Universidad de Granada

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

Modelos Avanzados de Bases de Datos

Bigtable: A Distributed Storage System for structured data

Autores: Alexander Moreno Borrego
Carlos Jesús Fernández Basso

Correos: alexmobo@correo.ugr.es
karloos@correo.ugr.es

DNI: 39906263-K
75927137-C

Profesor: Carlos Cano Gutiérrez

Contenido

Introducción

BigTable es una base de datos distribuida de almacenamiento clave-valor, desarrollada por Google, para satisfacer la necesidad a

Terminología y conceptos básicos

- Lenguaje *Sawzall*: Lenguaje desarrollado por *Google* especialmente diseñado para el procesamiento de datos.
- *MapReduce*: Marco de trabajo desarrollado por Google para ejecutar grandes cálculos computacionales paralelos.
- *Google File System* (*GFS*): Sistema de archivos distribuido y escalable para aplicaciones con un intenso uso de datos y altamente distribuidas.[1]
- *Chubby Lock Service*: Servicio para garantizar y proporcionar el acceso concurrente a recursos compartidos. Destinado a utilizarse en un sistema distribuido de acoplamiento flexible construido con un número elevado de pequeños ordenadores conectados por una red de alta velocidad.[2]
- *SSTable* (*Sorted Strings Table*): Formato de archivo que consta de parejas de *strings* clave-valor, ordenadas por la clave.

[1] *Scalable distributed file system for large distributed data-intensive applications*. Definición extraída y traducida del resumen del artículo del trabajo (Puede encontrarse más abajo en la bibliografía).
[2] *Chubby is a lock service intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network.* Definición extraída y trducida de la introducción del artículo del trabajo (Puede encontrarse más abajo en la bibliografía).

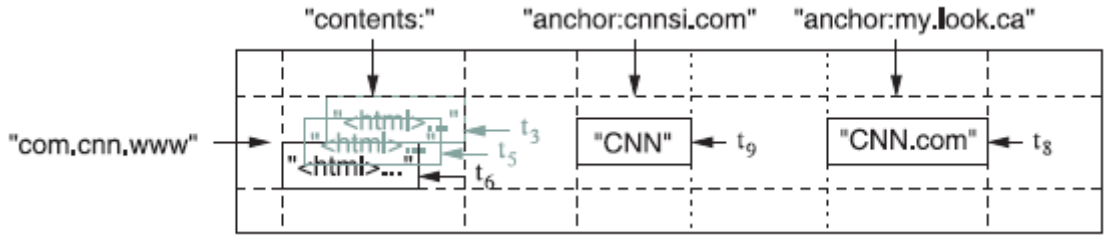
Descripción

Modelo de datos

BigTable se organiza en *clusters*. Un *cluster* es una serie de procesos que proporcionan un conjunto de *tablets*. A su vez, una tablet es un mapa ordenado, multidimensional, persistente, distribuido y disperso. Se organiza en tres dimensiones:

- Filas o *rows* (string)
- Columnas o *columns* (string)
- Marcas de tiempo o *timestamps* (entero de 64 bits)

Estas tres dimensiones referencian una celda o *cell*. Las filas se agrupan para formar la unidad de balanceo de carga y las columnas se agrupan para formar la unidad de control de acceso y conteo de recursos.



Rows

Se mantienen las filas ordenadas lexicográficamente por su clave (usualmente 10-100 bytes). La fila es la unidad de coherencia transaccional en *BigTable*, no soporta transacciones a través de múltiples filas.

Las filas con claves consecutivas son agrupadas en *tablets*. Como resultado es esta agrupación, el acceso a un rango pequeño de claves es eficiente e involucra a un pequeño número de máquinas. Esta propiedad puede explotarse muy bien ya que cada uno puede elegir sus claves libremente.

Columns

Las claves de las columnas se agrupan en *column families*. Los datos de las columnas de una misma familia normalmente son del mismo tipo y tienen que ser declaradas antes de poder introducir datos dentro de las columnas que conforman la familia.

Cambiando el esquema de la tabla se pueden borrar familias enteras, y con ello los datos de las columnas que la forman. Hasta que *BigTable* no soporte las transacciones a través de múltiples filas, la información de una columna que resida en más de una fila no podrá se borrada atómicamente.

