas bases de datos utilizan una estructura de grafo con nodos, aristas y propiedades para representar y almacenar datos. Por definición, una base de datos orientada a grafos es cualquier sistema de almacenamiento que provea libre indexado por adyacencia. Esto significa que cada elemento contiene un puntero directo a su elemento adyacente y no requiere búsqueda por índices. Se distinguen las bases de datos generales orientadas a grafos que pueden almacenar cualquier tipo de grafo, de las especializadas tales como bases de datos de red y triple-store [32].
  Algunos casos destacables de bases de datos orientadas a grafos son Neo4j, HyperGraphDB, AllegroGraph y VertexDB.

  En la siguiente ilustración puede observarse una base de datos orientada a grafo:

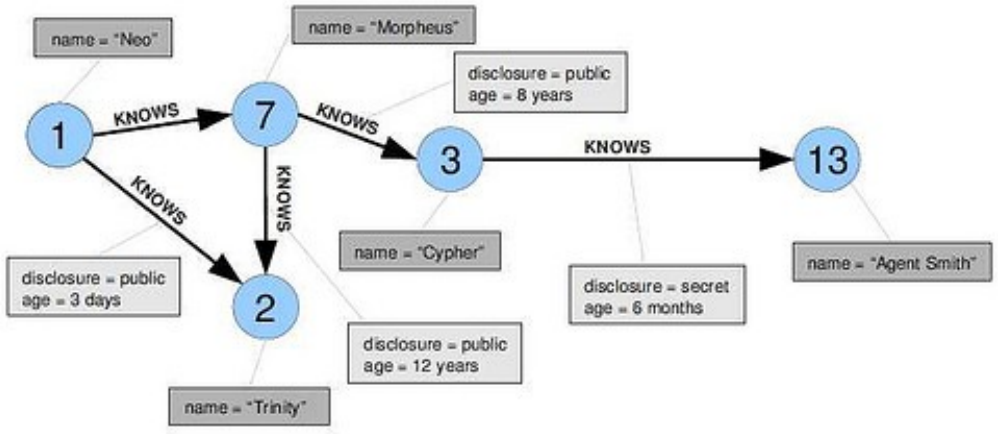

Ilustración 5: Base de datos orientada a grafos

Ilustración de http://highscalability.com/neo4j-graph-database-kicks-buttox

En la siguiente figura puede verse la citada "evolución" de los almacenamientos NoSQL:

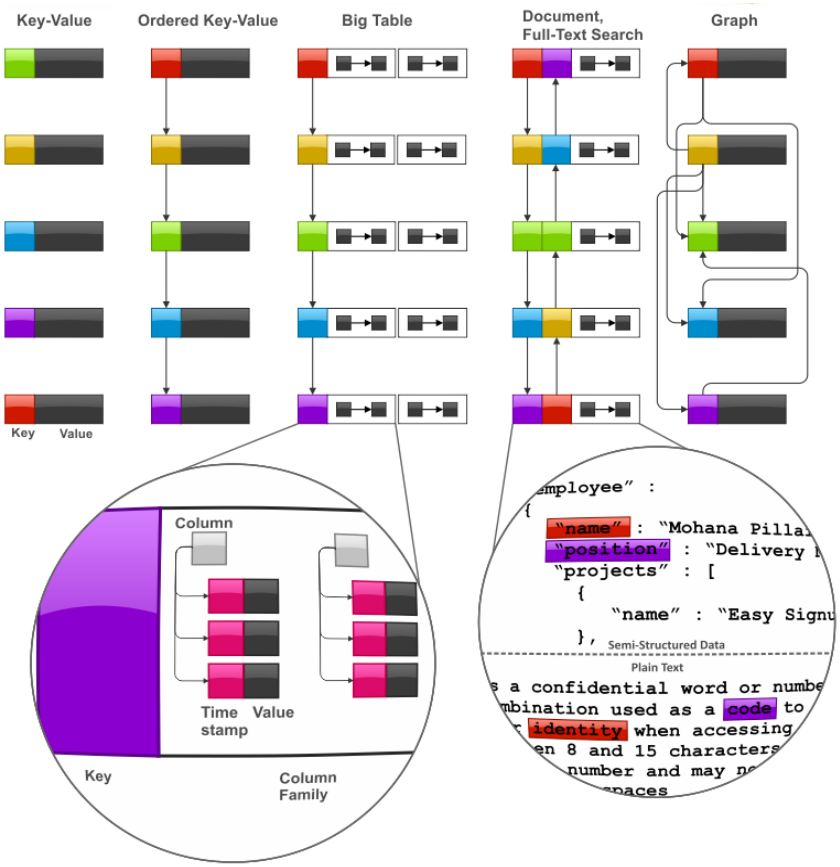

Ilustración 6: Evolución de las bases de datos NoSQL

Ilustración de http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/

Una amplia y actualizada clasificación es mantenida por Stefan Edlich en su website [33], aquí incluye más de 150 almacenamientos NoSQL, entre ellos:

- **Almacenamientos de columna amplia/familia de columnas:** Cassandra, Hadoop, HyperTable, etc.
- **Almacenamientos de documentos:** MongoDB, CouchDB, Terrastore, etc.
- **Almacenamientos clave-valor/tupla:** DynamoDB, Riak, MemcacheDB, etc.
- **Bases de datos orientadas a grafos:** Neo4J, VertexDB, AllegroGraph, etc.
- **Bases de datos multimodelos:** ArangoDB, OrientDB, FatDB, etc.
- **Bases de datos orientadas a objetos:** Versant, Objectivity, VelocityDB, etc.
- **Soluciones de bases de datos en la nube y data grid:** GigaSpaces, GemFire, Infinispan, etc.
- **Bases de datos XML:** EMC Documentum xDB, eXist, Berkeley DB XML, etc.
- **Bases de datos multidimensional:** Globals, Intersystems Cache, GT.M, etc.
- **Bases de datos multivalor:** U2, OpenInsight, ESENT, etc.
- **Event sourcing:** Event Store.
- **Otras bases de datos relacionadas a NoSQL:** IBM Lotus/Domino, ISIS Family, VaultDB, etc.
- **Sin categorizar:** KirbyBase, FileDB, Tokutek, etc.

3. Propiedades ACID vs BASE

3.1 Teorema de CAP

Durante el simposio de "Principios de computación distribuida" de ACM en el año 2000, Eric Brewer, un profesor de la universidad Berkeley de California y cofundador de Inktomi, a través de una presentación titulada "Hacia sistemas distribuidos robustos", estableció la conjetura que los servicios web no pueden asegurar en forma conjunta las siguientes propiedades: Consistencia (C), Disponibilidad (A) y Tolerancia a particiones (P), esto es lo que se conoce como el teorema de CAP [9].

Posteriormente en el año 2002, Seth Gilbert y Nancy Lynch de MIT publicaron una demostración formal de la conjetura de Brewer, convirtiéndola en un teorema [34]. Si bien esta demostración ha sido criticada [35], el teorema de CAP ha sido ampliamente adoptado por grandes compañías web como Amazon [21], así como por la comunidad NoSQL.

El teorema de CAP establece que en un sistema distribuido con datos compartidos, se debe optar por favorecer dos de las tres características: Consistencia, Disponibilidad y Tolerancia a particiones. Bajo estas restricciones, Brewer indica que se debe utilizar como criterio de selección, los requerimientos que se consideren más críticos para el negocio, optando entre propiedades ACID y BASE [9].

Con frecuencia el teorema de CAP ha sido malinterpretado y aplicado, en particular, muchos diseñadores concluyen incorrectamente que el teorema impone restricciones en los sistemas de bases de datos durante su normal funcionamiento y por lo tanto implementan los sistemas innecesariamente limitados, cuando en realidad, el teorema de CAP solo establece limitaciones de cara a ciertos tipos de fallas y no limita las capacidades de un sistema durante su normal operación: *"Es incorrecto asumir que los sistemas de bases de datos que reducen la consistencia en ausencia de particiones, lo estén haciendo debido a una decisión basada en el teorema de CAP"* [36].

Por otro lado, algunos autores consideran que el teorema de CAP es impreciso y no abarca de forma adecuada aspectos como la latencia. El profesor Daniel Avadi propone reemplazar CAP por PACELC: Si existe una partición (P), ¿cómo elige el sistema favorecer entre disponibilidad y consistencia (A y C); y sino (E) cuando el sistema se encuentra funcionando normalmente bajo ausencia de particiones, como elige el sistema favorecer entre latencia (L) y consistencia (C)? [36]

La siguiente tabla basada en la presentación de Brewer, establece cuales son las alternativas, algunas características y ejemplos:

Alternativa	Características	Ejemplos
CA: Consistencia + Disponibilidad (sacrificando Tolerancia a particiones)	❖ Protocolos de commit en 2 fases ❖ Protocolos de validación de caché	❖ Bases de datos centralizadas ❖ Cluster de bases de datos ❖ LDAP ❖ Sistema de archivos xFS
CP: Consistencia + Tolerancia a particiones (sacrificando Disponibilidad)	❖ Mecanismos de bloqueo pesimista ❖ Protocolos de Quorum mayoritario: Paxos	❖ Bases de datos distribuídas ❖ Bloqueo distribuído
AP: Disponibilidad + Tolerancia a particiones (sacrificando Consistencia)	❖ Protocolos de resolución de conflictos: Gossip ❖ Manejo de expiraciones y leasing ❖ Enfoque optimista	❖ Sistema distribuido Coda ❖ Web cache ❖ DNS

Tabla 1: Alternativas en Teorema de CAP

Algunos de los protocolos, mecanismos y enfoques asociados a cada alternativa son desarrollados en secciones posteriores de este trabajo.

3.2 Propiedades ACID en sistemas distribuidos

Las propiedades ACID presentes en las transacciones de las bases de datos relacionales, han simplificado enormemente el trabajo de los desarrolladores de aplicaciones, ya que ofrecen garantías en cuanto a:

- Atomicidad: Todas las operaciones en la transacción serán completadas o ninguna lo será.
- Consistencia: La base de datos estará en un estado valido tanto al inicio como al fin de la transacción.
- Aislamiento: La transacción se comportará como si fuera la única operación llevada a cabo sobre la base de datos (una operación no puede afectar a otras).
- Durabilidad: Una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

Los proveedores de bases de datos hace tiempo reconocieron la necesidad de particionamiento e introdujeron el protocolo de commit en 2 fases para proveer las garantías de las propiedades ACID en múltiples instancias de bases de datos [37]. El protocolo se divide en dos partes o fases:

- Primera fase: el coordinador de la transacción solicita a cada base de datos involucrada que realice un precommit de la operación e indique si es posible realizar el commit. Si todas las bases de datos confirman que pueden proceder con el commit, entonces se inicializa la fase 2.
- Segunda fase: el coordinador de la transacción solicita a cada base de datos que realice commit de los datos.

Si alguna base de datos no logra realizar el commit, entonces se solicita a todas las bases de datos que realicen roll back de esa transacción. En vista de éste protocolo, ¿cuál es el inconveniente? Ya que estamos obteniendo consistencia entre las particiones. Si Brewer estuviera en lo correcto, entonces estaríamos afectando la disponibilidad:

La disponibilidad de un sistema es el producto de la disponibilidad de los componentes requeridos para la operación. Entonces, una transacción que involucre dos bases de datos en un protocolo de commit de 2 fases tendrá como disponibilidad la disponibilidad de cada base de datos. Por ejemplo, si asumimos que cada base de datos posee 99.9 % de disponibilidad, entonces, la disponibilidad de la transacción resulta de 99.8 %, o un tiempo de inactividad de 43 minutos por mes [37].

3.3 Propiedades BASE

Así como las propiedades ACID proveen consistencia, las propiedades BASE proveen disponibilidad.

BASE refiere a básicamente disponible (BA), estado flexible (S) y eventualmente consistente (E): *"Una aplicación trabaja básicamente todo el tiempo (BA), no requiere ser consistente todo el tiempo (S), pero eventualmente estará en un estado conocido (E)"* [38].

Las propiedades BASE son opuestas a las ACID: mientras que ACID es pesimista y fuerza la consistencia al finalizar cada operación, BASE es optimista y acepta que la consistencia de la base de datos esté en un estado flexible. Aunque este enfoque pueda parecer imposible de lidiar, en realidad es bastante manejable y permite niveles de escalabilidad que no pueden ser alcanzados con ACID [37].

La disponibilidad en las propiedades BASE es alcanzada a través de mecanismos de soporte de fallas parciales, que permite mantenerse operativos y evitar una falla total del sistema. Así, por ejemplo, si la información de usuarios estuviera particionada a través de 5 servidores de bases de datos, un diseño utilizando BASE alentaría una estrategia tal que una falla en uno de los servidores impacte solo en el 20% de los usuarios de ese host [37].

