Universidad Zaragoza

Proyecto Fin de Carrera

Scalability of a cloud-based data store: Improving HBase performance

Autor

Mario Cerdán Lázaro

Director

Keijo Heljanko

Ponente

José Ángel Bañares Bañares

INGENIERÍA INFORMÁTICA

2014

Scalability of a cloud-based datastore: Improving HBase performance

Resumen

Cloud computing has opened doors to a new era of enterprises that harness the new Cloud enabled business. More and more novel applications are leveraging this paradigm every day, which translates to a never seen increase in the amount of stored data. This phenomenon is commonly known as Big Data; the presence of rapidly expanding high-volume data sets. Many of these applications bring new challenges to databases and therefore the scalability of the Cloud-based databases has become a top-research issue of the Cloud Computing infrastructure. As an alternative to the well-known relational databases, NoSQL databases have born to fit Big Data application requirements. Traditional relational databases as they are often implemented are not sufficient anymore for Internet scale distributed systems dealing with Big Data. Nevertheless, NoSQL have proved to be robust in Big Data applications.

The purpose of this project is to scaling out the data of a San Diego company that produces software for wireless multimedia, which is currently implemented on a MySQL cluster. In order to improve the performance of the computation, we propose a solution using Apache HBase, a NoSQL database.

This final project proposes implementations as well as comparison details of a number of computation techniques conducted in HBase along with different open-source distributed computing components such as Hadoop HDFS and MapReduce, and presents benchmarks of our developed solution.

Índice

1. Memoria

Este proyecto final de carrera ha sido realizado en Aalto University (Helsinki, Finlandia). Por esta razón, la mayor parte de la memoria está en inglés.

La estructura de la memoria se divide en dos partes:

-1) Resumen en castellano de la memoria técnica.

-2) La memoria más detallada y técnica desarrollada en inglés. Esta parte contiene todas las referencias bibliográficas y la lista de figuras usadas en el proyecto.

La primera parte describe, a modo de resumen, el proyecto final de carrera (PFC) y continua con una introducción al tema donde se desarrolla el PFC. La siguiente subsección explica el desarrollo del proyecto. Por último, se presentan los resultados y conclusiones obtenidas.

1.1 Introducción

La computación en la nube se puede definir como un nuevo estilo de computación que ha transformando una gran parte de la industria IT (Information Technology), permitiendo a las compañías crear aplicaciones ofrecidas como servicios en Internet sin tener que hacer un gran desembolso de capital. Años atrás se requería de un gran esfuerzo monetario para llevar a cabo nuevos servicios IT, y ahora gracias a la irrupción en el mercado de los llamados Cloud Providers, compañías que venden recursos de computación en la nube a un precio muy reducido, estos costes se han visto reducidos enormemente, lo que ha llevado a la aparición de millones de servicios IT que antes no eran económicamente viables.

La computación en la nube se ofrece en tres modelos diferentes de prestación de servicios:

- Software como Servicio: Facebook es un ejemplo auto-explicativo de este tipo de servicio. Compañías ofrecen sus servicios/aplicaciones en la nube. Para hacer uso de ellas basta con dirigirse a su página web.

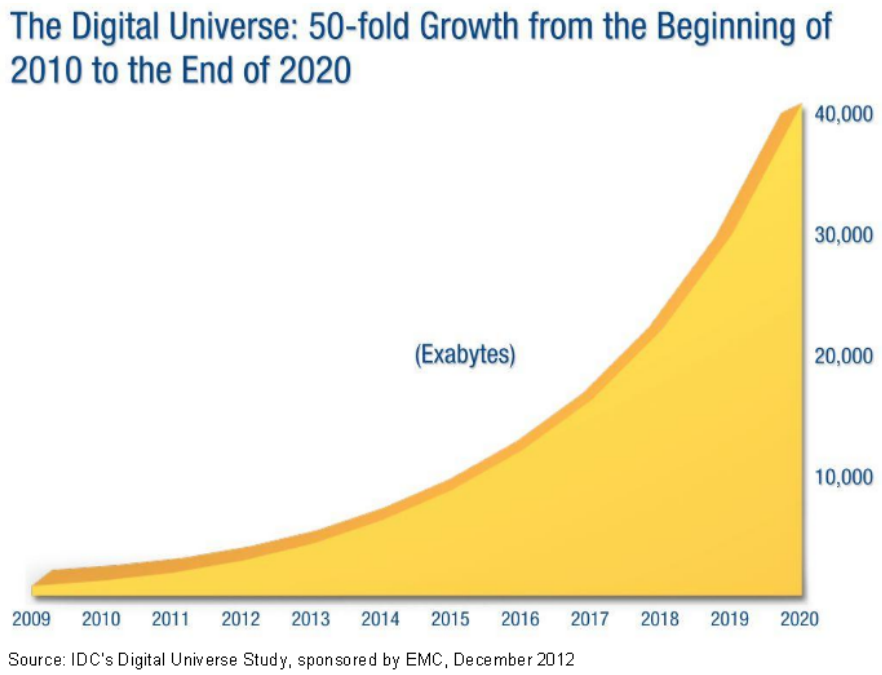
- Plataforma como Servicio: Google ofrece Google App Engine, una plataforma en la que el usuario puede desarrollar y ejecutar aplicaciones desarrolladas en diferentes lenguajes y tecnologías pero sin poder controlar la infraestructura en la que se ejecutan.

- Infraestructura como Servicio: Amazon Web Services ofrece infraestructura en la nube para el usuario final. EC2, ElasticCache o Amazon Dynamo (un ejemplo de base de datos) son ejemplos. En este modelo, el usuario controla todos los aspectos de su aplicación así como de la infraestructura en la que se ejecutan.

En el anexo se estudia y caracteriza el paradigma de Computación en la nube, así como los principales proveedores del mismo (Págs. 10-20)

1.1.1 Big Data

El rápido crecimiento del sector de computación en la nube ha traído consigo un nuevo problema. La cantidad de datos que se generan y consumen ha crecido y continúa creciendo exponencialmente año tras año. Este fenómeno es conocido como Big Data.



(Crecimiento de los datos generados de 2009 a 2020)

Este crecimiento ha hecho que los sistemas de manejo de información (Database Management Systems (DBMS)) se hayan convertido en una parte crítica y fundamental del llamado Big Data.

En el anexo se presenta como surgió el paradigma de Big Data y en qué situación se encuentra actualmente (Págs. 20-23).