La sintaxis para las claves de las columnas es la siguiente: *family:qualifier*.

El control de acceso y los conteos de memoria y disco se realizan en este nivel familia-columna.

Timestamps

Diferentes celdas pueden contener múltiples versiones de los mismos datos indexados por un *timestamp*. Pueden ser asignados automáticamente por *BigTable* (en microsegundos) o hacerlo el usuario. Se ordenan en orden descendente, es decir, datos más nuevos primero.

BigTable proporciona un *Garbage Collector* de versiones, se puede configurar en dos modos:

1. Mantener sólo las *n* versiones más nuevas.
2. Mantener sólo las versiones en un rango de tiempo (por ejemplo, las versiones de los últimos 7 días).

API

BigTable cuenta con un API pública (escrita en C++) que proporciona funciones para crear y eliminar *clusters*, tablas y familias de columnas, para cambiar los metadatos de *cluster*, tablas y familias de columnas, y también para cambios de privilegios de acceso.

También se puede escribir, consultar y borrar valores de filas individuales o iterar sobre un conjunto de ellas. Estas son las funciones más sencillas que proporciona el API de *BigTable*.

BigTable funciona sobre *Linux*.

Una cuestión muy importante es que proporciona una *interface* al cliente que le permite simular las transacciones sobre varias filas. Además permite a los usuarios ejecutar *scripts* en el lenguaje *Sawzall*.

También puede ser utilizado con *MapReduce*.

Infraestructura

BigTable se apoya en muchas otras piezas desarrolladas por Google. No usa *clusters* dedicados sino que puede correr con otros procesos a su lado.

BigTable usa GFS (*Google File System*) para almacenar los datos y el formato *SSTable* (*Sorted Strings Table*) para sus ficheros de almacenamiento. Este formato contiene una secuencia de bloques (64 KB por defecto) indexados al final del archivo. Cada vez que se abre un fichero *SSTable* se copia el índice en memoria para optimizar los accesos al disco. Opcionalmente, permite copiar todo el fichero *SSTable* en memoria para no tocar el disco.

También utiliza *Chubby Lock Service* para sincronizar el acceso a datos compartidos, entre otras cosas. *Chubby* utiliza 5 réplicas de cada archivo (una como *master*) para mantener la alta disponibilidad de los archivos y mantiene la consistencia entre ellos mediante el algoritmo de *Paxos*. Proporciona directorios y archivos que se modifican de forma atómica. También mantiene sesiones entre los clientes.

Implementación

La implementación de *BigTable* tiene tres grandes componentes:

- Una librería lincada a cada cliente
- Un servidor maestro
- Un conjunto de servidores de *tablets* (pueden añadirse y eliminarse dinámicamente según las exigencias del trabajo de las *tablets*)

El servidor maestro se encarga de asignar las *tablets* a los servidores de *tablets*, detectar la adición o supresión de los servidores dinámicamente, el balanceo de carga y el *garbage collecting* de archivos en GFS. También se encarga de los cambios en el *schema* sufridos por la creación y eliminación de *tables* y *column families*.

Cada uno de los servidores de *tablets* dentro del conjunto se encarga de las peticiones de lectura y escritura de un conjunto determinado de *tablets*, y de separar *tablets* demasiado grandes.

Los clientes se comunican directamente con los *tablet servers*, lo que evita que el flujo de datos pase a través del *master server*.