Desde la visión de algunos autores, podemos encontrar diversos "niveles" de consistencia, siendo dos extremos la consistencia estricta y la consistencia eventual [39]:

Consistencia estricta: *"Todas las operaciones deben retornar los datos de la última operación de escritura completa, sin importar a que replica se dirigió la operación"*. Esto implica que tanto las operaciones de lectura como las de escritura para un set de datos dado, deben ser ejecutadas en un mismo nodo o debe ser asegurada la consistencia estricta a través de un protocolo de transacciones distribuidas (como el protocolo de commit de 2 fases o Paxos). Tal nivel de consistencia no puede ser alcanzado junto a disponibilidad y tolerancia a particiones de acuerdo al teorema de CAP.

Consistencia eventual: Los lectores verán las escrituras con el paso del tiempo: *"En un estado transitorio, el sistema eventualmente retornará los valores de la última escritura"*. Los clientes por lo tanto deben lidiar con un estado inconsistente de los datos mientras las actualizaciones estén en progreso. Así, por ejemplo, en una base de datos replicada, las actualizaciones podrían ir a un nodo quien replicará la última versión al resto de los nodos que eventualmente tendrán la última versión del set de datos.

Otros niveles de consistencia identificados son los siguientes [3]:

Consistencia "Lee tus propias escrituras" (RYOW): Significa que un cliente ve sus actualizaciones inmediatamente después que han sido realizadas y completadas, sin importar si la operación de escritura impacto en un servidor y la de lectura en otro diferente. Las actualizaciones de otros clientes no son visibles inmediatamente para dicho cliente.

Consistencia de sesión: Significa que un cliente ve sus actualizaciones inmediatamente después que han sido realizadas y completadas solo si las operaciones de lectura posteriores a la actualización son ejecutadas en la misma sesión.

Consistencia casual: Implica que si un cliente lee la versión X de un set de datos y en consecuencia escribe la versión Y, cualquier cliente leyendo la versión Y también podrá ver la versión X.

Consistencia de lectura monótona: Si un proceso ha visto un valor particular de un objeto, cual acceso subsecuente nunca retorna un valor previo.

3.3.1 Identificación de consistencia flexible

Según las conjeturas de Brewer, se debe identificar oportunidades donde se pueda flexibilizar la consistencia. Esto con frecuencia es difícil, debido a la tendencia de ambos grupos (stakeholders y desarrolladores) a afirmar que la consistencia es de suma importancia para el éxito de la aplicación. La inconsistencia temporal no puede ser ocultada al usuario final, por ello, tanto los ingenieros como los dueños del producto deben estar involucrados en la selección de las oportunidades de flexibilidad de consistencia [37].

A continuación se presenta un ejemplo simplificado que ilustra algunos aspectos que pueden ser considerados al evaluar la consistencia en las propiedades BASE:

Supongamos el siguiente esquema donde se modela una relación entre usuarios y sus transacciones de compras y ventas:

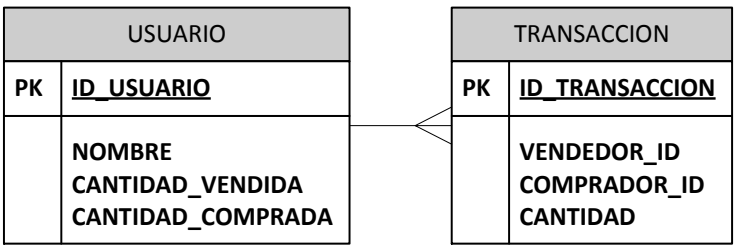

Ilustración 7: Ejemplo esquema Usuario-Transacción

La tabla USUARIO contiene información de usuarios incluyendo la cantidad total vendida y comprada.

La tabla TRANSACCION contiene cada transacción efectuada, relacionando un vendedor, un comprador y la cantidad acordada.

En general, la consistencia entre grupos funcionales es más fácil de flexibilizar que la consistencia dentro de los grupos. En el esquema del ejemplo hay dos grupos funcionales: usuarios y transacciones. Cada vez que se vende un ítem, una fila es agregada a la tabla TRANSACCION y los contadores para el comprador y vendedor son actualizados.

Utilizando una transacción ACID, las sentencias SQL serían como se muestra a continuación:

```
Begin transaction
  Insert into transaccion(id_transaccion,vendedor_id,comprador_id,cantidad);
  Update usuario set cantidad_vendida=cantidad_vendida+$cantidad
      where id_usuario=$vendedor_id;
  Update usuario set cantidad_comprada=cantidad_comprada+$cantidad
      where id_usuario=$comprador_id;
End transaction
```

Ilustración 8: Ejemplo transacción ACID

Las columnas cantidad comprada y cantidad vendida en la tabla USUARIO pueden ser consideradas un cache de la tabla TRANSACCION, solo están presentes para mejor la eficiencia del sistema. Considerado esto, la restricción de consistencia podría ser flexibilizada. Para ello, las expectativas del comprador y vendedor deberían ser establecidas para que sus balances actuales no reflejen el resultado de una transacción inmediata. De hecho, esta demora es muy común hoy día (por ejemplo al realizar extracciones en cajeros automáticos).

La manera en que se modificaran las sentencias SQL para flexibilizar la consistencia, dependerá de cómo estén definidos los balances. Si son solamente "estimados", permitiendo omitir las últimas transacciones, las sentencias SQL pueden modificarse como se muestra en la siguiente figura:

```
Begin transaction
  Insert into transaccion(id_transaccion,vendedor_id,comprador_id,cantidad);
End transaction
Begin transaction
  Update usuario set cantidad_vendida=cantidad_vendida+$cantidad
      where id_usuario=$vendedor_id;
  Update usuario set cantidad_comprada=cantidad_comprada+$cantidad
      where id_usuario=$comprador_id;
End transaction
```

Ilustración 9: Ejemplo transacción desacoplada

A partir de este cambio hemos desacoplado las actualizaciones de las tablas USUARIO y TRANSACCION y por lo tanto, la consistencia entre las tablas ya no puede garantizarse. De hecho, una falla entre la primera y segunda transacción resultará en que la tabla USUARIO sea inconsistente, pero si el contrato estipula que los totales son "estimados", este enfoque podría ser adecuado.

¿Qué sucedería si los "estimados" ya no son aceptables? ¿Cómo pueden desacoplarse las actualizaciones en las tablas USUARIO y TRANSACCION? Una alternativa es utilizar colas para mensajes persistentes. Existen varias opciones para implementar mensajes persistentes, pero el factor crítico es asegurar que la persistencia sea sobre el mismo recurso de la base de datos. Esto es necesario para permitir que la cola sea transaccional en los commit, sin involucrar el protocolo de commit de 2 fases.

Así, considerando el enfoque de mensajes persistentes tendremos la siguiente transacción:

```
Begin transaction
  Insert into transaccion(id_transaccion,vendedor_id,comprador_id,cantidad);
  Queue message "update USUARIO("vendedor",vendedor_id,cantidad);
  Queue message "update USUARIO("comprador",comprador_id,cantidad);
End transaction
For each message in queue
  Begin transaction
    Dequeue message
    If message.balance == "vendedor"
      Update USUARIO set cantidad_vendida=cantidad_vendida + message.cantidad
        where id_usuario = message.id_usuario;
    Else
      Update usuario set cantidad_comprada=cantidad_comprada + message.cantidad
        where id_usuario = message.id_usuario;
    End if
  End transaction
End For
```

Ilustración 10: Ejemplo transacción con cola de mensajes persistentes

Encolando un mensaje persistente dentro de la misma transacción que se realiza la inserción, se conserva la lógica de actualización de balances. La transacción es contenida en una sola instancia de base de datos y por lo tanto no afectará la disponibilidad del sistema. Un componente diferente procesará los mensajes, para ello desencolará cada mensaje y aplicará las actualizaciones a la tabla USUARIO. Este esquema parece cubrir todos los aspectos, pero aún existe un problema. El mensaje persistente se encuentra en el host de la transacción para evitar el protocolo de 2 fases durante el encolamiento. Si el mensaje se desencola dentro de la transacción que involucra al host del usuario, todavía tenemos una situación de protocolo de 2 fases.

Una solución es no hacer nada. Al desacoplar para que la actualización se realice desde un servidor independiente, se conserva la disponibilidad del componente de cara al cliente y tal vez este nivel de disponibilidad es aceptable a los requerimientos del negocio.

Si suponemos que el protocolo de 2 fases no es aceptable como solución bajo ningún punto de vista, podemos optar por otro enfoque, para ello primero definiremos el concepto de operación idempotente:

"Una operación es considerada idempotente si puede ser aplicada una o múltiples veces con el mismo resultado. Las operaciones idempotentes son útiles ya que permiten fallas parciales, pues la aplicación repetida de ellas no modifica el estado final del sistema"

Para el ejemplo considerado la aplicación de idempotencia no es sencilla, debido a que las operaciones de actualización son raramente idempotentes. En el ejemplo, la actualización incrementa las columnas de balances y por lo tanto aplicar la operación más de una vez, obviamente resultaría en un balance incorrecto. Incluso en operaciones de actualización donde simplemente se asigna un valor, la idempotencia podría ser difícil de aplicar debido al orden de las operaciones. Si el sistema no puede garantizar que las actualizaciones sean aplicadas en el orden en que fueron recibidas, el estado final del sistema será incorrecto.

Siguiendo con el ejemplo, se requiere un mecanismo para rastrear que actualizaciones han sido aplicadas exitosamente y cuales se encuentran pendientes. Una técnica es utilizar una tabla que identifique las transacciones que han sido aplicadas.

La siguiente tabla muestra un ejemplo de una tabla de seguimiento de transacciones, detalla que balances han sido actualizados y el identificador del usuario donde se aplicó la actualización de balance:

ACTUALIZACIONES_APLICADAS	
PK	**ID_ACTUALIZACION**
	ID_TRANSACCION **BALANCE** **ID_USUARIO**

Ilustración 11: Tabla actualizaciones aplicadas

En base a esta tabla, la transacción quedaría como se indica a continuación:

```
Begin transaction
  Insert into transaccion(id_transaccion,vendedor_id,comprador_id,cantidad);
  Queue message “update USUARIO(“vendedor”,vendedor_id,cantidad);
  Queue message “update USUARIO(“comprador”,comprador_id,cantidad);
End transaction
For each message in queue
  Peek message
  Begin transaction
    Select count(*) as procesado from actualizaciones_aplicadas
       where id_transaccion=message.id_transaccion
       and balance=message.balance
       and id_usuario=message.id_usuario;
    if procesado == 0
      If message.balance == “vendedor”
        Update usuario set cantidad_vendida=cantidad_vendida + message.cantidad

           where id_usuario = message.id_usuario;
      Else
        Update usuario set cantidad_comprada=cantidad_comprada + message.cantidad
           where id_usuario = message.id_usuario;
      End if
      Insert into actualizaciones_aplicadas
         (message.id_transaccion, message.balance, message.id_usuario);
```

```
    End If
  End transaction
  If transaction successful
    Remove message from queue
  End If
End For
```

Ilustración 12: Ejemplo transacción con cola de mensajes persistentes + Tabla de actualizaciones aplicadas

El ejemplo arriba citado depende de la posibilidad de examinar el mensaje en la cola y quitarlo una vez procesado exitosamente. Esto puede ser realizado con dos transacciones independientes: una en la cola de mensajes y otra en la base de datos. Las operaciones encoladas no son persistidas (commit) a menos que lo sean previamente las operaciones de la base de datos. Así, el algoritmo ahora soporta fallas parciales y aún garantiza transaccionalidad sin recurrir al protocolo de commit de 2 fases.

Existe una técnica más simple para asegurar actualizaciones idempotentes, siempre y cuando la única preocupación sea el orden. Sea el mismo esquema del ejemplo anterior pero suponiendo que se desea rastrear la última fecha de venta y compra del usuario:

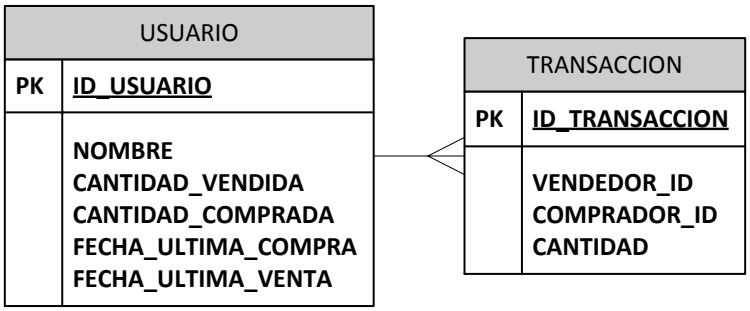

Ilustración 13: Ejemplo esquema Usuario-Transacción considerando última fecha de compra/venta

Se podría utilizar un esquema similar al anterior para actualizar la fecha, pero tendríamos un problema si tuviéramos dos compras que ocurren en un lapso de tiempo muy próximo, y nuestro sistema de mensaje no asegura el orden de las operaciones. Para ese caso, dependiendo del orden en que son procesados los mensajes, se podría tener un valor incorrecto de fecha de última compra.