1.1.2 Bases de datos NoSQL

Las bases de datos NoSQL (non-relational databases) han surgido a raíz del problema Big Data. Durante las últimas tres décadas, la amplia mayoría de empresas IT han usado y continúan usando bases de datos relacionales (RDBMS). Esta tecnología ha sido adoptada por muchas empresas para el almacenamiento estructurado de sus datos, pero con el impresionante incremento de los datos generados y consumidos (Big Data), algunas han visto como estos sistemas RDBMS (Relational Database Management System) empezaban a tener problemas de escalabilidad. Así pues y como solución a

estos problemas, las bases de datos NoSQL han empezado a aparecer. Estas bases de datos usan modelos de datos menos restrictivos que el relacional, a menudo a cambio de la perdida de alguna de las propiedades ACID.

Sus principales ventajas son:

- Simplicidad de diseño: No requieren estructuras de diseño fijas para los datos, lo cual permite una mayor flexibilidad a la hora de insertar los mismos.

- Escalabilidad horizontal: Añadiendo más nodos a la base de datos NoSQL se obtienen un mayor rendimiento del sistema. Este concepto es el opuesto a "escalabilidad vertical", en el cual se obtiene un mayor rendimiento al añadir más recursos a un nodo particular del sistema, como puede ser añadir más memoria RAM o más discos duros.

- Mayor disponibilidad: Las bases de datos NoSQL replican los datos que se insertan en ellas, lo cual implica que estos datos están disponibles en diferentes nodos. Así pues, la pérdida o desconexión de un nodo no implica la perdida de datos.

Sus desventajas son:

- A diferencia de las bases de datos relacionales, las bases de datos NoSQL no garantizan totalmente ACID: atomicidad, coherencia, aislamiento y durabilidad.

- Generalmente no soportan consultas complicadas como pueden ser las consultas JOIN.

Cada día más y más compañías que crean o hacen uso de grandes cantidades de datos como pueden ser Google, Amazon, o redes sociales como Twitter o Facebook, usan bases de datos NoSQL lo que les permite poder ofrecer datos en tiempo real perdiendo algo de coherencia, pues anteponen el rendimiento y disponibilidad de datos a no poder servir a sus usuarios lo que solicitan en el acto.

En el anexo se estudia en profundidad el concepto de bases de datos NoSQL, el teorema CAP y ACID (Págs. 23-29).

1.1.3 Nuestro desafío

El grupo de investigación de computación en la nube así como de sistemas distribuidos de la universidad de Aalto: "Distributed Systems" tiene como uno de sus objetivos el estudio y desarrollo de aplicaciones que hacen uso del paradigma Cloud Computing.

En nuestro caso, se presentó a este grupo el problema de escalabilidad que sufría una gran empresa de San Diego (EEUU) dedicada al almacenamiento de metadatos para una aplicación de búsqueda de videos en la Web. Hasta ese momento, podían manejar sin graves problemas la enorme cantidad de datos que sus usuarios creaban día a día, pero con la rápida expansión de la empresa, estaban empezando a sufrir grandes problemas de escalabilidad así como deficiencias en el manejo y transformación de esos datos. Este es un claro ejemplo del problema descrito arriba como Big Data. Así pues, para solucionarlo, en el grupo de Distributed Systems se planteó probar un enfoque distinto en cuanto al manejo y almacenamiento de todos los datos de la empresa, llegando a la conclusión de que se debían usar las llamadas tecnologías NoSQL o Cloud-based datastores.

Después del estudio de varias soluciones NoSQL como Cassandra, Riak o MongoDB se optó por cambiar todo el sistema de almacenamiento y tratamiento de datos de la empresa de MySQL a Apache HBase.

HBase es una base de datos NoSQL, distribuida y de código libre inspirada en Google BigTable y escrita en Java. Este proyecto se encuentra dirigido por la fundación Apache y es parte del proyecto Hadoop. Sus principales características son:

- Tolerancia a fallos: HBase mantiene los datos de forma redundante en varios nodos.

- Escalabilidad horizontal: Explicada anteriormente.

- Consistencia: A pesar de ser una base de datos NoSQL, mantiene todos los datos y sus réplicas consistentes.

- Usa Hadoop HDFS para el almacenamiento de datos, lo que permite el uso de algoritmos de procesamiento de datos como MapReduce (Framework desarrollado por Google que permite el procesamiento de millones de datos de una manera rápida y eficaz).

Los datos que almacena esta compañía no siguen un esquema fijo, lo que se traduce en que algunos datos tienen más campos que otros. HBase es perfecto para este tipo de situaciones, pues permite almacenar filas de datos con

diferente número de columnas en cada una de ellas o incluso diferentes tipos de datos.

En el anexo en inglés, se puede encontrar un estudio completo y detallado de HBase (Págs. 30-40).

Esta propuesta de cambio tecnológico trae consigo numerosos riesgos:

- Desconocimiento/Falta de experiencia de la tecnología por parte de la empresa.

- Cambio radical en el almacenamiento de los datos así como en la lectura de los mismos.

En este proyecto, para mitigar estos riesgos proponemos:

- El despliegue completo de la infraestructura.

- Desarrollos para hacer un uso eficiente de esta.

- Validación de la infraestructura.

1.2 Desarrollo del proyecto

Este proyecto final de carrera ha sido desarrollado en el departamento de Distributed Systems de la universidad de Aalto situada en Helsinki (Finlandia). En concreto, en el marco del programa de investigación "Data to Intelligence SHOK" financiado por Tekes, la agencia finesa de financiación para la tecnología y la innovación. El principal objetivo del mismo ha sido la evaluación de bases de datos Cloud-based como reemplazo de bases de datos tradicionales.