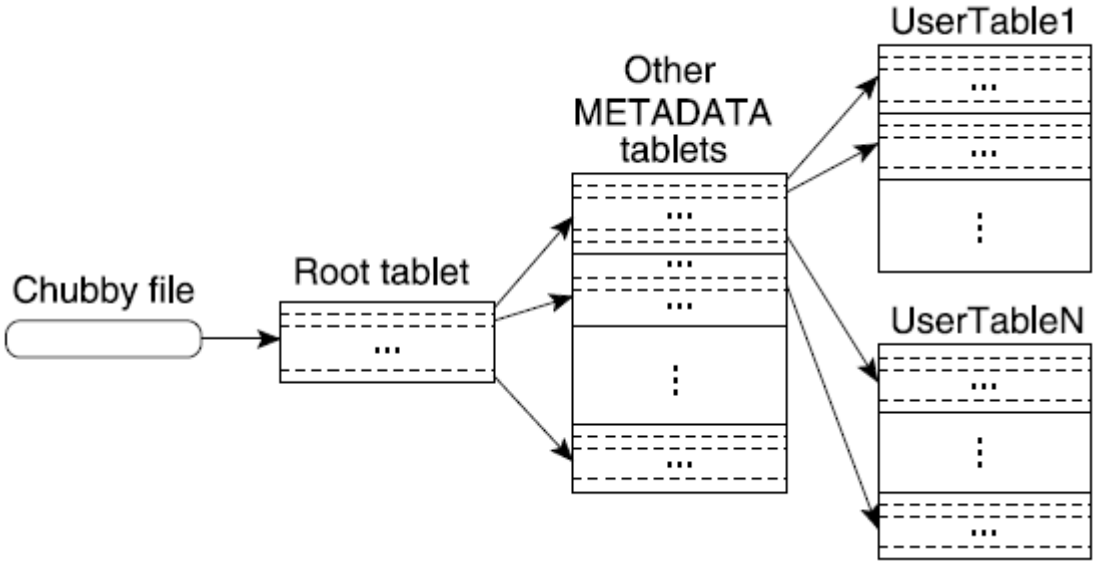
Un *cluster* almacena varias tablas y cada tabla almacena varias *tablets* (inicialmente solo una). A medida que esta *tablet* va creciendo en tamaño, se va dividiendo en *tablets* más pequeñas (aproximadamente de 1GB). No se recomienda que las *tablets* sean extremadamente grandes porque *BigTable* no lo soporta (unos pocos cientos de GB es el valor recomendado).

Localización de *Tablets*

Se usa una jerarquía de tres niveles análoga a un árbol B+.

- *Chubby File*: Archivo almacenado en *Chubby* que contiene la localización de la tableta raíz (*Root Tablet*).
- *Root Tablet*: Contiene las localizaciones de todas las *tablets* de una tabla de meta-datos especial. Esta *tablet* nunca se divide.
- *METADATA table*: Almacena la localización de una *tablet* bajo una clave de fila que codifica el identificador de tabla de la Tablet y su fila final. Cada fila de esta tabla almacena 1KB de datos aproximadamente, con un máximo de 128MB.

  También guarda un log de todos los eventos pertenecientes a cada *tablet*, información que ayuda a las tareas de *debug* y análisis de rendimiento.



La librería lincada al cliente atraviesa toda esta jerarquía para localizar las *tablets* y guarda la localización que encuentra. Si la información de la localización es incorrecta retrocede en la jerarquía recursivamente.

Si la *cache* del cliente esta vacía el algoritmo de localización requiere de tres accesos de ida y vuelta sobre la jerarquía, incluyendo una lectura de Chubby.

Si la *cache* del cliente es antigua el algoritmo de localización requiere de seis accesos de ida y vuelta sobre la jerarquía por que las entradas antiguas solo se descubren mediante fallos.

Las localizaciones se guardan en memoria, es decir, se reduce el acceso a GFS.

Asignación de *Tablets*

Cada *tablet* solo se asigna a un servidor de *tablets* y el servidor *master* es el que se encarga de ello. Para asignar tablets el *master server* envia una petición de carga de *tablet* (*tablet load request*). Un *tablet server* solo accepta *tablets* del *master server* actual.

BigTable usa *Chubby* para monitorear los *tablet servers*. Cuando se inicia un *tablet server* se le asigna una llave exclusiva en archivo único en un directorio especifico en *Chubby*. El *master server* monitorea este directorio para vigilar a los *tablet servers*. Un *tablet server* deja de servir *tablets* cuando pierde la llave exclusiva.

El *master server* es el encargado de repartir las *tablets* que puedan quedar colgadas por una caida o paro de un *tablet server*. Periodicamente pregunta por el estatus de los *tablet severs*. Cuando el *master server* no puede contactar con un *tablet server* lo da por perdido y elimina el fichero del servidor en *Chubby*. Una vez eliminado el fichero el *master server* ya puede mover las *tablets* a otros servidores.