Esta situación puede ser considerada a través de la siguiente modificación en las sentencias SQL:

```
For each message in queue
  Peek message
  Begin transaction
    Update USUARIO set fecha_ultima_compra=message.fecha_transaccion
      where id_usuario = message.comprador_id
      and fecha_ultima_compra<message.fecha_transaccion;
```

```
  End transaction
  If transaction successful
    Remove message from queue
  End If
End For
```

Ilustración 14: Ejemplo transacción para actualización de última fecha de compra

Es decir, sencillamente no permitiendo que se actualice el campo "fecha_ultima_compra" con fechas antiguas, puede obtenerse operaciones de actualización independientes del orden en que arriban los mensajes. Esta solución también puede utilizarse para evitar cualquier tipo de actualización fuera de un orden establecido, como alternativa a utilizar fechas, podría utilizarse un identificador de transacciones incremental.

3.3.2 Diseño bajo consistencia eventual

En el punto anterior se ha analizado como desacoplar y flexibilizar la consistencia para mejorar la disponibilidad, pero resta analizar cómo impacta esta flexibilidad y consistencia eventual en el diseño de la aplicación.

Desde una visión tradicional de la ingeniería de software, se observa a los sistemas como bucles cerrados o máquinas de estado, donde puede predecirse a cada momento el comportamiento en términos de entradas y salidas. Esto remite a una necesidad en la creación de sistemas de software correctos. En las propiedades BASE se sigue obteniendo dicha predictibilidad, pero requiere realizar una observación global del comportamiento.

Por ejemplo, consideremos un sistema donde los usuarios pueden transferirse activos. El tipo de activo es irrelevante, podría ser dinero u objetos. Para este ejemplo, asumiremos que hemos desacoplado las dos operaciones de dar y recibir activos entre usuarios, a través de una cola de mensajería que proveerá el desacoplamiento.

A primera vista, el sistema parece no determinístico y problemático: pues existe un periodo de tiempo donde el activo ha dejado de pertenecer a un usuario y todavía no ha sido asignado a otro. El tamaño de la ventana de tiempo puede ser determinado dependiendo del diseño del sistema de mensajería, sin embargo, existe un lapso entre el estado inicial y el estado final en el cual ningún usuario es dueño del activo.

Si consideramos esta problemática desde la perspectiva del usuario, este lapso podría no ser relevante o incluso conocido por el usuario. Tanto el receptor como el emisor del activo podrían no conocer cuando se asigna el mismo, pero si el lapso es de solo segundos, será invisible o al menos tolerable para los usuarios que están transfiriendo el activo. En esta situación, el comportamiento del sistema es considerado consistente y aceptado por los usuarios, a pesar de estar bajo una implementación de consistencia eventual.

3.3.3 Arquitectura dirigida por eventos

¿Qué sucedería si se requiere conocer cuando el estado se ha vuelto consistente? En este caso, se podría contar con algoritmos pero que apliquen solo para informar estados consistentes relevantes.

Continuando con el ejemplo anterior, ¿qué sucedería si se necesita notificar al usuario que el activo ha arribado? Se podría crear un evento en la transacción que realiza commit del activo, para que el usuario que recibe el activo pueda realizar un posterior procesamiento basado en el estado alcanzado. Este enfoque puede obtenerse a través de EDA (arquitectura dirigida por eventos), la cual provee importantes mejoras en escalabilidad y desacoplamiento [37].

4. Técnicas y Patrones de diseño

4.1 Introducción

Existe una amplia variedad de técnicas, mecanismos y patrones de diseño utilizados en las bases de datos NoSQL. Algunas aplican a determinados modelos de datos (como MemCache que aplica a almacenamientos clave-valor), mientras que otras pueden emplearse en forma indistinta en cualquier modelo de datos (vector clock, consistent hashing, etc.).

La implementación de estas técnicas varía ligeramente entre los diferentes productos y se halla atada al modelo de datos lógico y físico subyacente, priorización de requerimientos no funcionales y características del hardware entre otros factores.

Las técnicas que se citan en las próximas secciones atacan a más de un requerimiento y el verdadero éxito de su aplicación consiste en encontrar un balance entre ellas.

4.2 Técnicas y Patrones de diseño para escalabilidad y alta disponibilidad

Cuando se prioriza la escalabilidad y la alta disponibilidad, se está dando lugar a utilizar modelos de consistencia eventual, bajo estas consideraciones, los datos se almacenan en esquemas distribuidos entre nodos, donde pueden ser leídos y alterados por cada nodo. Este esquema permite balancear cargas a través de la replicación y distribución de set de datos entre nodos, en las siguientes secciones estos nodos son referidos como "nodos replicas".

4.2.1 Consistencia

Existe una serie de técnicas para considerar las modificaciones concurrentes y las versiones a las cuales los set de datos eventualmente convergen. Algunas de ellas:

- ***Uso de Timestamps:*** el uso de un orden cronológico es la solución más obvia. Sin embargo, los timestamps se basan en relojes sincronizados y no capturan la causalidad [39].

- ***Bloqueo optimista:*** se guarda un único contador o valor de reloj por cada pieza de datos. Cuando un cliente trata de actualizar un set de datos, debe proveer el valor del contador/reloj de la revisión que desea actualizar. El equipo de desarrollo detrás del Proyecto Voldemort ha encontrado una debilidad en esta técnica, al mostrar que no funciona correctamente en un escenario distribuido y dinámico donde hay incorporaciones y salidas abruptas y sin previo aviso de servidores. Para permitir lógica de causalidad entre versiones (que versión es considerada la más reciente por un nodo sobre el cual una actualización ha sido recibida) se debe almacenar, mantener y procesar gran cantidad de información histórica, debido a que el esquema de bloqueo optimista necesita un orden total de los números de versiones para realizar razonamiento de causalidad. Dicho orden total es fácilmente alterado en un sistema distribuido y dinámico [3].

- ***Almacenamiento multiversión:*** Implica almacenar un timestamp por cada celda de una tabla. Estos timestamps no necesariamente se corresponden con la fecha/hora actual, sino que pueden ser valores ficticios bajo un orden

predefinido. Para una fila específica, pueden existir múltiples versiones concurrentes que se proveen como "instantáneas" del set de datos. Bajo esta técnica, además de poder solicitar la última versión de un set de datos, podría solicitarse la última versión de un set de datos previa a una versión N.

El almacenamiento multiversión es una técnica incluida dentro del paradigma de consistencia eventual, pero también puede ser considerada desde la visión de la concurrencia, en cuyo caso forma parte del método de control de concurrencia multiversión (MCC o MVCC) ampliamente utilizada en las bases NoSQL (ejemplo: CouchDB, Berkeley DB, etc.) y sistemas de versionado (ejemplo: Subversion, Git, etc.).

En MVCC cuando se requiere actualizar un ítem de datos, no se sobrescribe la vieja versión con la nueva, sino que se marca como obsoleta y se incluye la nueva versión, de esta forma se almacenan múltiples versiones pero solo una es la última y los lectores pueden acceder al dato que se encontraba cuando empezaron la lectura, incluso si éste fue modificado o borrado parcialmente por otro.

Otra ventaja es que permite a la base de datos evitar la tarea de "rellenar" espacios en memoria o disco, pero requiere que el sistema realice un barrido y borrado de los objetos de datos obsoletos.

En una base de datos orientada a documentos como CouchDB, permite optimizar el almacenamiento y acceso, a través de la escritura contigua en secciones de disco y por lo tanto, cuando se actualiza, el documento completo puede ser reescrito más que bits o piezas mantenidas en una estructura de base de datos no contigua linkeada.

Puede resumirse esta técnica como un "control de concurrencia optimista, con un esquema de comparación e intercambio basado en timestamps" [37].

Implementación

Como se mencionó, MVCC utiliza timestamps o un identificador de transacción incremental para alcanzar la consistencia. MVCC se asegura que una transacción nunca tenga que esperar por un objeto de base de datos a través del mantenimiento de versiones. Cada versión tendrá un timestamp de escritura y permitirá a una transaccion Ti leer la versión más reciente de un objeto que preceda a dicho timestamp TS(Ti).

Si una transacción Ti desea escribir un objeto, y existe otra transacción Tk, el timestamp de Ti debe preceder al timestamp de Tk (TS(Ti) < TS(Tk)) para que la operación de escritura pueda llevarse a cabo. Es decir, que una escritura no puede ser completada si existen transacciones pendientes con un timestamp anterior.

Cada objeto también tendrá un timestamp de lectura, y si una transacción Ti desea escribir un objeto P y el timestamp de dicha transacción es anterior al timestamp de lectura del objeto (TS(Ti) < RTS(P)), la transacción Ti es abortada y reiniciada (pues se intentando escribir una actualización con una versión antigua). Caso contrario, Ti crea una nueva versión de P y asigna los timestamps de lectura y escritura de P con el timestamp de la transacción TS(Ti).

Ejemplo

Supongamos que el estado de la base de datos es el siguiente:

Tiempo	Objeto 1	Objeto 2
1	"Hola" escrito por Tx1	
0	"Foo" escrito por Tx0	"Bar" escrito por Tx0

Ilustración 15: Ejemplo de control de concurrencia multiversión – Primer parte

La transacción Tx0 en el tiempo 0 escribió el Objeto 1 = "Foo" y el Objeto 2 = "Bar". En el tiempo 1, la transacción Tx1 escribió el Objeto 1 = "Hola" y dejo el Objeto 2 con su valor original. El nuevo valor del Objeto 1 reemplazará el valor escrito en el tiempo 0 para todas las transacciones que comiencen después que Tx1 persistió (commit), en ese punto la versión 0 del objeto 1 puede ser barrida y borrada.

Ahora bien, si a continuación una transacción de larga duración Tx2 empieza una operación de lectura del Objeto 1 y 2, luego que Tx1 persistió y existe otra transacción concurrente Tx3 que borra el Objeto 2 y agrega el Objeto 3 = "Foo-Bar", el estado de la base de datos en el tiempo 2 será el siguiente:

Tiempo	Objeto 1	Objeto 2	Objeto 3
2		(borrado) por Tx3	"Foo-Bar" escrito por Tx3
1	"Hola" escrito por		

	Tx1		
0	“Foo” escrito por Tx0	“Bar” escrito por Tx0	

Ilustración 16: Ejemplo de control de concurrencia multiversión – Segunda parte

Dado que Tx2 y Tx3 se ejecutaron concurrentemente, Tx2 observará la versión de la base de datos previa al tiempo 2, es decir, previa a que la transacción Tx3 persista la escritura, esto es Objeto 2=”Bar” y Objeto 1 = “Hola”. Esta es la forma en que MVCC permite aislar “instantáneas” de lectura sin que se generen bloqueos [40].

- ***Vector clock:*** Un vector clock es definido como una tupla V[0], V[1],…,V[n] de valores de reloj pertenecientes a cada nodo. En un ambiente distribuido, el nodo i mantiene una tupla de valores de reloj tales que representan el estado del nodo en sí mismo y los estados “conocido” de los otros nodos (replicas) en un tiempo dado: $V_i[0]$ es el valor del reloj del primer nodo, $V_i[1]$ es el valor del reloj del segundo nodo,… $V_i[i]$ es el valor del reloj para el nodo en sí mismo,… $V_i[n]$ es el valor del reloj para el ultimo nodo. Los valores de reloj pueden ser timestamps obtenidos de un reloj local perteneciente a un nodo, números de versión/revisión o algún otro tipo de valores ordenados [3].

  Así, por ejemplo, el vector clock del nodo número 2 podría tomar los siguientes valores:

  $$V_2[0] = 45,\ V_2[1] = 3,\ V_2[2] = 55$$

  Es necesario aclarar que las tuplas de valores de reloj referencian a un mismo set de datos. Entonces, desde la perspectiva del nodo 2, esto se interpreta: una actualización en el nodo 1 produjo la revisión 3, una actualización en el nodo 0 produjo la revisión 45 y finalmente la actualización más reciente fue realizada por el nodo 2 y produjo la revisión 55 [3].

  La actualización de los vector clocks es realizada en base a las siguientes reglas:

- Si se ejecuta una operación interna al nodo i, éste incrementa su reloj $V_i[i]$. Esto significa que las actualizaciones internas son reflejadas inmediatamente por el nodo en ejecución.

- Si el nodo i envía un mensaje al nodo k, primero avanza su propio vector clock $V_i[i]$ y luego adjunta el vector clock V_i al mensaje para el nodo k. De esta forma, el nodo i le dice al nodo receptor sobre su estado interno y el estado que él conoce de los otros nodos en el momento en que envío el mensaje.

- Si el nodo i recibe un mensaje del nodo j, primero avanza su propio vector clock $V_i[i]$ y luego combina su vector clock con el vector clock adjunto en el mensaje del nodo j ($V_{mensaje}$), en base al siguiente criterio:

$$\mathbf{V_i = max(V_i, V_{mensaje})}$$

Para comparar los vector clocks V_i y V_j ($V_{mensaje}$) y lograr un ordenamiento parcial y causal, se aplica la siguiente regla:

$$\mathbf{V_i > V_j \Leftrightarrow \forall k\ (V_i[k] \geq V_j[k]) \wedge \exists k'\ (V_i[k'] \geq V_j[k'])}$$

Es decir, V_j será menor a V_i si y solo si, para cada nodo k, el valor del vector clock de V_j es menor o igual al valor del vector clock de V_i, y si al menos una de esas relaciones es estrictamente menor. En esta relación se considera que $j \rightarrow i$, esto denota que el evento j sucedió previo al evento i (relación de causalidad) [41].