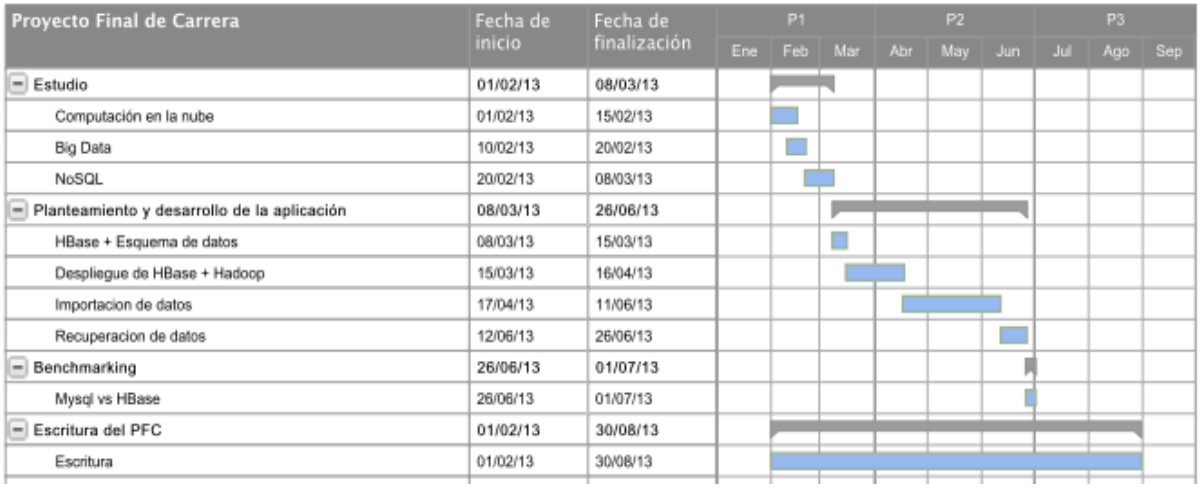
Dentro del grupo de investigación, mi posición era de "research assistant" bajo la dirección de Keijo Heljanko (Assoc. Professor). Teníamos una reunión semanal para comentar que había conseguido, que problemas y dificultades había encontrado y que tareas iba a desarrollar la siguiente semana.

José Ángel Bañares, profesor de la Universidad de Zaragoza, ha sido mi supervisor del proyecto en España. Por medio de correos electrónicos actualizaba a José Ángel Bañares con el estado del proyecto así como solucionaba diferentes dificultades que surgían a medida que avanzaba en el proyecto.

Mi trabajo estaba centrado en el estudio y reemplazamiento total del sistema de gestión de datos de la compañía descrita anteriormente. Comencé a trabajar en el proyecto en Febrero del 2013 y terminé a principios de Septiembre de 2013.

1.2.1 Tareas

A continuación, mostramos un diagrama de Gantt donde podemos ver como se han repartido las diferentes fases del proyecto a lo largo del tiempo.



1.2.2 Análisis de herramientas / Tecnologías

Cuando empecé a trabajar en mi proyecto final de carrera, mi primer objetivo fue conseguir documentación útil para hacerme con el contexto del trabajo así como entender correctamente todo lo relacionado con el tema. Una vez adquirida la idea general sobre Cloud Computing, estudiadas diferentes soluciones NoSQLs como Cassandra, Riak o Hbase, y paradigmas muy usados en conjunción con problemas de Big Data tales como MapReduce o Storm, trabajos y mejoras existentes en el campo que iba a desarrollar y más información relacionada con Big Data, decidimos junto con la empresa que la solución NoSQL que más se adaptaba a su equipo de trabajo así como a sus datos en general era HBase.

1.2.3 Análisis de datos / Información

Todos sus datos provienen de documentos XML con diferente número de elementos en cada uno de ellos. Un ejemplo conceptual de como lucen estos datos se puede encontrar en la siguiente figura:

```
<element 1>
 ...
</element 1>
<element 2>
        <sub-element 1>
                "Hi, I am a looooooong string"
        </sub-element 1>
        <sub-element 2>
                "Hi, I am a looooooong string version 2"
        </sub-element 2>
        ...
        <sub-element n>
                <sublist1>
                        ...
                </sublist1>
        </sub-element n>
</element 2>
 ...
<element N>
 ...
</element N>
```

Cada elemento que la compañía desea almacenar puede contener un número indeterminado de sub-elementos y estos a su vez pueden contener más sub-elementos.

El modo de almacenamiento que ofrece HBase encaja perfectamente con los datos de esta compañía. Cada fila de la base de datos estará formada por una clave principal que será el identificador único del elemento y cuatro columnas principales que a su vez contendrán un número variable de subcolumnas formadas por simples cadenas de caracteres de longitud indeterminada, números enteros o cualquier otro tipo de dato necesario.

De acuerdo con las especificaciones de la compañía, las consultas que más se repiten en el acceso de datos están basadas en el identificador de uno o varios elementos. Es por esto que se decidió usar el ID de cada elemento como la clave principal.

Una vez encontrada la base de datos NoSQL que mejor encajaba con sus necesidades y después del estudio de sus datos, se procedió a desplegar HBase en un clúster cedido por la universidad de Finlandia para la realización del proyecto.

1.2.4 Despliegue / Instalación

Para probar el diseño se montó un servidor HDFS/HBase versión CDH 0.92 sobre Triton desde cero. Triton es un clúster de computación de alto rendimiento dedicado a la computación científica en la universidad de Aalto. La implementación de HDFS/HBase sobre Triton se hizo usando 5 nodos: 1 nodo *HBase Master* + *Namenode* y 4 nodos *RegionServer* + *Datanode* (ver características de Triton y sus nodos en el anexo Págs. 44-45).

1.2.5 Tuning de Hadoop, HBase y JVM

Una vez desplegado el clúster HDFS/HBase, se procedió a la optimización del mismo, así como de Hadoop y de la máquina virtual de Java (JVM).

En esta etapa se estudiaron, modificaron y testearon toda serie de opciones de configuración de HBase/Hadoop para adaptar nuestro clúster a nuestra carga de trabajo (gran cantidad de escrituras por segundo). Todos los parámetros relacionados con las *MemStores* (memoria donde se almacenan los datos guardados en HBase antes de volcarlos a disco) se estudiaron y se modificaron para obtener un rendimiento superior al que se obtiene por defecto. Para ello se tuvo en cuenta las características de las que dispone cada nodo: cantidad de memoria RAM, numero de núcleos por nodo, etc. (Págs. 51-52 del anexo para más detalles).

También se estudiaron, modificaron y testearon los parámetros de la máquina virtual de Java (JVM) para adaptarlos a las características de los nodos de nuestro clúster y así obtener un rendimiento superior. Estos parámetros incluyen la cantidad de memoria RAM asignada a la máquina virtual de Java de cada nodo *(Java Heap* Space) y los recolectores de basura que ofrece la JVM (*ParNew Collector, ConcurrentMarkSweepGC, MSLAB*).

Para información más detallada acudir al anexo (Págs. 49-51).