Cuando el master server se inicia realiza los siguientes pasos:

1. Adquiere un cerrojo en *Chubby* permitiendo solo una instancia del *master server*.
2. Escanea el directorio de *Chubby* en busca de *tablet servers* vivos.
3. Se ponen en contacto con cada servidor para saber que *tablets* estan asignadas y para dar a conocer que él es el *master server*.
4. Explora la tabla de meta-datos para saber que tablets no estan asignadas.

El escaneo de la tabla de meta-datos no puede llevarse a cabo antes de que las *tablets* de meta-datos no se hayan asignado. Por lo tanto, se asegura de asignar la r*oot tablet* por que es ésta la que contiene los nombres de todas las *tablets* de meta-datos.

El conjunto de *tablets* solo cambia cuando se unen o se dividen *tablets*.

Servicio de *Tablets*

El estado de una *tablet* se guarda en *GFS*. Las actualizciones se guardan en un *log* que marca “puntos de restauración”. Los cambios recientes en las *tablets* se guardan en memoria (*memtable*) mientras que los mas antiguos en ficheros *SSTable* en disco. Para reconstruir una *tablet* solo hay que mirar el último punto de restauración y realizar todas las operaciones que se han realizado desde entonces.

Cuando alguien quiere escribir en o leer de una *tablet* primero se comprueba que esta sea correcta y se comprueban los permisos de quien quiere modificar esa

tablet en *Chubby*. Todos los cambios se irán almacenando por si hubiera que reconstruir esa *tablet*.

Compresión

Cuando el tamaño de una *tablet* alcanza un umbral determinado su *memtable* se congela y se crea otra nueva. La *memtable* congelada se escribe en disco como un archivo *SSTable* en *GFS* y se crea un "punto de restauración" en el *log*. Esta operación llamada compresión menor (minor compaction) tiene dos finalidades:

1. Reducir el uso de memoria
2. Reducir la cantidad de datos que hay que leer en la posible recuperación de la *tablet*

Cada cierto periodo de tiempo, un proceso en *background* realiza la unión de todas las *SSTables* y la *memtable* referentes a una *tablet*. La *SSTable* resultante de esa compresión descarta todas las anteriores. Esta operación recibe el nombre de compresión mayor (*major compaction*).

Cuando una maquina intenta escribir en un archivo de *GFS*, éste intenta crear una réplica en el ordenador que está intentando escribir. Por otro lado, estas compresiones se realizan en todas las réplicas que *GFS* tiene almacenadas para un mismo archivo. *BigTable* se beneficia de esto porque en las lecturas intenta leer de la réplica que el lector tiene más cerca creando un acceso más rápido a los datos.

Manejo del *schema*

El *schema* se almacena en *Chubby*. Éste es un aliado perfecto de *BigTable* porque proporciona escrituras atómicas y una *cache* consistente de archivos pequeños.

Refinamientos posteriores

- *Locality groups*: Se pueden separar las familias de columnas para proporcionar un acceso más restringido, puede haber columnas que se consulten más frecuentemente que otras. Se pueden cargar totalmente en memoria para su uso.
- Compresión: Los cliente pueden escoger entre comprimir o no un *locality group*. Los algoritmos escogidos ponen por delante la rápida descompresión a una gran compresión de los datos.
- *Caching* para más rendimiento en la lectura: Hay dos niveles de cache:
    1. *Scan cache*. Se cachean los pares clave-valor de los índices de los archivos *SSTable*.

2. *Block cache*. Se cachean los bloques de *SSTable* enteros que se leen de GFS.

- *Bloom filters*: Se usan para reducir los accesos a disco cuando una *SSTable* no está en memoria. Nos permite saber si un *SSTable* contiene información para una fila/columna especificada.

- Implementación de *log*: Utilizar un log por tablet es muy costoso en términos de *GFS* y accesos a disco, por lo tanto, solo se utiliza un *log* por *tablet server*. Esto complica la recuperación de *tablets* en el caso de que un *tablet server* falle o caiga, es decir, si 100 máquinas quieren hacerse cargo de las *tablets* de ese servidor caído, se realizarán 100 accesos a disco.