Si no se cumple la relación es porque se ha generado un conflicto por actualizaciones concurrentes y debe ser resuelto por ejemplo por la aplicación cliente (la resolución dependerá del enfoque seguido).

La siguiente ilustración muestra la distribución en los vector clocks:

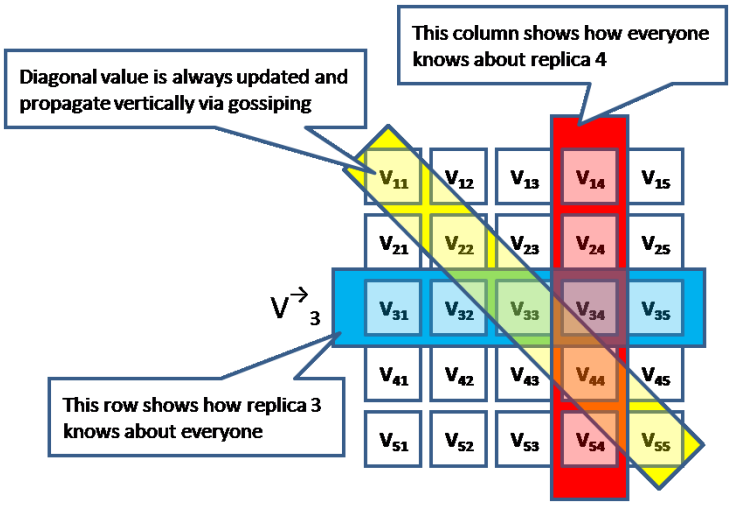

Ilustración 17: Vector clocks

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

Los vector clocks pueden ser utilizados para resolver la consistencia de escritura entre múltiples replicas, debido a que aplican razonamiento causal entre las actualizaciones. En general, las aplicaciones clientes participan de esta estrategia manteniendo un vector clock de la última réplica del nodo con el que han interactuado y utilizan este vector clock en sus funciones dependiendo del modelo de consistencia requerido: por ejemplo para el modelo de consistencia de lectura monótona, presentado anteriormente, una aplicación cliente adjunta su ultimo vector clock (recibido en interacciones previas) y el nodo replica contactado se asegura de responder con un vector clock mayor que el enviado por el cliente. Esto permite que el cliente pueda estar seguro de recibir solo nuevas versiones del set de datos (nuevas comparadas con las versiones que tenía previamente).

Las ventajas de vector clock en comparación con las técnicas de timestamps, bloqueo optimista y almacenamiento multiversión son las siguientes:

- No depende de relojes sincronizados.
- No requiere un ordenamiento total de los números de revisión para realizar un razonamiento causal.
- No necesita almacenar y mantener múltiples versiones/revisiones de cada pieza de datos en todos los nodos.

- ***Protocolo Gossip:*** Es un protocolo de comunicación P2P basado en distribución de información en forma epidémica [42].

  El concepto de la comunicación gossip (rumor) puede ser ilustrado a partir de la analogía de compañeros de oficina esparciendo rumores: Supongamos que cada hora los compañeros de trabajo se juntan alrededor de la máquina de café. Cada empleado se junta con otro, elegido al azar y comparte el último rumor. Asi, por ejemplo, al día siguiente, Alice empieza un nuevo rumor: ella comenta a Bob que piensa que Charlie afeito su bigote. En la siguiente reunión, Bob le dice a Dave, mientras Alice repite la idea a Eve. Después de cada encuentro en la máquina de café, el número de individuos que ha escuchado el rumor se duplica (no se considera la situación cuando se cuenta el rumor dos veces a la misma persona, por ejemplo si Alice intenta contar el rumor a Frank y Frank ya lo escucho de Dave).

  La fortaleza del protocolo Gossip radica en la propagación, siguiendo la analogía, si Dave tuvo problemas en entender el rumor cuando fue transmitido por Bob, probablemente pueda interpretar el rumor cuando se encuentre con alguien más [42].

  La aplicación de éste protocolo provee escalabilidad y evita un único punto de falla (SPOF). Sin embargo, se observa como desventaja que en un cluster de n nodos se requiere una cantidad de interacciones de orden logarítimico y solo puede proveerse consistencia eventual [39].

  El protocolo Gossip debe satisfacer las siguientes condiciones:

  - El protocolo requiere interacciones periódicas, por parejas y entre nodos.
  - La información intercambiada durante estas interacciones es de tamaño limitado.
  - Durante estas interacciones el estado de al menos uno de los participantes cambia para reflejar el estado del otro.
  - No se asume la comunicación confiable.
  - La frecuencia de las interacciones es baja comparada con la latencia en mensajes típicos, de esta forma los costos del protocolo son reducidos.

- Existe cierta aleatoriedad en la selección de pares. Los pares pueden ser seleccionados de todo el conjunto de nodos o de un conjunto reducido de nodos adyacentes.

El protocolo Gossip puede utilizarse en forma conjunta con los vector clocks y de esta forma resolver la consistencia, conflictos de versiones y propagar estados/operaciones entre replicas en una base de datos particionada [3].

Existen diferentes implementaciones de Gossip dependiendo del modelo de transferencia:

Modelo de transferencia de estados: En un modelo de transferencia de estados, datos o deltas de datos son intercambiados entre clientes y servidores y entre servidores. En este modelo, los nodos servidores de base de datos mantienen vector clocks para sus datos y árboles de versiones de estados para la resolución de conflictos de versiones (esto para resolver los conflictos de actualizaciones concurrentes mencionados en vector clock), mientras que las aplicaciones clientes mantienen vector clocks para los set de datos que ya han requerido o actualizado.

El intercambio y procesamiento se realiza de la siguiente manera:

- *Procesamiento de consultas:* Cuando una aplicación cliente consulta por un set de datos, envía su propio vector clock junto a la consulta. El nodo servidor responde con parte de su árbol de estados para el set de datos que precede al vector clock adjuntado a la consulta del cliente (de esta forma se garantiza la consistencia de lectura monótona) y con el vector clock del servidor. A continuación, el cliente incrementa su vector clock a través de la combinación con el vector clock del nodo servidor que fue adjuntado a la respuesta. Para este paso, el cliente además debe resolver los potenciales conflictos de versiones, la resolución es necesaria en tiempo de lectura para evitar que opere o envíe actualizaciones sobre una revisión de datos desactualizada comparada con la revisión que el nodo replica mantiene [3].

En la siguiente figura se ilustra el intercambio de mensajes de consulta para el modelo de transferencia de estados con vector clocks:

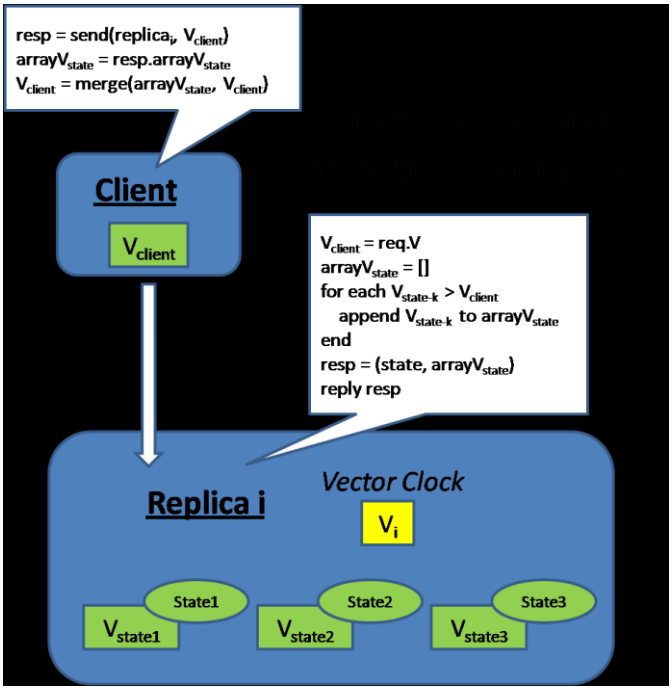

Ilustración 18: Consulta en modelo de transferencia de estados con vector clocks

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

- *Procesamiento de actualizaciones:* Como en el caso de las consultas, los clientes deben adjuntar su vector clock del set de datos a ser actualizado junto con el pedido de actualización. El servidor replica contactado valida si el estado del cliente (de acuerdo al vector clock transmitido) precede al estado del servidor y si es así omite el pedido de actualización (ya que el cliente debe adquirir la última versión y resolver los conflictos de versiones en tiempo de lectura). Si el cliente transmitió un vector clock mayor que el que posee el servidor, entonces se ejecuta la actualización [3].

  En la siguiente figura se ilustra el intercambio de mensajes de actualización para el modelo de transferencia de estados con vector clocks:

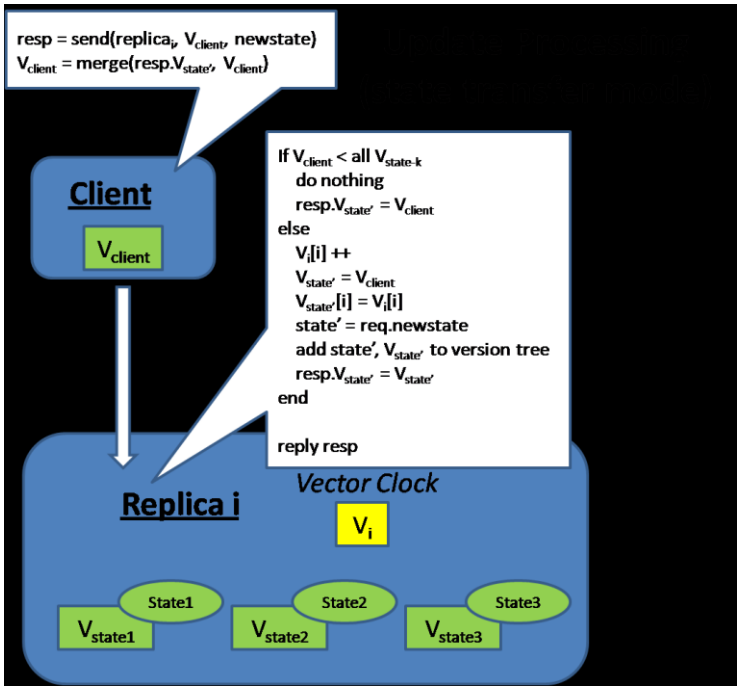

Ilustración 19: Actualización en modelo de transferencia de estados con vector clocks

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

- *Gossip entre nodos:* Las réplicas responsables de las mismas particiones de datos intercambian sus vector clocks y árboles de versiones y tratan de combinarlos para mantenerlos sincronizados.

  En la siguiente figura se ilustra el intercambio de mensajes entre nodos para el modelo de transferencia de estados con vector clocks:

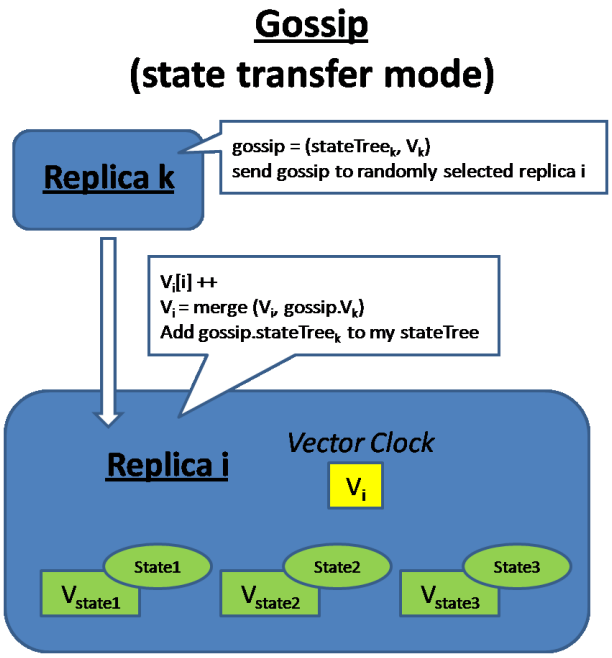

Ilustración 20: Gossip entre nodos en modelo de transferencia de estados con vector clocks

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

Modelo de transferencia de operaciones: En contraste con el modelo de transferencia de estados, las operaciones aplicables a datos mantenidos localmente son comunicadas entre nodos. Una obvia ventaja es la reducción del ancho de banda consumido en el intercambio de operaciones.