1.2.6 Fase de Importación de datos

Una vez optimizado el clúster HDFS/HBase, se continuó con la importación de una gran parte de los datos de la empresa a nuestro nuevo clúster para simular el comportamiento que obtendría la empresa en sus servidores. Esta fase de importación tuvo varias etapas:

1º Etapa: Importación de los datos a través de una aplicación implementada íntegramente para el experimento usando la API escrita en Java de HBase. Los resultados se muestran a continuación:

Número de elementos importados	12186983
Elementos por segundo	1508

Esta etapa presenta numerosas deficiencias (descritas íntegramente en el anexo):

- Al comienzo de la fase de importación, la API de HBase trabaja con un solo *RegionServer*, mientras el resto de nodos se encuentran totalmente desocupados.

- El número de threads esperando conexiones en HBase es muy bajo por defecto (*hbase.region.handler.count* = 10). Además nuestra aplicación no aprovecha este parámetro pues solo crea una conexión / un thread.

- Nula compresión de datos.

- No se usa el *Write-Buffer* que proporciona HBase.

- *Write Ahead Log* (WAL) está activado por defecto. Cuando escribimos datos en Hbase, estos van al WAL y las *MemStores* de Hbase. En caso de desconexión

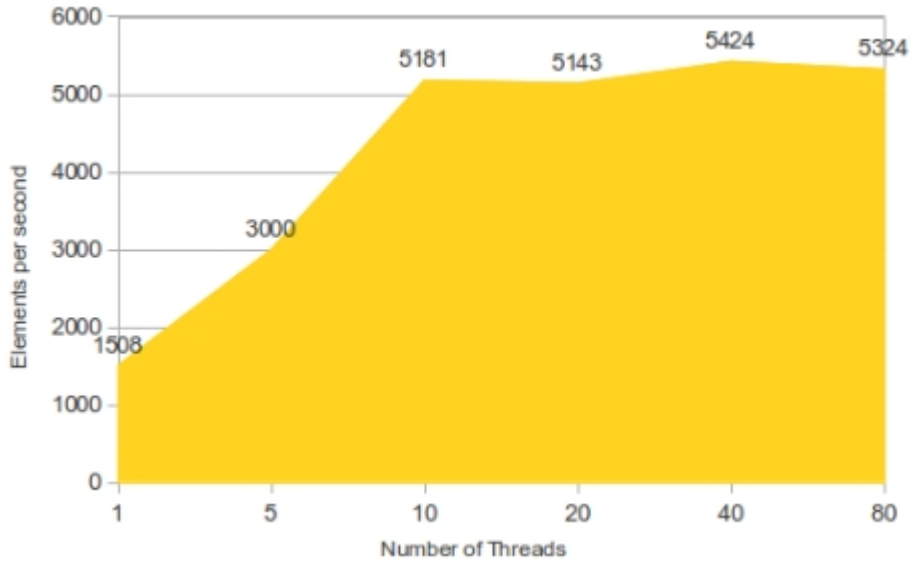
de un *RegionServer,* los datos recientemente insertados pueden ser replicados de nuevo.

- Todos los archivos a importar se encuentran en el sistema de archivos *ext4.* No se usa el sistema de datos distribuido Hadoop HDFS.

2º Etapa: Mejora de la aplicación gracias al uso de threads.

En esta etapa se modificó la aplicación descrita anteriormente para hacer uso de threads. De este modo podemos aprovechar mejor el número de threads que HBase usa a la espera de conexiones y transferencia de datos, así como los 12 núcleos que posee nuestra computadora. La clase *HbasePool* permite crear un conjunto de conexiones a HBase y usarlas en threads distintos. También se usó el *Write-buffer* que ofrece la API de HBase, lo que se tradujo en un incremento del número de elementos insertados por segundo.

En la siguiente gráfica se muestra el rendimiento obtenido gracias a la paralelización en la inserción de datos.



La figura muestra que se alcanza el máximo de inserciones de elementos por segundo usando ~10 threads, lo que equivale a ~5200 elementos por segundo.

Esta implementación mejora sustancialmente a la primera pero no es suficiente pues todavía continúan algunos de los problemas enumerados anteriormente como la compresión de datos, WAL, Hadoop HDFS o el problema de solo usar un *RegionServer* (más información completamente detallada en el anexo Págs. 56-57).

En estas implementaciones, WAL siempre estará activo pues premiamos integridad de datos a velocidad cuando hacemos uso de la API de HBase. En posteriores etapas WAL esta desactivado por defecto (HBase – MapReduce).

3º Etapa: Uso del algoritmo MapReduce.

HBase usa como sistema de almacenamiento de datos el sistema de archivos Hadoop Distributed File System (HDFS) y este a su vez permite usar su framework de procesamiento de datos Hadoop MapReduce, el cual es una implementación del paradigma MapReduce. Este framework permite procesar y generar grandes cantidades de datos de forma paralela usando un número cualquiera de computadoras.

HBase está completamente integrado con Hadoop MapReduce, por lo que su uso es sencillo y altamente recomendable cuando se requiere analizar grandes cantidades de datos.

Para hacer uso de este framework, se instaló en cada uno de los nodos del clúster HBase, Hadoop MapReduce (Un *JobTracker* en el nodo maestro de HBase y un *TaskTracker* en cada *RegionServer* del clúster (para más detalle sobre HDFS y Hadoop MapReduce, así como de que parámetros se usaron y como se mejoró el rendimiento de Hadoop, Págs. 41-43 y 52 del anexo).

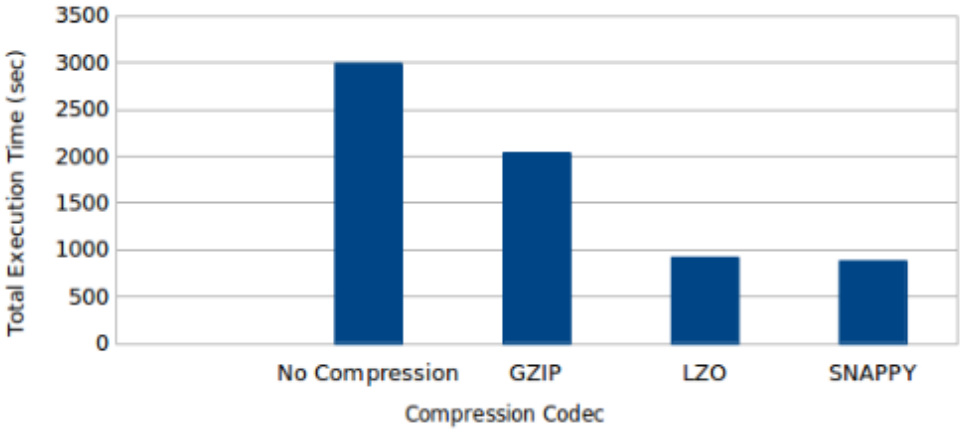
Una primera solución fue la creación de un trabajo MapReduce en Java usando las librerías que proveen Apache Hadoop y HBase. Este último permite importar datos directamente generados en el formato de almacenamiento que usa (*HFiles*) y Apache Hadoop posee una librería que permite crear dichos archivos como salida de un trabajo MapReduce. Esto es exactamente lo que la primera solución MapReduce implementada hace: Lee todos los datos desde HDFS y genera los archivos finales de HBase, *HFiles,* que pueden ser introducidos directamente en la base de datos usando la herramienta llamada HBase *Bulk Load*.