  Para evitar esto, se ordenan las entradas del *log* para que todas las mutaciones de las mismas *tablets* estén juntas. Después se separa el *log* en ficheros de 64MB y se leen en paralelo. La lectura es dirigida por el *master server*.

  En la práctica, cada *tablet* server tiene 2 hebras de escritura en *logs* pero solo una es la que se utiliza, en caso de que la hebra activa se ralentice se desactiva la escritura por esa hebra y se activa la otra hebra en el otro *log*. Las entradas tienen un número de secuencia para facilitar la recuperación de los dos *logs*.

- Aumento de la velocidad de recuperación de *tablets*: La velocidad de recuperación se aumenta compactando la *tablet* antes de que el *tablet server* pare.

- Explotando la inmutabilidad: La única estructura mutable en *BigTable* es la *memtable*. Para reducir las disputas en las lecturas, cada fila se copia y se permiten las lecturas y escrituras en paralelo.

  El problema del borrado de filas en un mismo archivo se sustituye por el borrado de archivos obsoletos por el *garbage collector*. El *master* es el encargado de eliminarlas.

  Finalmente, esta inmutabilidad permite dividir las *tablets* rápidamente. En lugar de crear una *SSTable* para cada una de las *tablets* hijas se permite compartir la *SStable* del padre.

Ejemplos de utilización

- *Google Analytics* (analytics.google.com): Servicio que ayuda a los *webmasters* con estadísticas de tráfico de sus páginas web.

  Un ejemplo de tabla es la *raw click table*. En *raw click table* cada fila es una sesión de un usuario. El nombre de la fila es el nombre de la página web y el tiempo de inicio de la sesión. Este esquema se asegura de que las sesiones de las mismas páginas web esta juntas y ordenadas cronológicamente.

- *Google Earth* (maps.google.com o earth.google.com): Proporciona a los usuarios imagines por satélite en alta resolución de la superficie terrestre. Este sistema usa una tabla para el procesamiento de datos y un conjunto diferente de ellas para servir los datos.

  Cada imagen en la *imagery table* corresponde a un segmento geográfico. Las filas se nombran para que los segmentos geográficos adyacentes se encuentren cerca unas de otras. Contiene una *column family* para tener un seguimiento de las fuentes de los datos para cada segmento.

- *Personalized Search* (www.google.com/psearch): Servicio que archiva las consultas y los clics de los usuarios en el buscador web, el buscador de imágenes y las noticias.

  A cada usuario le corresponde una fila en la tabla y el nombre de cada fila es un identificador único que tienen todos los usuarios. Todas las acciones de los usuarios son almacenados en una tabla. Hay *column families* para cada tipo de acción y una para todas ellas.

Otros proyectos

HBase

HBase es una base de datos, *noSQL* y de código abierto, distribuida. Está basada en *BigTable*.

El modelo de datos es una réplica del de *BigTable*.

Puede ser accedido a través de *APIs* de diferentes lenguajes como *Java*, *Jython*, *Groovy*, *Scala* y *JRuby* (como lenguajes que utilizan la *JVM*), y *REST* y *Thrift* (como lenguajes que no utilizan la *JVM*). También puede utilizarse con *Hadoop MapReduce,* implementación *open-source* de *MapReduce,* para el procesamiento de datos en paralelo.

La infraestructura se apoya en Hadoop, que proporciona *HDFS* (*Hadoop Distributed File System*), como sistema de ficheros.

En la implementación de *HBase* hay que contar con caché de bloques y *Bloom filters* para la optimización de las consultas sobre grandes volúmenes de datos y *autosharding*.

Multiplataforma.

LevelDB

LevelDB es una base de datos, *noSQL* y de código abierto, de almacenamiento para sistemas embebidos que se utiliza como una librería. Es un almacenamiento basado en BigTable desarrollado por Google. Soporta multitud de sistemas operativos.

El modelo de datos es una réplica del de *BigTable*.

Se puede acceder y trabajar con los datos a través de un *API* escrita en C++.

La infraestructura en la que se apoya puede ser tan variada como el sistema operativo donde se instale el software (sistemas *UNIX*, *Mac OS X*, *Windows* y *Android*). Utiliza la librería *Snappy*, desarrollada por *Google*, para la compresión de los datos. Utiliza el formato *SSTable* para guardar los datos en los archivos.