En el modelo de transferencia de operaciones, es particularmente importante aplicar las operaciones en el orden correcto en cada nodo. Por lo tanto, un nodo replica primero debe determinar la relación causal entre operaciones (a través de una comparación de vector clocks) antes de aplicarlas a los datos. A continuación, el nodo debe posponer la aplicación de operaciones hasta que todas las operaciones precedentes hayan sido ejecutadas, esto implica mantener una cola de operaciones demoradas o log que debe ser intercambiado y consolidado con los correspondientes de las otras replicas [3].

Se debe diferenciar "orden causal" que refiere a aplicar cambios a las "causas" antes de aplicar cambios a los "efectos", de "orden total" que refiere aplicar la operación en la misma secuencia [39].

En este modelo, un nodo replica mantiene los siguientes vector clocks:

- V_{state}: es el vector clock correspondiente a la última actualización de estado de un dato.

- V_i: es el vector clock para el nodo en sí
- V_j: es el vector clock recibido del último mensaje Gossip del nodo j (para cada uno de los nodos replicas)

El intercambio y procesamiento se realiza de la siguiente manera:

- *Procesamiento de consultas:* Cuando una aplicación cliente consulta por un set de datos, envía su propio vector clock junto a la consulta. El nodo servidor contactado determina si posee una vista causalmente posterior a la recibida en la consulta del cliente y responde a éste con la última vista del estado, ya sea el vector clock enviado por el cliente o el correspondiente a un vector clock posterior al del nodo en sí.

  En la siguiente figura se ilustra el intercambio de mensajes de consulta para el modelo de transferencia de operaciones con vector clocks:

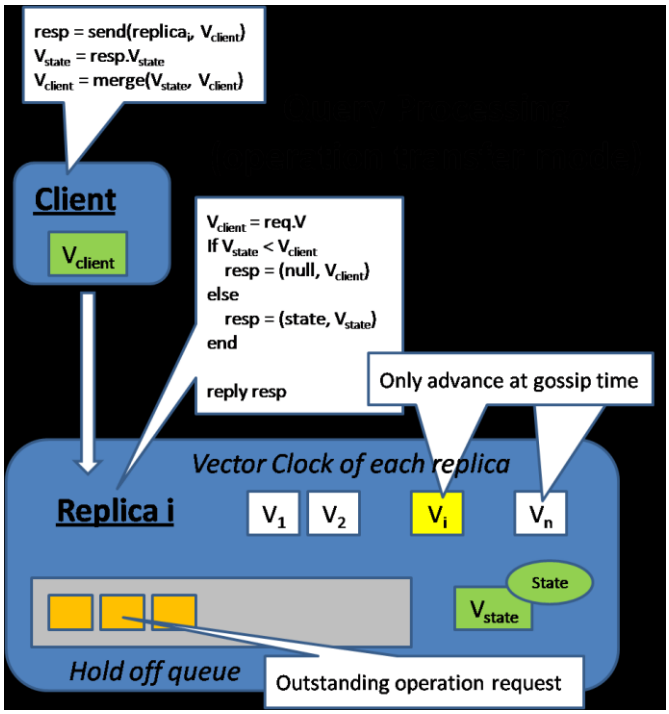

Ilustración 21: Consulta en modelo de transferencia de operaciones con vector clocks

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

- *Procesamiento de actualizaciones:* Cuando un cliente envía un pedido de actualización, el nodo replica contactado almacena esta operación hasta que pueda ser aplicada a su estado local, teniendo en cuenta los tres

vector clocks y la cola de operaciones ya almacenadas. La operación de actualización almacenada es etiquetada en dos vector clocks: V_{client}, que representa la vista del cliente cuando se envió la actualización, y $V_{received}$ que representa la vista del nodo replica cuando recibió la actualización, es decir, V_i. Recién cuando todas las operaciones causalmente precedentes a la operación de actualización recibida han arribado y han sido aplicadas, la operación de actualización para ese cliente puede ser ejecutada.

En la siguiente figura se ilustra el intercambio de mensajes de actualización para el modelo de transferencia de operaciones con vector clocks:

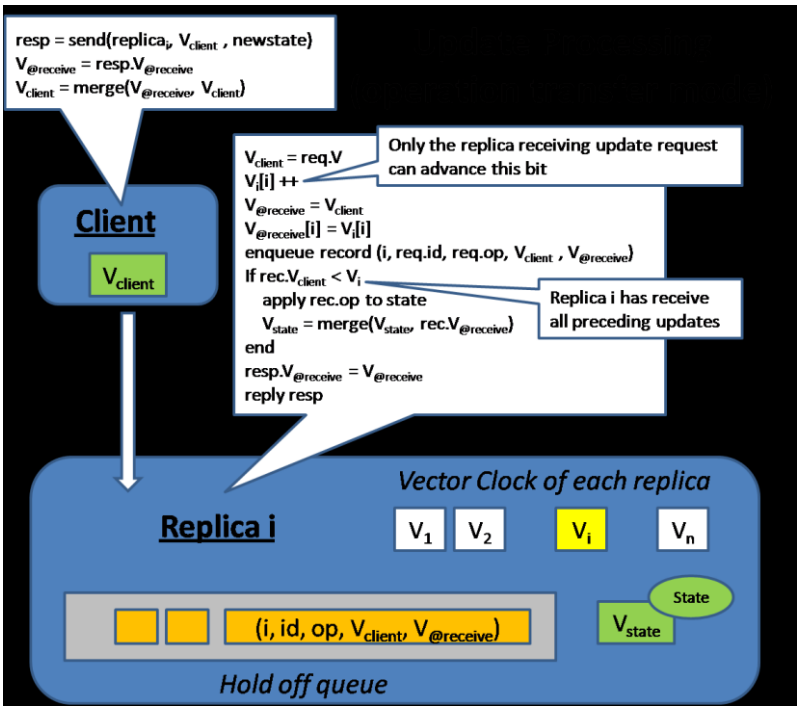

Ilustración 22: Actualización en modelo de transferencia de operaciones con vector clocks

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

- *Gossip entre nodos:* En el modelo de transferencia de operaciones, los nodos replican intercambian las colas o logs de operaciones pendientes y cada uno actualiza sus vector clocks. Así, al finalizar el intercambio, cada nodo decide si ciertas operaciones de la cola pueden ser aplicadas (ya que todas las operaciones dependientes han sido recibidas). Si más de una operación puede ser ejecutada a la vez, el nodo replica establece el orden causal correcto de las operaciones analizando los vector clocks

adjuntos a las operaciones y aplica dichas operaciones en su estado de datos local. En caso de actualizaciones concurrentes en diferentes nodos replicas (resultando en múltiples secuencias de operaciones validas a la vez), se debe hacer una distinción entre los siguientes casos:

- **Actualizaciones concurrentes conmutativas:** en este caso no importa el orden de la aplicación.
- **Actualizaciones concurrentes no conmutativas:** en este caso el orden parcial establecido a través de la comparación de vector clocks no es suficiente sino que se requiere determinar un orden total. Una variante implica introducir un numero de secuencia global obtenido de un contador central.

Luego de aplicar la actualización localmente, la operación no debe ser quitada inmediatamente de la cola porque podría no haberse intercambiado entre todas las réplicas aún. Por lo tanto, se debe chequear continuamente el vector clock de cada una de las réplicas luego del intercambio de las colas y recién luego de confirmar que todos los nodos han recibido la operación de actualización podemos quitarla de la cola.

En la siguiente figura se ilustra el intercambio de mensajes entre nodos para el modelo de transferencia de estados con vector clocks:

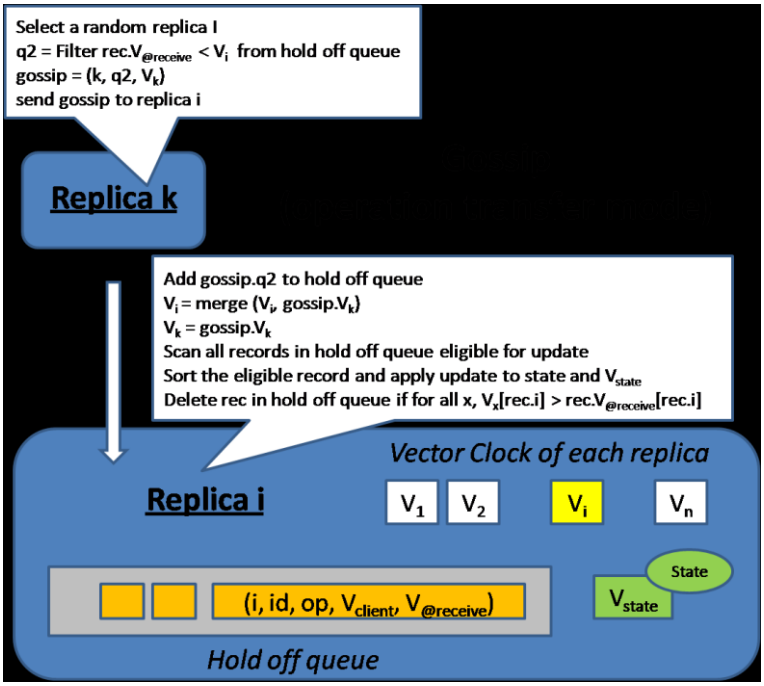

Ilustración 23: Gossip entre nodos en modelo de transferencia de operaciones con vector clocks

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

4.2.2 Particionamiento

El particionamiento de datos es una tarea usualmente requerida en sistemas de gran escala que exceden la capacidad de almacenamiento de datos en un solo equipo. El particionamiento busca asegurar la confiabilidad y permitir el escalamiento a través del balanceo de cargas.
Existen diferentes técnicas de particionamiento dependiendo del tamaño del sistema y factores como el dinamismo (frecuencia con la que nodos de almacenamiento se incorporan o abandonan un clúster de nodos).

A continuación se citan algunas técnicas:

- ***Memory Caches:*** puede ser visto como un particionamiento transitorio de bases de datos a memoria, ya que se replican las partes de la base de datos más frecuentemente requeridas a memoria principal, reduciendo la latencia en las consultas de los clientes.

  Una implementación es el protocolo memcached, que consiste en un arreglo de procesos con una cantidad de memoria asignada, que pueden ser ejecutados en varias máquinas en una red y son configurados a través de la aplicación. El protocolo memcached posee implementaciones en los diferentes lenguajes de programación y se utiliza en las aplicaciones clientes a través de una API de almacenamiento clave-valor, proveyendo las operaciones básicas get(key), put(key,value), remove(key).
  Memcached almacena objetos en la memoria cache a través de una clave calculada tras aplicar una función de hash contra las instancias configuradas de memcached. Si un proceso de memcached no responde, la mayoría de las implementaciones de la API ignoran el proceso/nodo que no responde y utilizan en su lugar los nodos activos, lo cual lleva a recalcular implícitamente el hashing de los objetos en cache, ya que parte de ellos se encuentran referenciando a un servidor de memcached diferente luego de la caída del nodo. Finalmente cuando el nodo caído vuelve a unirse al arreglo de servidores memcached, se recalcula la función de hash para parte de las claves de datos, se asignan al nuevo nodo incorporado y se quitan de memoria los objetos de los servidores de memcached que habían recibido la redistribución durante la caída.
  Memcached aplica una estrategia de los objetos menos utilizados recientemente (LRU) para la limpieza de la memoria cache y adicionalmente permiten especificar timeouts para objetos [3].

  Los siguientes pseudocodigos muestran como difiere un código de consulta con la implementación de memcached [43]:

```
function get_foo(int userid)
{
   result = db_select("SELECT * FROM users WHERE userid = ?", userid);
   return result;
}
```

Ilustración 24: Ejemplo de consulta sin Memcached

```
function get_foo(int userid)
{
   /* primero miramos en la cache */
   data = memcached_fetch("registro:" + userid);


   if (!data)
   {
      /* no se ha encontrado en cache, buscar en base de datos */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);

      /* almacenamos en la caché para la próxima consulta */
      memcached_add("registro:" + userid, data)
   }
   return data;
}
```

Ilustración 25: Ejemplo de consulta con Memcached

Se realiza la búsqueda primero en Memcached y si existe la clave "registro:userid", se utiliza ese valor. Si no encuentra resultado, aplicará la consulta contra la base de datos y almacenará la clave/valor en Memcached para posteriores consultas. Para la limpieza de la caché es necesario incluir una función de actualización de datos para evitar tener datos obsoletos:

```
function update_foo(int userid, string dbUpdateString)
{
   /* primero actualizamos la base de datos */
   result = db_execute(dbUpdateString);

   if (result)
   {
      /* actualización exitosa: obtener datos de la base de datos para actualizar cache */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);

      /* almacenamos en la caché para la próxima consulta */
      memcached_set("registro:" + userid, data)
   }
}
```

Ilustración 26: Ejemplo de sincronización con Memcached

Si la consulta a la base de datos fue exitosa, se actualiza el dato almacenado en la caché con el nuevo dato de la base de datos. Otra opción es invalidar la cache con la función "delete", de forma que las siguientes consultas a la caché no encuentren el dato y se fuerce a que la base de datos sea consultada [43].

Existen otras implementaciones de memory caches como las realizadas por servidores de aplicaciones como JBoss [3].