La primera versión del código MapReduce reveló un problema ampliamente discutido en el mundo de HDFS/MapReduce: Los archivos de datos de tamaño menor que el tamaño de bloque de HDFS (64 megas por defecto). En una primera fase del algoritmo MapReduce, este analiza el tamaño de los datos que tiene que procesar y "divide" dichos datos de forma que cada "división" es tratada por un *mapper* (primera fase de MapReduce). Los archivos que contienen los datos a insertar resultaron ser demasiado pequeños, por lo que el algoritmo MapReduce generaba un gran número de *mappers*, lo que se traducía en una pérdida de rendimiento. La solución fue crear un único y gran

SequenceFile (un tipo de archivo usado por Hadoop) que contenía todos los ficheros a tratar. Este tipo de archivos son divisibles, lo que los convierte en ideales para HDFS y MapReduce.

La compresión de los datos cuando se usa MapReduce puede resultar de gran ayuda. Es por esto que para la realización de este proyecto se estudió y probó con la compresión de los datos que generan tanto los *mappers* como los archivos finales que generan los *reducers*: *HFiles.*

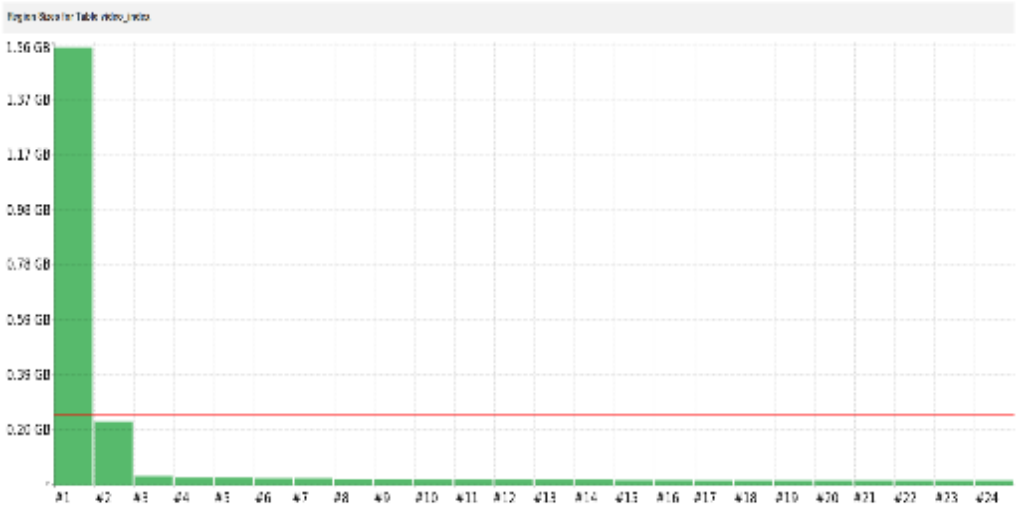
El algoritmo de compresión Snappy resultó ser el mejor de los tres que se estudiaron (GZIP, LZO y Snappy). El resultado final muestra que el tiempo de ejecución total de la importación de datos usando la compresión Snappy es de 879 segundos. Snappy es un códec de compresión de datos desarrollado por Google que premia la velocidad por delante de la compresión. Este códec comprime entre un 20% y 100% menos los datos que GZIP, pero necesita muchos menos ciclos de CPU. Es por esto y numerosos estudios sobre compresión de datos en Hadoop (Págs. 60-61 del anexo) que se decidió continuar el proyecto usando Snappy.



Después de importar todos los datos haciendo uso de MapReduce, los logs de Hadoop MapReduce así como HBase revelaron que todos los datos terminaban almacenados en un solo *RegionServer* y no repartidos entre todos ellos. Esto se debe a que por defecto, las librerías Hbase MapReduce crean tantos *reducers* como regiones tiene la base de datos en la que se desea importar datos. Pero si la base de datos está vacía, esta estará solamente compuesta/hará uso de una región. Este era exactamente el caso que revelaban los logs.

HBase permite crear tablas con un determinado número de regiones en vez de una región por defecto, por lo que el siguiente paso fue la pre-creación de la tabla que usan los datos con 24 regiones (el clúster Hadoop MapReduce

desplegado para la realización del proyecto posee un total de 24 espacios para *reducers*, por lo que si se usan los 24 a la vez, el trabajo MapReduce puede ser terminado en una sola "ola" de *reducers*). El seguimiento de los logs reveló que aunque entonces sí que se usaban más *reducers* y cada uno de ellos generaba los *HFiles* en su región (24 regiones en total), alguno de ellos analizaba más de diez veces la cantidad de datos que otros *reducers* examinaban. La siguiente figura muestra como la región número uno almacena el 70% de los datos.