En cuanto a implementación, es similar a la representación de *tablet* de *BigTable*. Cada base de datos se compone de un directorio con una serie de ficheros:

- ***Log.*** Guarda las actualizaciones recientes. Cuando se alcanza un tamaño en memoria se vuelca en un archivo *.sst. Se guarda en *memtable*.

- ***SSTable.*** Guardan los datos ordenados por clave.
- ***Manifest.*** Contiene información sobre que *SSTables* componen todos los niveles de datos con sus correspondientes rangos de claves y otra información importante. Se crea uno cada vez que se abre la base de datos.
- ***Current.*** Guarda cuál es el último *manifest* creado.

Por lo que respecta a la compresión se van compactando archivos por niveles cuando estos superan el límite de tamaño para ese nivel N, dando lugar a archivos de nivel N+1. Cada vez que se compactan archivos el *Garbage Collector* elimina los archivos *log* que no son el actual y, las tablas que no son referenciadas por el *manifest* y no son la salida de la actual compresión.

Hypertable

Hypertable es un proyecto, *noSQL* y *open-source,* basado en *BigTable*. Es una base de datos distribuida de alto rendimiento y altamente escalable.

El modelo de datos es una réplica del de *BigTable*.

Puede ser accedido a través de 2 *APIs* diferentes: *HQL* (*Hypertable Query Language*) y *Thrift API.* Puede ser utilizado también con *Hadoop MapReduce*. Además, se permite ejecutar *scripts* con los lenguajes *Hive* and *Pig*.

Su infraestructura se apoya sobre *Hadoop*, utilizando *HDFS* como sistema de ficheros (también puede utilizarse sobre otros *DFS* o incluso *FS*). Utiliza *Hyperspace* como servidor de alta disponibilidad de cerrojos.

En cuanto a su implementación es casi idéntica a la de *BigTable*, con la única diferencia. La diferencia está en que por encima del *DFS* se encuentra una capa que abstrae el sistema de ficheros para que puede utilizarse cualquiera de los siguientes: *HDFS*, *MapR*, *Ceph*, *KFS* o el sistema de ficheros local.

Cassandra

Cassandra es una base de datos, *noSQL* y *open-source*, distribuida, altamente escalable y eventualmente consistente. Basada en *BigTable* y *Dynamo* (de Amazon) publicada por Facebook en 2008.

El modelo de datos es una réplica del de *BigTable*.Implementa un modelo de replicación "sin puntos de falla" muy parecido al de *Dynamo*.

Hay muchas APIs de acceso para *Cassandra*, algunas de ellas son las siguientes: *Java* (*Hector*), *Python* (*Pycassa*), *PHP* (*PHPcassa*). Estas librerías son las mejores que hay hoy por hoy. También puede usarse, desde las últimas versiones de *Cassandra*, *Cassandra CLI* y *CQL* (*Cassandra Query Language*).

La infraestructura

Algunas empresas como Digg, Twiter, Rackspace vieron el potencial de Casandra y decidieron colaborar con el proyecto y participar en su desarrollo.

Multiplataforma

DynamoDB

Dynamo es una base de datos NoSQL propietaria y de código cerrado, por lo que su uso es exclusivo de la empresa que la implementó: Amazon.

En octubre de 2007 se publicó un *paper* con el diseño y la especificación, a grandes rasgos, de Dynamo, rompiendo con conceptos como la consistencia o modelo relacional. Su objetivo se definió claramente: escalabilidad y disponibilidad.

DynamoDB puede ser gestionada en unos pocos clics desde la consola de administración de AWS. Así si lo requerimos podemos iniciar una nueva tabla de DynamoDB, aumentar o disminuir los recursos, según las necesidades consultando las estadísticas de rendimiento.

La infraestructura es Dynamo (storage system). Todos los datos se almacenan discos de estado sólido (SSD) para asegurar que el acceso de datos sea más rápido.

DynamoDB integra Amazon Elastic MapReduce lo que permite realizar análisis complejos de grandes cantidades de información usando Hadoop sobre AWS.

Multiplataforma.