- ***Clustering:*** consiste en contar con un conjunto de servidores de bases de datos unidos mediante una red de alta velocidad, de tal forma que desde la visión de la aplicación cliente, el conjunto es visto como un único servidor. De un clúster se espera que presente las siguientes prestaciones a un costo relativamente bajo [44]:

  - Alto rendimiento
  - Alta disponibilidad
  - Balanceo de carga
  - Escalabilidad

  Si bien este enfoque puede ayudar a escalar la capa de persistencia de un sistema, existen críticas a las características intrínsecas del clustering. Michael Stonebraker et. al. consideran que son solo un agregado a las DBMS que no fueron diseñadas originalmente para manejar la distribución [6].

- ***Separación de lecturas y escrituras:*** consiste en especificar uno o más servidores dedicados. Las operaciones de escritura para todo o parte de los datos son ruteadas hacia el/los servidor/es maestro/s y un número de servidores replicas esclavos satisfacen las operaciones de lectura. Existen dos alternativas:

  - El maestro replica las operaciones de escritura a los esclavos en forma asincrónica (luego de ser aplicada en el mismo maestro): En este caso si el maestro falla antes de completar la replicación para al menos un esclavo, existe una ventana de tiempo de pérdida de datos.

  - El maestro replica las operaciones de escritura a los esclavos en forma sincrónica: En este caso el maestro esperará hasta que la operación sea propagada a al menos un servidor esclavo.

  Si el cliente puede tolerar cierto grado de estancamiento de datos las operaciones de lectura pueden ir a cualquier replica permitiendo un balanceo de carga entre

esclavos para las operaciones de lectura. Si el cliente no tolera dicho estancamiento, necesariamente se debe rutear la petición al maestro.

Bajo este modelo no existe un nodo físico particular ejerciendo el rol de maestro, sino que la asignaciones de roles sucede a nivel de nodo virtual. Cada nodo físico posee varios nodos virtuales y más de uno podría tomar el rol de maestro para algunas particiones, mientras que otros nodos virtuales pueden tomar el rol de esclavos. Por lo tanto, la carga en las operaciones de escritura también es distribuida entre diferentes nodos físicos, pero esto debido al particionamiento y no a la replicación. Cuando un nodo físico falla, los nodos virtuales maestros de ciertas particiones se perderán, por estos casos, usualmente se asigna como nuevo maestro al nodo esclavo más actualizado [45].

El modelo maestro/esclavo para la separación de lecturas y escrituras funciona bien si la tasa de lecturas/escrituras es alta. La replicación de datos puede realizarse a través de transferencia de estados o transferencia de operaciones [45].

- ***Sharding:*** implica particionar los datos de forma tal que un set de datos frecuentemente requeridos y actualizados juntos, residan en el mismo nodo, y el volumen de datos total sea distribuido en forma relativamente uniforme entre los servidores (en función de la capacidad de procesamiento de cada equipo).

  Los shards o particiones de datos pueden ser replicados por motivos de confiabilidad y balanceo de carga y podría contarse con una réplica dedicada solo a las operaciones de escritura o con múltiples replicas que mantengan una partición de datos. Para permitir tal escenario de sharding, debe existir un mapeo entre particiones de datos (shards) y nodos de almacenamiento responsables de esas particiones. Este mapeo puede ser estático o dinámico y puede ser determinado por una aplicación cliente, por un componente/servicio dedicado o por una infraestructura de red entre la aplicación cliente y los nodos de almacenamiento.

  Una desventaja de la aplicación de sharding es que no es posible la aplicación de "joins" entre shards de datos, por lo tanto la aplicación cliente o una capa proxy debe realizar varias peticiones y pos procesamiento (filtrados, agregaciones, etc.) de los resultados.

  Algunos autores consideran que la aplicación de sharding implica perder todas las características que hacían útiles a los RDBMS y que además es operacionalmente odioso. Esta valoración refiere al hecho que sharding no fue incluido en el diseño original de RDBMS sino incluido posteriormente como una alternativa al particionamiento. Contrariamente, muchas bases de datos

NoSQL han incluido sharding como una funcionalidad clave e incluso algunas como MongoDB proveen particionamiento y balanceo de datos automático entre nodos [3].

Para la aplicación de sharding es necesario decidir una política de particionamiento y como se realizara el mapeo de los objetos a los servidores. Una alternativa es aplicar una función de hash para las claves primarias de los objetos persistidos contra el set de nodos disponibles:

partición = hash(o) mod n

Donde:

o = objeto a aplicar función de hash
n = número de nodos activos

El lado negativo de este procedimiento es que parte de los datos deben ser redistribuidos cuando un nodo abandona o se incorpora al clúster de nodos activos. En un escenario donde se conserve la información en cache, la redistribución de datos podría ejecutarse implícitamente si no se encontraran las entradas en cache, forzando a leer los datos nuevamente de la base de datos y aplicando hashing contra los servidores de cache actualmente disponibles y posteriormente permitiendo que la información inválida de cache sea limpiada con alguna política de limpieza como LRU. El problema con este enfoque es que no es aceptable para almacenamientos de datos ya persistidos, dado que éstos no se encuentran presentes en los nodos disponibles y no pueden ser reconstruidos, para esta situación debe considerarse una solución alternativa como Consistent Hashing [3].

- ***Consistent Hashing:*** La idea de Consistent Hashing fue introducida por David Karger et. al. en 1997 en el contexto de un paper relacionado a una familia de protocolos de cache para redes distribuidas que podían ser utilizadas para decrementar o eliminar la ocurrencia de hot spots en la redes [3]. Posteriormente, Consistent Hashing fue adoptado en otros campos tales como la implementación de tablas de hashing distribuidas en Chord [46], clientes de Memcached e integrada en bases de datos como Dynamo y Proyecto Voldemort [3]. Consistent Hashing posee implementaciones en los siguientes lenguajes: C++, C#, Erlang, Go, Java, PHP, Python, Ruby.

  La idea detrás del algoritmo de Consistent Hashing reside en aplicar la misma función de hash tanto a los objetos como al caché. De esta forma los nodos reciben un intervalo del rango de la función de hash y los nodos adyacentes

pueden encargarse de parte del intervalo de sus vecinos si estos abandonan el cluster y asignar parte de su propio intervalo si un nuevo nodo se incorpora en forma adyacente.
Consistent Hashing permite que las aplicaciones clientes puedan calcular con que nodo se deben contactar para requerir o escribir un set de datos, sin necesidad de contar con un servidor de metadata que contenga el mapeo entre servidores de storage y particiones de datos como es el caso de Google File System (GFS) [3].

A continuación se presenta la lógica de Consistent Hashing:

Supongamos la siguiente distribución de nodos y objetos, donde contamos con tres nodos A, B y C (cada uno con un caché) y cuatro objetos 1, 2, 3 y 4, éstos últimos son mapeados a un rango resultante de una función de hash la cual es representada a través de una distribución en anillo [47]:

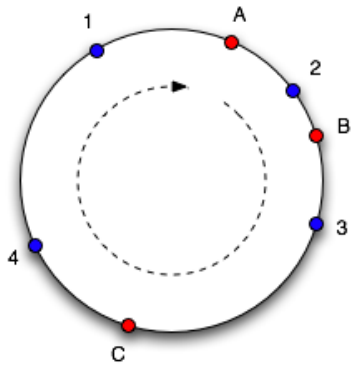

Ilustración 27: Consistent Hashing – Situación inicial

Ilustración de https://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

Que objeto es mapeado a que nodo es determinado por el movimiento alrededor del anillo en sentido horario. Por lo tanto, los objetos 4 y 1 son mapeados al nodo A, el objeto 2 es mapeado al nodo B y el objeto 3 es mapeado al nodo C. Cuando un nodo abandona el sistema, los objetos en caché serán mapeados a sus nodos adyacente en sentido horario en el anillo, y cuando un nodo se incorpore al sistema, éste se incluirá en el hashing del anillo y tomará objetos de sus nodos adyacentes. En la siguiente figura puede verse el impacto de quitar e incorporar nodos al anillo:

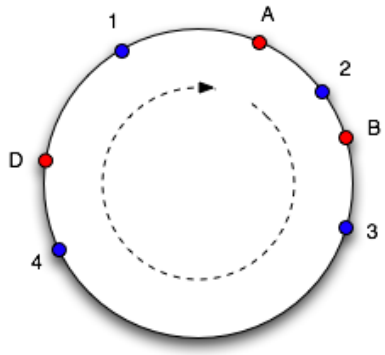

Ilustración 28: Consistent Hashing – Incorporación y partida de nodos

Ilustración de https://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

Si el nodo C abandona el anillo y el nodo D se incorpora a éste, los objetos 3 y 4 serán mapeados al nodo D. Este ejemplo muestra que modificar el número de nodos no requiere volver a mapear todos los objetos al nuevo set de nodos, sino solo parte de ellos.

Este enfoque de Consistent Hashing presenta algunos problemas: primero, la distribución de nodos en el anillo es aleatoria ya que sus posiciones son determinadas por una función de hash y los intervalos entre nodos podrían estar desbalanceados, lo que implica un desbalance en la distribución de objetos de cache en estos nodos (como puede verse en la Ilustración 28, donde el nodo D tiene asignado un rango intervalo mayor que el asignado para los nodos A y B). Una alternativa a éste problema es incorporar en el hashing un número de réplicas para cada nodo físico en el anillo, estas réplicas se conocen como *nodos virtuales*. En la siguiente figura se observa una distribución utilizando nodos virtuales:

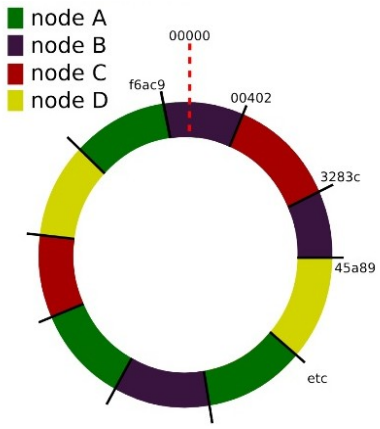

Ilustración 29: Consistent Hashing – Aplicación de nodos virtuales

Ilustración de http://es.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases

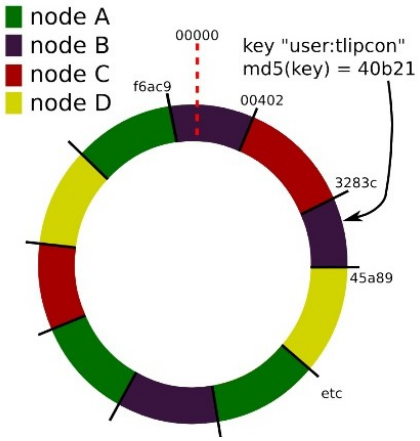

Ilustración 30: Consistent Hashing – Asignación de objetos a nodos virtuales

Ilustración de http://es.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases

La ventaja de esta alternativa es que el número de nodos virtuales para un nodo físico puede ser definido individualmente en función de la capacidad de hardware (cpu, memoria, capacidad de almacenamiento) y no requiere ser el mismo para todos los nodos físicos.

El ingeniero Tom White presenta en su blog un ejemplo de implementación de Consistent Hashing en Java y una posterior simulación considerando nodos virtuales [47]:

```
import java.util.Collection;
import java.util.SortedMap;
import java.util.TreeMap;

public class ConsistentHash<T> {

 private final HashFunction hashFunction;
 private final int numberOfReplicas;
 private final SortedMap<Integer, T> circle = new TreeMap<Integer, T>();

 public ConsistentHash(HashFunction hashFunction, int numberOfReplicas,
                       Collection<T> nodes) {
  this.hashFunction = hashFunction;
  this.numberOfReplicas = numberOfReplicas;

  for (T node : nodes) {
    add(node);
  }
 }

 public void add(T node) {
  for (int i = 0; i < numberOfReplicas; i++) {
    circle.put(hashFunction.hash(node.toString() + i), node);
  }
 }
```

```
public void remove(T node) {
  for (int i = 0; i < numberOfReplicas; i++) {
    circle.remove(hashFunction.hash(node.toString() + i));
  }
 }

 public T get(Object key) {
  if (circle.isEmpty()) {
    return null;
  }
  int hash = hashFunction.hash(key);
  if (!circle.containsKey(hash)) {
    SortedMap<Integer, T> tailMap = circle.tailMap(hash);
    hash = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();
  }
  return circle.get(hash);
 }

}
```

Ilustración 31: Consistent Hashing - Ejemplo de implementación

Ilustración de https://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

En esta implementación el anillo es representado como un mapa ordenado de enteros (`SortedMap`), que contendrá los valores de hash y los nodos (`T`).
Cuando un objeto `ConsistentHash` es creado, cada nodo es agregado al anillo tantas veces como lo indique `numberOfReplicas`. La ubicación de cada replica es elegida a través del hashing del nombre del nodo junto a un sufijo numérico (`i`), y el nodo es almacenado en cada una de estas instancias en el mapa.
Para obtener un objeto (método `get`), se utiliza el valor de hash del objeto para buscar en el mapa. La mayor parte del tiempo no existirá un nodo almacenado en este valor de hash (dado que el espacio asignado a los valores de hash típicamente será mucho mayor que el número de nodos, incluso considerando los nodos virtuales), por lo tanto, para estos casos, el nodo será buscado a partir de la primer clave que figure en la cola del mapa. Si la cola del mapa esta vacía entonces obtenemos la primera clave del anillo.