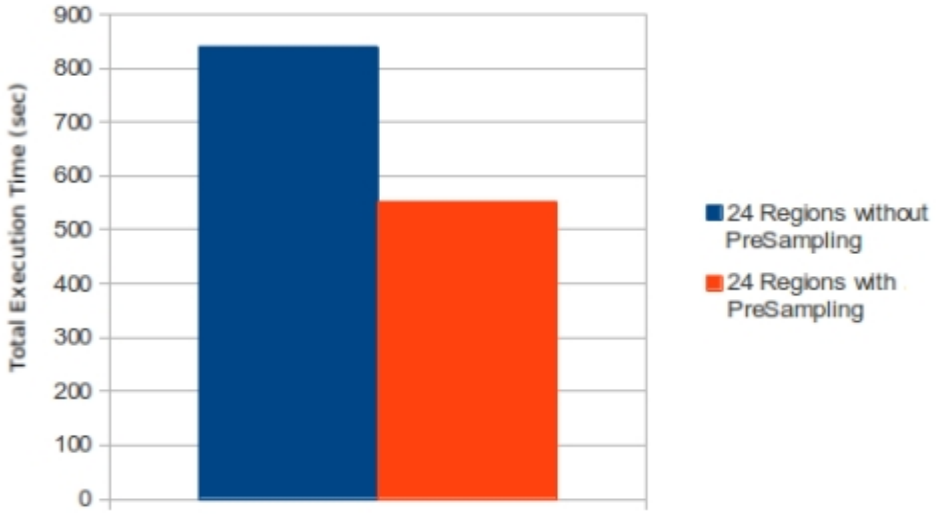


Este fenómeno es conocido como *Skew Data* en el mundo MapReduce y se refiere a una carga de trabajo no balanceada. En el caso a estudiar, esta no se encuentra balanceada porque algunos de los *reducers* analizan un número mayor de datos que otros, y esto se debe a las claves principales usadas en nuestra base de datos no están igualmente distribuidas. Es decir, una gran cantidad de ellas son muy similares (ej.: un alto porcentaje de claves principales empiezan por 'aaaa' y pocas por 'zzzz').

Para resolver este problema, se desarrolló una ligera herramienta en Java que hace uso del algoritmo MapReduce y que toma muestras de las salidas de datos de los *reducers* y genera un archivo con ellas. Después, dicho archivo puede ser usado en combinación con *TotalOrderPartitioner* de Hadoop para establecer que claves se envían a que *reducer* o como entrada de la herramienta de HBase que permite crear una tabla con las regiones, y los puntos iniciales y finales de claves que almacenaran (*Admin.createTable(table, splitPoints)).*

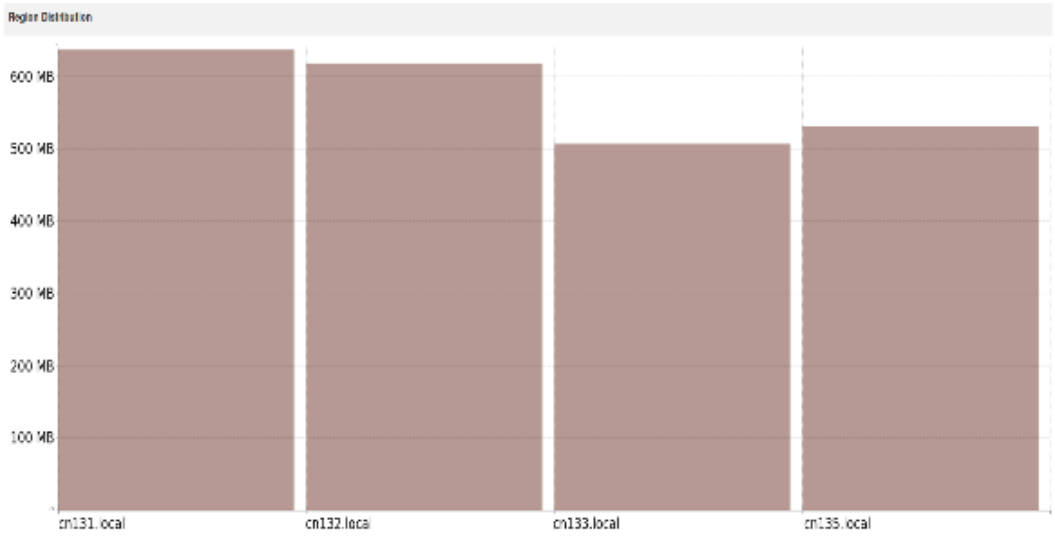
Este archivo asegura que los datos serán distribuidos en el clúster HBase de un modo más uniforme.

El tiempo de ejecución de esta herramienta es de tan solo 83 segundos. El tiempo total de ejecución del principal trabajo MapReduce se muestra en la siguiente figura:



El tiempo total de importación de datos con MapReduce haciendo uso de la herramienta de muestreo de datos es de 552 segundos, lo que supone una mejora del 34.29% respecto al mismo trabajo MapReduce sin pre-muestreo de datos.

La siguiente figura muestra el estado final del clúster HBase (cada columna representa un *RegionServer* de nuestro cluster y su altura determina la cantidad de datos que almacenan cada uno).



Para más información sobre esta herramienta de muestreo y su impacto en HBase ver páginas 62-66 del anexo.

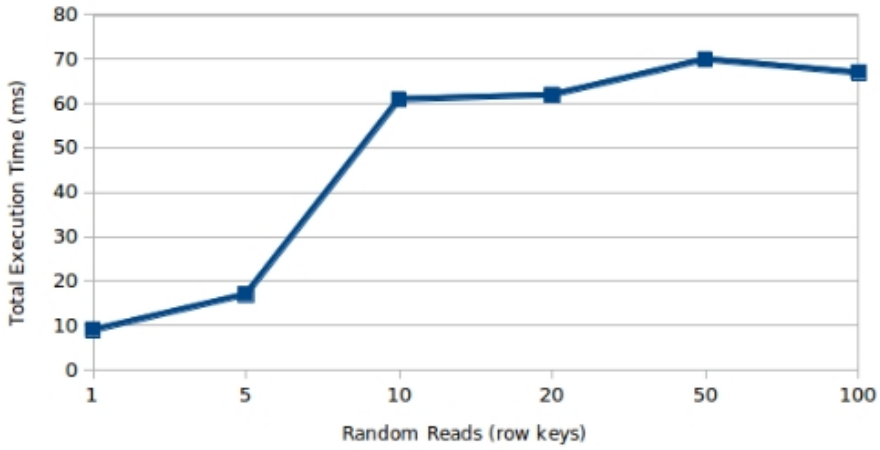
4º Etapa: Tuning de Hadoop.

En esta etapa se estudiaron diferentes parámetros de un clúster Hadoop MapReduce con el fin de mejorar el rendimiento ofrecido por el mismo. Siguiendo algunas publicaciones en el campo de mejora de Hadoop se modificaron parámetros como el tamaño de bloque en HDFS y se tomaron medidas. Dicho experimento revelo que el mejor tamaño de bloque para el trabajo eran 128MB. Dicho tamaño crea menos *mappers* en el trabajo MapReduce y mejora el tiempo total de ejecución en unos segundos.