Un aspecto interesante de Dynamo es el garantizar, en el 99.9% de los accesos, un tiempo de respuesta menor de 300ms, que provee a los clientes una experiencia "siempre en línea", sin caídas ni restricciones del servicio.

MongoDB

Es un sistema de base de datos NoSQL orientado a documentos, desarrollado bajo el concepto de código abierto.

Para almacenar los documentos, MongoDB utiliza una serialización binaria de JSON, llamada BSON, que es una lista ordenada de elementos simples. El núcleo de la base de datos es capaz de interpretar su contenido, de modo que lo que a simple vista parece un contenido binario, realmente es un documento que contiene varios elementos. Estos datos están limitados a un tamaño máximo de 4 MB; para tamaños superiores se requiere del uso de GridFS.

Una de sus características reseñables es la del *Autosharding* mediante el cual una base de datos puede ser fraccionada y el programador la ve como una base de datos única, aumentando asi su facilidad de uso y su escalablidad.

Plataformas:

- Linux
- Windows
- Mac OS X

COUCHDB

CouchDB es un sistema de base de datos documental orientado a documentos, que se distribuye bajo licencia Open Source.CouchDB almacena los datos como documentos. CouchDB se diseñó con el objetivo de ser altamente escalable y a bajo costo, pudiendo usarse servidores de bajo coste.

Esta especialmente diseñado para la WEB dando facilidad de uso en este campo es mucho menos especializada que MongoDB y por lo tanto podemos tener mayor abanico de usos más amplio. Por ejemplo al poder tener el uso de transacciones podremos utilizarlo para aplicaciones contables.

Multiplataforma.

Además de estas comparaciones se adjunta un **anexo** con una tabla de comparaciones de estas tablas

Conclusiones

Aquí mostramos a nuestro parecer los mejores usos de las Bases de datos comparadas:

	BigTable	DynamoBD	LevelDB	CouchDB
Mejor Uso	diseñado para escalar a través de cientos o miles de máquinas	para soluciónes grande Data Base	Bases de datos Nosql no distribuidas	acumulación, en ocasiones cambiar datos con consultas predefinidas

	Hypertable	MongoDB	Cassandra	HBase
Mejor Uso	Mejor que Hbase	consultas dinámicas,con frecuencia escrituras, rara vez leen los datos estadísticos	escribir con frecuencia, leer menos	lectura aleatoria escribir en grandes bases de datos

Bibliografía

- Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. 2008. Bigtable: A distributed storage systemfor structured data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages. DOI = 10.1145/1365815.1365816. http://doi.acm.org/10.1145/1365815.1365816
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. SIGOPS Oper. Syst. Rev. 37, 5 (October 2003), 29-43. DOI=10.1145/1165389.945450 http://doi.acm.org/10.1145/1165389.945450
- Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, Berkeley, CA, USA, 335-350.
- Apache Software Foundation. The Apache HBase™ Reference Guide [en línea]. Disponible de World Wide Web: http://hbase.apache.org/0.94/book.html
- Google Inc: Jeffrey Dean, Sanjay Ghemawat. LevelDB [en línea]. Disponible de World Wide Web: https://code.google.com/p/leveldb/
- Hypertable, Inc. Documentation | Hypertable – Big data. Big performance [en línea]. Disponible de World Wide Web: http://hypertable.com/documentation/
- Apache Software Fundation. Cassandra Wiki [en línea]. Disponible de World Wide Web: http://wiki.apache.org/cassandra/

Otras referencias de apoyo

- Wikipedia, the free encyclopedia. Hbase [en línea]. Disponible de World Wide Web: http://en.wikipedia.org/wiki/Hbase
- Wikipedia, the free encyclopedia. LevelDB [en línea]. Disponible de World Wide Web: http://en.wikipedia.org/wiki/LevelDB
- Wikipedia, la enciclopedia libre. LevelDB [en línea]. Disponible de World Wide Web: http://es.wikipedia.org/wiki/LevelDB
- Wikipedia, the free encyclopedia. Hypertable [en línea]. Disponible de World Wide Web: http://en.wikipedia.org/wiki/Hypertable
- Wikipedia, the free encyclopedia. Apache Cassandra [en línea]. Disponible en World Wide Web: http://en.wikipedia.org/wiki/Apache_Cassandra