Tom White realiza una simulación en base al código de la figura 31, almacenando 10000 objetos en 10 nodos y utilizando nodos virtuales:

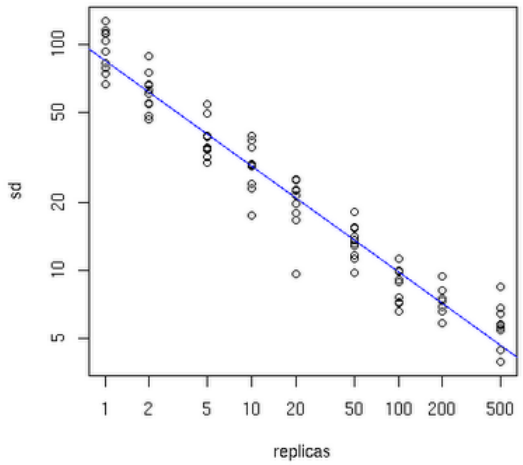

Ilustración 32: Consistent Hashing – Balanceo con nodos virtuales

Ilustración de https://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

En el eje X se representa el número de réplicas o nodos virtuales y en el eje Y se representa la desviación estándar, en ambos casos la escala es logarítmica.
Cuando el número de réplicas es pequeño, la distribución de objetos entre nodos se encuentra desbalanceada, dado que la desviación estándar como un porcentaje del número medio de objetos por nodo es alto. A medida que el número de réplicas se incrementa, la distribución de objetos se vuelva más balanceada. En esta simulación se alcanza un nivel aceptable de balance cuando se cuenta con una cantidad de réplicas del orden de las 100 o 200 (para ese caso la desviación estandar está entre 5%-10% de la media).

Otra consideración en la aplicación de Consistent hashing para almacenamientos persistentes, es que si un nodo abandona el anillo, los datos almacenados en ese nodo se vuelven inaccesibles a menos que hayan sido replicados previamente a otros nodos, y por otro lado, cuando se incorpora un nuevo nodo, los nodos adyacentes dejan de ser responsables de algunos set de datos los cuales aún almacenan pero ya no les son solicitados debido a que los objetos correspondientes ya no son solicitados a éstos a través de la función de hash. Para estos problemas una alternativa es introducir un factor de replicación (r). De esta forma, no solo el nodo siguiente es responsable de un objeto, sino también los siguientes r nodos físicos en sentido horario [48].

El Proyecto Voldemort ha implementado esta técnica como se ilustra a continuación:

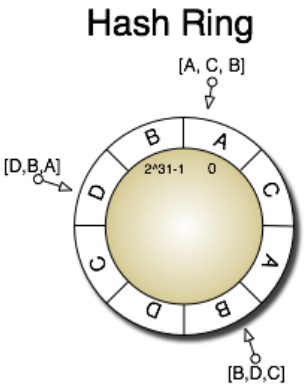

Ilustración 33: Consistent Hashing – Aplicación de nodos virtuales y replicación de datos

Ilustración de http://www.project-voldemort.com/voldemort/design.html

Las letras A, B, C y D representan nodos de almacenamiento, los cuales de acuerdo al concepto de nodos virtuales, son mapeados múltiples veces en el anillo, y los círculos con flechas representan objetos de datos que son mapeados en el anillo en la posición indicada. En el ejemplo, los valores posibles para hash van desde 0 hasta 2^{31}-1 y el factor de replicación es tres, por lo que por cada objeto de dato, tres nodos físicos son responsables (esto se indica a través de los grupos entre corchetes) [48].

Relación número de réplicas vs. Operaciones de lectura/escritura

Como se mencionó la introducción de réplicas en un esquema particionado además de traer beneficios de confiabilidad, permite distribuir las cargas de las operaciones de lectura, ya que las consultas pueden impactar en cualquier nodo físico responsable de un set de datos requerido. Sin embargo, la posibilidad de balanceo de cargas de operaciones de lecturas no aplica a escenarios donde los clientes deben decidir entre múltiples versiones de un set de datos y por lo tanto deben leer de un quorum de servidores, lo cual termina reduciendo la ventaja del balanceo de cargas.

El equipo del Proyecto Voldemort establece que si se desea mantener un modelo de consistencia tal como "lee tus propias escrituras" (RYOW), se debe cumplir la siguiente relación:

$$\mathbf{R + W > N}$$

Donde:

N representa el número de réplicas para un dato a ser leído o escrito

R representa el número mínimo de nodos a ser contactados o considerados en operaciones de lectura
W representa el número mínimo de nodos a ser contactados o considerados en operaciones de escritura. [48]

Bajo el esquema de quorum o votación, además se debe cumplir la siguiente relación:

$$\mathbf{W > N/2}$$

Así, por ejemplo, suponiendo una estructura de 5 copias en 5 nodos: A, B, C, D y E. Si se desconecta el nodo A del resto de los nodos y una operación de escritura actualiza el nodo A, los otros nodos ya no serán consistentes con A, por lo que cualquier operación de lectura sobre estos nodos retornará un dato obsoleto. Entonces, ¿cuantas copias deberíamos leer o escribir para mantener la consistencia?

Para el ejemplo considerado se debe satisfacer las siguientes dos relaciones [49]:

$$\mathbf{W > 5/2 \wedge R > 5 - W}$$

Los pares que cumplen esta relación son los siguientes: (1,5), (2,5), (3,5), (4,5), (5,5),…, (2,4), (3,4), (4,4), (5,4), (3,3).

Si R=1 y W=5, entonces podemos leer de cualquier nodo, pero una escritura implica actualizar los 5 nodos. Bajo esta relación, cada operación de lectura siempre obtendrá el dato actualizado.

Si mantenemos W=5 e incrementamos R, se reduce la velocidad de lectura (pues debemos leer más nodos para asegurar la consistencia). Si bajo esta relación desconectamos el nodo A, podemos seguir leyendo solo de un nodo pero ya no puede alcanzarse el nivel de escrituras requerido.

Si R=3 y W=3, solo se requiere actualizar 3 copias, pero se debe leer 3 nodos para asegurar la consistencia. Si bajo esta relación desconectamos el nodo A, se puede mantener la cantidad de lecturas y escrituras pero no es posible leer ni escribir sobre el nodo A.

Como se observa, la elección de los valores de R y W afectan la performance y disponibilidad, si en el sistema hay más lecturas que escrituras es preferible elegir un valor bajo para R permitiendo leer pocos nodos para asegurar la consistencia. Por otro lado, si R=1 y W=5, la escritura no puede ser alcanzada si se desconecta un nodo, en este caso puede pensarse en los valores R=2 y W=4, de esta forma se alcanza la escritura incluso si se desconecta uno de los nodos.

- ***Membresía:*** En una base de datos particionada donde podrían incorporarse o quitarse nodos al sistema en cualquier momento sin que impacte esta operación a todos los nodos, es necesario que los nodos se comuniquen entre sí especialmente ante cambios en la membresía.

  Cuando un nuevo nodo se incorpora al sistema las siguientes acciones suceden:

  1. El nuevo nodo incorporado anuncia su presencia y su identificador a los nodos adyacentes o a todos los nodos a través de un mensaje broadcast.
  2. Los nodos vecinos al nodo incorporado ajustan sus objetos y replicas.
  3. El nodo incorporado copia los set de datos de los cuales ahora es responsable de sus vecinos. Esto puede ser realizado en forma asincrónica.
  4. Si en el paso 1, el cambio de membresía no ha sido notificado por broadcast, el nodo incorporado ahora anuncia su presencia a todos los nodos.

  En la siguiente figura se ilustra las acciones cuando se incorpora un nuevo nodo:

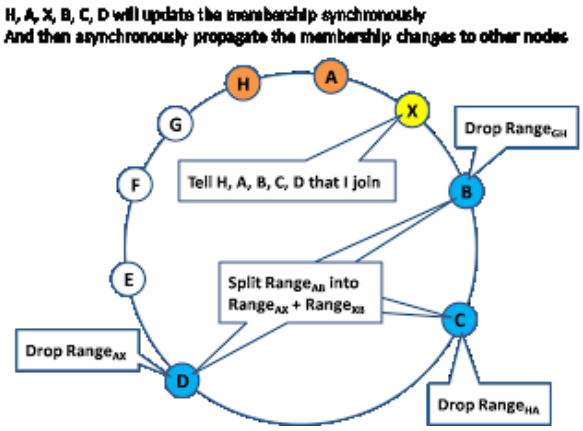

Ilustración 34: Cambios de membresía – Incorporación de nodo

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

En la figura 34 se ilustra un ejemplo de incorporación de un nuevo nodo, en este caso el nodo X se incorpora al sistema y existe un factor de replicación (r) igual a 3. El nodo es incluido entre los nodos A y B a través de la función de hash, por lo que los nodos H, A y B transfieren datos al nuevo nodo X y luego los nodos B, C y D pueden eliminar parte de sus datos para los cuales el nodo X es ahora responsable como tercer replica (junto a los nodos H, A y B).

Durante la transferencia de datos y ajuste de rangos, los nodos adyacentes al nuevo nodo podrían aun ser consultados y estos reenviar las consultas al nuevo nodo; por otro lado, el nuevo nodo podría estar en proceso de descarga de datos y no encontrarse aún listo para operar, por lo que podría utilizarse vector clock para determinar si un nuevo nodo incorporado está listo para atender la petición y si no lo está, el cliente puede contactar otra replica.

Cuando un nuevo nodo abandona el sistema las siguientes acciones suceden:

1. Los nodos en el sistema necesitan detectar si el nodo ha abandonado, debido a que podría haber fallado y no notificado de su partida. Es común en muchos sistemas que no se notifique sobre la partida de un nodo. Si los nodos del sistema se comunican regularmente por ejemplo a través del protocolo Gossip, éstos podrán detectar la partida de un nodo debido a que ya no responde.
2. Si la partida de un nodo ha sido detectada, los nodos vecinos deben reaccionar intercambiando datos entre ellos y ajustando sus objetos y replicas.

En la siguiente figura se ilustra las acciones durante la partida de un nodo:

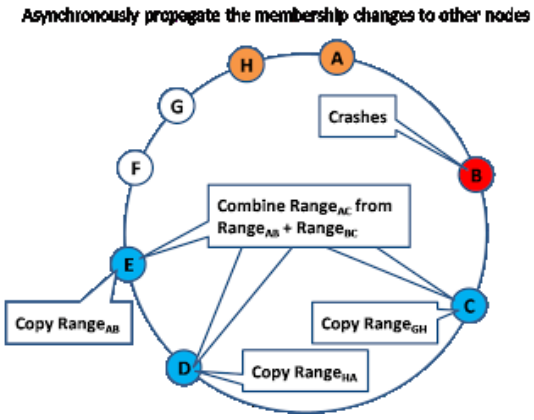

Ilustración 35: Cambios de membresía – Partida de nodo

Ilustración de http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

En la figura 35 se ilustra un ejemplo de partida de un nuevo nodo, en este caso el nodo B abandona el sistema debido a un fallo. Los nodos C, D y E se vuelven responsables de los nuevos intervalos de los objetos de hash y por lo tanto deben copiar datos de los nodos en sentido horario y reorganizar su representación interna de los intervalos ya que los rangos AB y BC ahora se han fusionado en AC [45].

4.2.3 Alta Disponibilidad

Como se mencionó, la alta disponibilidad no solo implica realizar replicación de datos (cuyas técnicas fueron exploradas en las secciones anteriores), sino además contar con mecanismos de detección y recuperación ante fallas eventuales o permanentes.

A continuación se citan algunas técnicas actualmente empleadas por Dynamo:

- ***Sloopy Quorum:*** es la aplicación de la técnica de quorum o votación pero no de forma estricta sino en forma desordenada/desprolija (sloopy). Esta variación implica que en una ejecución de operaciones de lectura o escritura, los primeros N nodos activos o "saludables" de una lista de preferencias para un ítem de datos son tomados en cuenta. Estos nodos no necesariamente son los primeros N en sentido horario alrededor del anillo de Consistent Hashing [12].

- ***Hinted Handoff:*** implica mantener un registro sobre los nodos consultados y que no respondieron. Las notificaciones se realizan a través de hints (indicios) que se incluyen en la metadata [12].