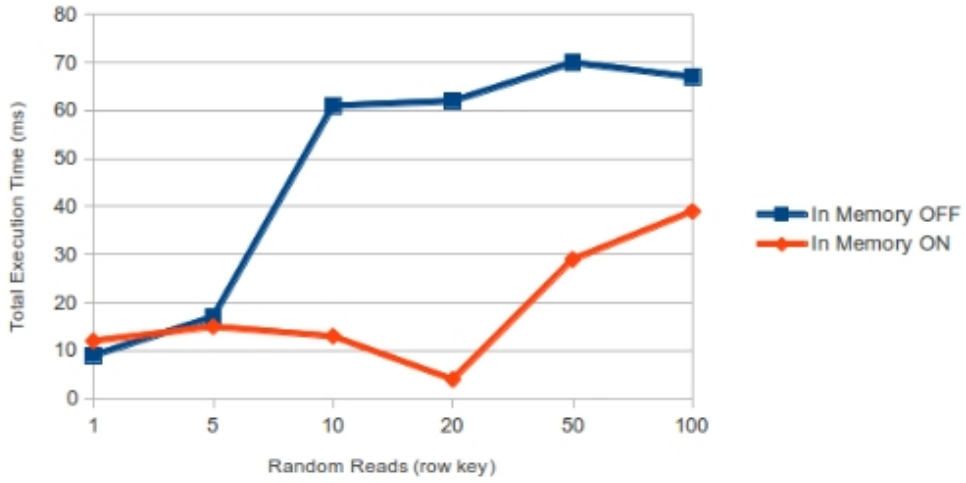
También se modificaron tamaños de buffer donde los *mappers* y *reducers* acumulan datos para tratarlos más adelante (Págs. 66-70 del anexo para ver todos los experimentos y sus impactos en el trabajo MapReduce).

1.2.7 Fase de recuperación de datos

La fase que siguió a la importación de los datos fue la lectura de los mismos desde HBase. Para estudiarla y llevarla a cabo se pidieron a la empresa las consultas más típicas con las que estaban teniendo problemas tanto de escalabilidad como de cualquier otra índole. La mayoría resultaron estar relacionadas con la lectura de datos aleatorios, es decir, la lectura de un número aleatorio de filas no consecutivas. Estas consultas se simularon en el clúster HBase mejorado gracias a la fase previamente descrita y con los datos ya importados. Las pruebas de lectura se implementaron en Java haciendo uso de la API de HBase y simulando las consultas SQL que hace la empresa en su clúster. Los resultados obtenidos se muestran en la figura aquí presentada:



Gracias a la lectura de las investigaciones que se están haciendo y se han hecho en este campo de las NoSQL así como del mundo de Big Data en general, se consiguieron mejorar los resultados iniciales. Las técnicas usadas han sido caches de lectura, diferentes propiedades de HBase para la lectura más rápida de partes concretas de los datos (In-Memory caches), diferentes tamaños de bloque para los *HFiles* de HBase adaptados a los datos de la empresa y Bloom Filters.



La figura muestra la mejora obtenida en la lectura de datos al usar las *In-Memory* caches de HBase (caches que pueden almacenar filas de datos enteras y/o la clave principal).

Para más información ver páginas 71-76 del anexo.

1.2.8 Fase de Benchmarking / Comparación

La última fase consistió en la comparación de nuestro clúster HBase totalmente configurado para las características de la empresa con un clúster MySQL similar al que dispone la empresa en sus servidores. Para ello, se desplegó un clúster MySQL de 5 nodos en Triton, y se configuró para obtener un rendimiento mayor y más cercano al de un sitio en producción y no uno por defecto. MySQL no ofrece la capacidad de *clustering* que HBase posee. Es por esto que se modificó la inserción y lectura natural de datos de MySQL (para leer/insertar/actualizar datos se usa el hash de la clave principal dividido entre el número de nodos. Esta fórmula determinará donde se sitúa el valor). Todas las pruebas de rendimiento se hicieron haciendo uso de la herramienta de código libre Yahoo! Cloud Serving Benchmark (YCSB), un estándar en la comparación de bases de datos NoSQL y SQL hoy en día.

Una vez desplegados HBase y MySQL, se procedió a realizar diversas pruebas que simulan distintos escenarios, tanto de escritura como de lectura en condiciones de estrés máximo y se documentó como se comportaban.

Las pruebas realizadas consistieron en tres tipos de trabajo de 1.000.000 de operaciones de distinto tipo cada uno según el test y fueron ejecutadas desde un nodo de Triton:

Tipo de trabajo	Inserciones %	Lectura %	Actualización %
Importación de datos	100	-	-
Mayoría de lecturas	-	95	5
Lecturas y Actualizaciones	-	50	50

Conclusiones:

- Importación de datos: HBase no tiene rival, llegando a importar más de 20.356 datos por segundo.

- Mayoría de lecturas: MySQL presenta una latencia de lectura menor que HBase, pero cuando el número de operaciones por segundo es mayor que 9734, MySQL se satura y su latencia empieza a aumentar.

- Lecturas y Actualizaciones: la latencia de actualización en HBase es mínima: 0,5 ms, frente a latencias de 30-90 ms en MySQL dependiendo de las operaciones por segundo.

Todas las pruebas se encuentran completamente detalladas y sus resultados explicados y presentados en graficas en el anexo. Págs. 77-82.

1.3 Trabajo Futuro

En este apartado se comentan algunas de las mejoras y trabajo futuro que se podría realizar siguiendo el tema estudiado (Págs. 83-84 del anexo):

- El esquema HBase utilizado para nuestros experimentos ha demostrado funcionar correctamente, pero se podría considerar un re-diseño en el que solo se usara una *ColumnFamily*. Esto nos daría un mayor control del comportamiento de HBase: cantidad de *StoreFiles,* tamaños de bloque y de caches de HBase.

- Usar *Short-Circuit*: Es una característica de HBase que permite saltarse el *DataNode* en la recuperación de datos de HBase.

- Usar más de un disco duro para los trabajos MapReduce.

- Hacer uso de HOYA. HOYA es una aplicación que permite desplegar *RegionServers* temporales en HBase durante altas cargas de trabajo o como en nuestro caso, durante la importación de datos haciendo uso de MapReduce.

- Usar *InfiniBand* en vez de *Gigabit Ethernet* como red de comunicaciones.

- Desplegar y testear Phoenix sobre HBase: Phoenix es un driver JDBC que permite realizar consultas SQL en HBase.

1.4 Resultados y Conclusión

En este proyecto final de carrera hemos presentado y evaluado métodos relacionados con la escalabilidad de los datos de una compañía. Para mejorar el rendimiento, hemos implementado un clúster HBase desde cero junto con diferentes soluciones basadas en algoritmos Big Data, como por ejemplo MapReduce.