  A continuación se presenta un ejemplo de la aplicación de Hinted HandOff, supongamos la siguiente configuración en anillo:

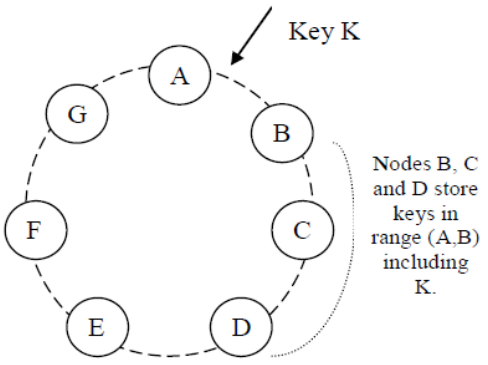

Ilustración 36: Hinted HandOff - Ejemplo

Ilustración de http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html

En esta configuración el factor de replicación r = 3. Si el nodo A falla y se vuelve inaccesible durante una operación de escritura, entonces una réplica de datos que en funcionamiento normal hubiera residido en A, ahora será enviada al nodo D. Esto es realizado para mantener los niveles de disponibilidad y garantía

de durabilidad deseados. La réplica de datos enviada a D tendrá un indicio (hint) en su metadata que sugiere que nodo era el receptor inicial de la réplica (en este caso A). Los nodos que reciben hints, los mantendrán en una base de datos local separada que será escaneada periódicamente. Cuando se detecte que se ha recuperado el nodo A, D intentará entregar la réplica de datos a A. Una vez que la transferencia sea exitosa, D podrá borrar el objeto de su almacenamiento local sin decrementar el número total de réplicas en el sistema.
El uso de Hinted Handoff asegura que las operaciones de lectura y escritura no fallen debido a que un nodo no se encuentra temporalmente disponible o haya fallas en la red. Las aplicaciones que necesiten el mayor nivel de disponibilidad pueden establecer el parámetro W=1, lo cual asegura que se aceptará siempre una operación de escritura, tan solo se requiere que un nodo escriba la clave en su almacenamiento local y por lo tanto, solo se rechazará si todos los nodos del sistema no se encuentran disponibles [12].

- ***AntiEntropia a través de árboles Merkle:*** La técnica de Hinted Handoff funciona bien si el nivel de cambios de membresía (incorporación y partida de nodos) es bajo y las fallas en los nodos son transitorias. Pero hay escenarios en los cuales las réplicas de datos podrían volverse inaccesibles antes de ser devueltas al nodo replica original. Para manejar este tipo de inconsistencias se puede utilizar un protocolo de AntiEntropia que mantenga las réplicas sincronizadas, como por ejemplo los arboles Merkle [12].

  Definiremos primero los conceptos de AntiEntropia y arboles Merkle:

  Antientropia es un tipo de protocolo de Gossip (rumor) donde se esparce la información hasta que ésta se vuelva obsoleta por la incorporación de nueva información [50].

  Arbol Merke es un tipo de árbol de hash donde las hojas son los resultados de una función de hash de los valores de claves individuales. Los nodos padres en los niveles superiores son los resultados de una función de hash de sus respectivos hijos.

  La principal ventaja de los arboles Merkle es que cada rama del árbol puede ser chequeada independientemente sin requerir que los nodos descarguen todo el árbol o todo el set de datos. Esto permite realizar chequeos eficientes de inconsistencias, ya que dos nodos determinan diferencias por comparación jerárquica de los valores de hash de sus árboles merkle: primero se examinan las raíces de los dos árboles, si son iguales entonces los valores de las hojas en el árbol son iguales y no se requiere sincronización de los nodos. Si las raíces no son iguales, los valores de algunas replicas difieren, en ese caso se inspecciona sus nodos hijos y así sucesivamente. Para los casos de inconsistencias, los nodos

intercambian los valores de hash de los hijos y el proceso continua hasta que se llega a las hojas. Los arboles Merkle minimizan la cantidad de datos a ser transferidos para sincronización y reducen el número de lecturas de disco realizadas durante el proceso de Antientropia [12].

5. Conclusiones

Cada uno de los teoremas, técnicas, mecanismos, razonamientos y líneas de pensamiento expuestos en este trabajo han sido ampliamente discutidos en el ámbito de investigación y existen muchos puntos de vistas y opiniones sobre si los almacenamientos NoSQL traen ventajas reales o son solo iniciativas o modas impuestas por empresas vanguardistas. Pero en cualquiera de los casos, es fundamental no perder de vista el problema real: los cambios son continuos debido a que las necesidades son infinitas y por lo tanto, el crecimiento de la cantidad de información y la aparición de nuevos nichos será una realidad a la cual las aplicaciones deben adaptarse.

A continuación se resume algunas de las conclusiones:

- Los almacenamientos NoSQL se construyen sobre la base de alcanzar alta disponibilidad, procesamiento de grandes cantidades de datos y escalabilidad a través de la distribución sobre hardware económico (commodity).
- Las bases de datos relacionales nunca sufrieron una total transformación o rediseño y esto se refleja en limitaciones o flaquezas del modelo a las nuevas necesidades.
- Algunas de las técnicas que aplican las bases de datos relacionales, como sharding, son adaptaciones para alcanzar requerimientos que no se hallaban presentes al inicio del diseño.
- La implementación de esquemas distribuidos en bases de datos relacionales a través de protocolos de commit de 2 fases o Paxos no son sencillas.
- Si bien falta madurez en algunos de los almacenamientos NoSQL, existen varios antecedentes de casos de éxito: BigTable (Google), Dynamo (Amazon), Cassandra (Facebook), Voldemort (Linkedin), Riak (Mozilla), etc.

Finalmente indicaremos algunas líneas de investigación futura:

- Como se mencionó, existen varios casos de éxito de almacenamientos NoSQL, pero tambien como se indicaba en el paper “Debunking the NoSQL Hype” [14], esto lo consiguieron empresas que cuentan con los recursos para contratar a los mejor ingenieros. Es necesario madurar las técnicas para que puedan ser implementadas sin necesidad de invertir grandes cantidades de dinero de implementación y mantenimiento.

6. Bibliografía

[1] “NoSQL Relational Database Management System”
http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page

[2] NoSQL: What's in a name?
http://blog.sym-link.com/2009/10/30/nosql_whats_in_a_name.html

[3] Strauch, Christof (2010). NoSQL Databases.Universidad Stuttgart Media.
http://www.christof-strauch.de/nosqldbs.pdf - Pags. 3-17.

[4] Stonebraker, Michael (2009); “The “NoSQL” Discussion has Nothing to Do With SQL”. Communications of the ACM .
http://cacm.acm.org/blogs/blog-cacm/50678-the-nosql-discussion-has-nothing-to-do-with-sql/fulltext.

[5] Monash, Curt (2011). Defining NoSQL.
http://www.dbms2.com/2011/10/02/defining-nosql/

[6] Stonebraker, Michael; Hachem, Nabil; Helland, Pat; et.al. (2007). The End of an Architectural Era (It’s time for a complete rewrite). Comunications of the ACM.
ACM 978-1-59593-649-3/07/09.

[7] Tanenbaum, Andrew; Van Steen, Maarten (2006). Distributed Systems.Principles and Paradigms - Second Edition.
http://cs.boisestate.edu/~amit/teaching/455/slides/chap-07v2.pdf

[8] Definición “Disponibilidad”. Diccionario de la lengua española - Vigésima segunda edición.
http://lema.rae.es/drae/?val=disponibilidad

[9] Brewer, Eric (2000). Toward Robust Distributed Systems.
http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[10] Escalabilidad.
http://es.wikipedia.org/wiki/Escalabilidad

[11] Tolerancia a Fallos.
http://es.wikipedia.org/wiki/Tolerancia_a_fallos

[12] DeCandia, Giuseppe; Hastorun, Deniz; Jampani, Madan; et.al. (2007). Dynamo: Amazon’s Highly Available Key-valueStore. Comunications of the ACM.

[13] Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; et.al (2006). Bigtable: A Distributed Storage System for Structured Data.
http://research.google.com/archive/bigtable-osdi06.pdf

[14] Downing, Alan (2011). Debunking the NoSQL Hype. Oracle White Paper.

[15] Metz, Cade (2011). Oracle Defies Self With 'NoSQL' Database
http://www.wired.com/wiredenterprise/2011/10/oracle-nosql-database/

[16] Izrailevsky, Yury (2011). NoSQL at Netflix
http://techblog.netflix.com/2011/01/nosql-at-netflix.html

[17] Meijer, Erik; Bierman, Gavin (2011). A co-Relational Model of Data for Large Shared Data Banks.
thttp://queue.acm.org/detail.cfm?id=1961297

[18] Hoff, Todd (2010). Troubles with Sharding - What can we learn from the Foursquare Incident?.
http://highscalability.com/blog/2010/10/15/troubles-with-sharding-what-can-we-learn-from-the-foursquare.html

[19] Stonebraker, Michael (2010). Clarifications on the CAP Theorem and Data-Related Errors.
http://voltdb.com/clarifications-cap-theorem-and-data-related-errors/

[20] Wada, Hiroshi; Fekete, Alan; Zhao, Liang; Lee, Kevin; Liu, Anna (2011).
Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective.
http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper15.pdf

[21] Vogel, Werner (2008). Eventually Consistent – Revisited
http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

[22] Kyte, Thomas (2009). This should be fun to watch...
http://tkyte.blogspot.com.ar/2009/01/this-should-be-fun-to-watch.html

[23] Travis, Jon (SpringSource). Mencionado por Lai, Eric (2009). No to SQL? Anti-database movement gains steam
http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_movement_gains_steam_?taxonomyId=173&pageNumber=1

[24] Loukides, Mike (2012). The NoSQL movement. How to think about choosing a database.
http://radar.oreilly.com/2012/02/nosql-non-relational-database.html

[25] Avinash, Lakshman; Prashant, Malik (2009). Cassandra - Structured Storage System over a P2P Network
http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf

[26] Soltero, Javier (SpringSource). Mencionado por Lai, Eric (2009). No to SQL? Anti-database movement gains steam
http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_movement_gains_steam_?taxonomyId=173&pageNumber=1

[27] Forbes, Dennis (2010).Getting Real About NoSQL and the SQL-Isn't-Scalable Lie.
http://dennisforbes.ca/index.php/2010/03/02/getting-real-about-nosql-and-the-sql-isnt-scalable-lie/

[28] Shalom, Nati (2009). No to SQL? Anti-database movement gains steam – My Take.
http://natishalom.typepad.com/nati_shaloms_blog/2009/07/no-to-sql-anti-database-movement-gains-steam-my-take.html

[29] Rotem-Gal-Oz, Arnon (2006). Fallacies of Distributed Computing Explained.
http://www.rgoarchitects.com/files/fallacies.pdf

[30] Katsov, Ilya (2012). NoSQL Data Modeling Techniques
http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/

[31] North, Ken (2009). Databases in the Cloud: Elysian Fields or Briar Patch?
http://www.drdobbs.com/database/databases-in-the-cloud-elysian-fields-or/218900502?pgno=3

[32] Graph Database.
http://en.wikipedia.org/wiki/Graph_database

[33] Edlich, Stefan. NOSQL - Your Ultimate Guide to the Non-Relational Universe!
http://nosql-database.org/

[34] Gilbert, Seth; Lynch, Nancy (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services
http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf

[35] Burguess, Mark. Deconstructing the `CAP theorem' for CM and DevOps
http://markburgess.org/blog_cap.html

[36] Abadi, Daniel (2012).Consistency Tradeoffs in Modern Distributed Database System Design

http://cs-www.cs.yale.edu/homes/dna/papers/abadi-pacelc.pdf

[37] Pritchett, Dan (2008). BASE: An Acid Alternative
http://queue.acm.org/detail.cfm?id=1394128

[38] Ippolito, Bob (2009). Drop ACID and think about data
http://blip.tv/pycon-us-videos-2009-2010-2011/drop-acid-and-think-about-data-1959086

[39] Lipcon, Todd (2009). Design Patterns for Distributed Non-Relational Databases.
http://es.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases

[40] Multiversion concurrency control
http://en.wikipedia.org/wiki/Multiversion_concurrency_control

[41] Vector clock
http://en.wikipedia.org/wiki/Vector_clock

[42] Gossip protocol
http://en.wikipedia.org/wiki/Gossip_protocol

[43] Memcached
http://es.wikipedia.org/wiki/Memcached

[44] Computer cluster
http://en.wikipedia.org/wiki/Cluster_(computing)

[45] Ho, Ricky (2009). NOSQL Patterns.
http://horicky.blogspot.com.ar/2009/11/nosql-patterns.html

[46] Stoica, Ion; Morris, Robert; Karger, David; et.al. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Application
http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

[47] White, Tom (2007). Consistent Hashing.
https://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

[48] Jay, Kreps; et. al (2010). Project Voldemort - A distributed database
http://www.project-voldemort.com/voldemort/design.html

[49] Koren, Israel; Krishna, Mani C. – Universidad de Massachusetts – Departamento de Ingeniería Eléctrica y Computación (2006). Fault Tolerant Computing

[50] Robbert van Renesse; Dumitriu, Dan; et.al. Efficient Reconciliation and Flow Control for Anti-Entropy Protocols
http://www.cs.cornell.edu/home/rvr/papers/flowgossip.pdf