Este proyecto evalúa el rendimiento obtenido desde tres puntos de vista:

1) La importación de un gran conjunto de datos a una nueva base de datos NoSQL. En este punto, varias soluciones se han planteado, desarrollado y testeado, revelando los beneficios y desventajas de cada una de ellas.

2) El rendimiento que se obtiene de la lectura de esos datos en el clúster HBase previamente mejorado para las escrituras (importación de datos). Una vez más, varias ideas conceptuales se han desarrollado, testeado y los resultados han sido expuestos.

3) Finalmente, nuestro clúster HBase en su conjunto ha sido comparado con un clúster MySQL similar al que posee y usa la compañía.

Hemos sido capaces de mejorar el rendimiento en todas y cada una de las áreas. Importando datos hemos pasado de una solución que usa la API de HBase cuyo tiempo de ejecución era de más de 5 horas a una solución basada en MapReduce que nos ha permitido reducir el tiempo total a solo 8 minutos y 33 segundos. Detrás de estos simples números hay muchas mejoras y problemas solucionados, tales como:

- Revisión de todas las características y parámetros ofrecidos por HBase.

- Revisión de todas las características y parámetros ofrecidos por Hadoop MapReduce.

- Tamaños de bloque en HDFS.

- Compresión de datos.

- Java Virtual Machine (JVM) tuning.

- Distribución de datos desigual (*Skew Data)*.

- *Bloom Filters.*

- *In-Memory* caches.

- Tamaños de bloque de *HFiles*.

- Regiones en HBase.

Recuperando información, también hemos sido capaces de mejorar los primeros resultados obtenidos usando conceptos tales como caches de lectura, tamaños de bloque cacheados o Bloom Filters. Todos y cada uno de los resultados han sido estudiados, comentados y mejorados cuando ha sido posible.

Dejando de lado los resultados, hemos desarrollado una aplicación que no solo sirve para distribuir correctamente los datos desiguales que teníamos, sino que sirve para cualquier trabajo MapReduce-HBase que sufra de datos desiguales en la salida de los *mappers*, más conocido como *mapper output*.

Gracias a la herramienta YCSB hemos testeado cuidadosamente nuestro clúster HBase con todas sus mejoras y comparado con uno similar pero que use MySQL, tal y como tienen y usan en la compañía. En este apartado hemos podido estudiar y comprobar que HBase se comporta realmente bien en escenarios de escritura y que está cercano en rendimiento de lectura al clúster MySQL.

También merece la pena constatar que hemos mejorado la herramienta de Yahoo! para los test de HBase, haciéndola más cercana a escenarios reales (creación de pre-regiones).

Podemos concluir que hemos obtenido resultados suficientemente satisfactorios como para que la compañía cambie su sistema backend a HBase.

Scalability of a Cloud-Based Data Store: Improving HBase performance

Aalto University

Mario Cerdan, Keijo Heljanko,
Rıdvan Döngelci
firstname.lastname@aalto.fi

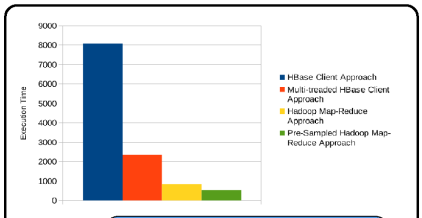
HELSINKI INSTITUTE FOR INFORMATION TECHNOLOGY

- We analysed and experimented with HBase NoSQL database to observe and improve its performance on video metadata.
- HBase is optimized for write heavy workloads. HBase is highly configurable and should be configured aggressively to get optimum performance.
- Java virtual machine, garbage collector, Memstore, compression and Hadoop parameters are crucial for HBase performance.
- Bulk data import to HBase is fairly important task for various purposes and should be done efficiently.

Summary

- Following four approaches are investigated for bulk data import to HBase:
  - **HBase Client Approach**
  - **Multi-threaded HBase Client Approach**
  - **Hadoop Map-Reduce Approach**
  - **Pre-Sampled Hadoop Map-Reduce Approach**

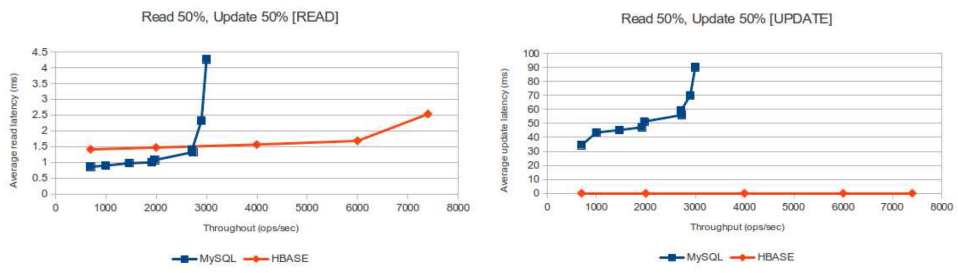
Bulk Data Import Approaches



Bulk Data Import Times

- We further compared HBase performance with MySQL performance to analyse benefits of NoSQL database.
- The benchmarking is based on industry standard Yahoo! Cloud Serving Benchmark (YCSB).
- 50% read and 50% update workload demonstrates that HBase is superior to MySQL with respect to updates and fairly competitive with respect to read performance.

HBase Performance Benchmarking



YCSB Results

Este poster fue creado para el proyecto final de carrera y presenta las conclusiones obtenidas a modo de resumen. Fue presentado en el Digile Data To Intelligence - D2I Spring Event:

"*D2I program is focused on big data, data reserves and user-centric service development. The aim of the program is to develop intelligent tools and methods for managing, refining and utilizing diverse data. The results enable innovative business models and services.****"***

Este evento reúne a empresas y universidades de Finlandia interesadas y/o que están investigando el paradigma Big Data y sus múltiples usos. Su principal objetivo es la divulgación de nuevos métodos, herramientas y estudios en el área de Big Data en Finlandia (más información aquí: datatointelligente.fi).

Por último, se presenta el email que la compañía nos envió después de presentarles los resultados obtenidos que resume cuan útil ha sido la investigación para ellos.

Email de uno de los ingenieros de la compañía:

> "*Thank You Mario and Keijo!*
>
> *Sorry for not telling this earlier but this was very impressive!*
>
> *Jani has already communicated this great package to our leads in server development and they were impressed about the results :)*
>
> *On behalf of PV I thank You Mario for the hard work and wish success for your future!*
>
> *Best Regards,*
>
> *Jarno*"