Escuela Politécnica Superior

Análisis de bases de datos y tendencias tecnológicas

Un caso de uso en Twitter para aplicaciones de análisis de sentimientos y opiniones

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Ricardo García Cebreiros

Tutor/es:
Armando Suárez Cueto
Patricio Martínez Barco

Julio 2016

Universitat d'Alacant
Universidad de Alicante

Justificación y Objetivos

El trabajo de fin de grado que aquí presento trata sobre bases de datos y el problema de rendimiento que se da en ciertas aplicaciones cuando el volumen de datos es muy grande y volátil.

La motivación del presente trabajo es la gestión y consulta de los datos necesarios en una base de datos. Es fácil imaginar que ante una cantidad de información de un volumen enorme, la respuesta inmediata que se espera en, por ejemplo, aplicaciones web, es difícil de conseguir. Los procesos se han diseñado para minimizar esta ralentización producida por el gran volumen de datos a tratar pero, no obstante, el primer cuello de botella es la propia base de datos.

El objetivo de este proyecto es estudiar diversas opciones de optimización un sistema de gestión de base de datos relacional (SGBDR) —MySQL— buscando mejorar el rendimiento, principalmente, de las consultas. La última meta, comprobar si la migración a un sistema NoSQL realmente redundaría significativamente en esa mejora.

Ha de decirse que se parte de un escenario ya establecido puesto que las dos aplicaciones están en producción y disponibles como aplicaciones web. Es decir, no se contempla probar SGBDR diferentes ni modificar el diseño del esquema de la base de datos.

Agradecimientos

A mis padres, por encargarse de mis estudios hasta ahora y soportarme en época de exámenes.

A mis compañeros y profesores que me han ayudado aún en horas intempestivas.

A mi tutor, Armando, por su tremenda ayuda y paciencia conmigo hasta el último momento.

Índice

Justificación y Objetivos 2
Agradecimientos 3
Introducción 7
Marco teórico 8
1. Taxonomía 9
Propiedades ACID 9
Propiedades BASE 11
Teorema CAP 11
Tipos de almacenamiento no relacional 12
2. Ranking de los Sistemas de BBDD 16
3. Puntos de vista de las diferentes tecnologías 17
Ventajas (Punto de vista NoSQL) 17
Críticas (punto de vista SQL) 19
4. Características y usos 21
MongoDB 21
Cassandra 22
Redis 23
HBase 24
Las aplicaciones GPLSI y el esquema de base de datos 25
Social Observer 25
Election Map 27
El esquema de base de datos 28
Funcionamiento de las aplicaciones 33
Objetivos y Metodología 34
Preparación del entorno 35

Importación de la BD 35
Consultas y primeras observaciones 36
Optimización de la BD 36
Monitorización de rendimiento (Benchmarking) 38
MySQLslap 38
Comando time 39
Primeras pruebas 39
Filtrado de las consultas 40
Desnormalización de la BD en MySQL 45
Benchmarking en la tabla desnormalizada 46
Filtrado de las consultas 47
Traslado a MongoDB 50
Exportación de los datos 50
Importación a MongoDB (mongoimport) 50
Benchmarking en MongoDB 52
Filtrado de consultas 54
Benchmarking de consultas de Administración 57
MySQL 57
MongoDB 57
Índices de datos 60
Consultas de administración 63
Conclusiones 66
Conocimientos adquiridos 68
Anexos 70
Anexo 0: Instalación En Linux Mint 70
Anexo 1: Importación en Linux Mint 71
Anexo 2: Consultas de aplicación 75
De usuario 75

De gestión 75
Anexo 3: Consultas de MySQL 80
Anexo 4: Optimización de la BD 86
Anexo 5: Tablas de Tiempos 90
Anexo 6: Desnormalización de la BD en MySQL 95
Anexo 7: Importación a MongoDB 97
Anexo 8: Consultas en MongoDB 99
Anexo 9: Consultas de administración 101
MySQL 101
MongoDB 103
Anexo 10: Índices 105
Bibliografía y enlaces 110
Otros enlaces consultados 113

Introducción

En el Departamento de Lenguajes y Sistemas Informáticos (DLSI) de la Universidad de Alicante, y más concretamente en el grupo de investigación de Procesamiento del Lenguaje y sistemas de Información (GPLSI) que está formado por investigadores de dicho departamento, se han desarrollado aplicaciones que explotan los datos que ofrecen las redes sociales. En particular Social Observer y Election Map son dos aplicaciones que trabajan a partir de los tweets de Twitter para analizar las opiniones que se vierten en ellos sobre temas o entidades concretos, en el primer caso, y sobre intención de voto en el segundo. Son aplicaciones fruto de las investigaciones en lo que se conoce como Procesamiento del Lenguaje Natural y, en particular, del área del análisis de opiniones y sentimientos.

Sin embargo, la enorme cantidad de información almacenada en la base de datos, producto del análisis de los parámetros necesarios para evaluar las opiniones reflejadas en los tweets, hace que se produzca un cuello de botella en estas aplicaciones, ralentizando la obtención de resultados y, en última instancia, el uso de dichas aplicaciones. Éste problema ha sido lo que ha planteado en los investigadores del GPLSI la idea de si sería necesario utilizar un sistema no relacional con el fin de conseguir mejor rendimiento en las consultas de datos.

En esta memoria, en primer lugar, hablaré sobre el nacimiento y los fundamentos de las conocidas como bases de datos NoSQL, partiendo de los problemas que la propia naturaleza de los sistemas relacionales no es capaz de solucionar satisfactoriamente, es decir, los motivos y para qué sistemas de información se hace necesario NoSQL. También incluye la imprescindible comparación entre Modelo Relacional y NoSQL y un análisis de las características que ofrecen algunos de los productos NoSQL más usados actualmente.

A continuación, una breve descripción de las aplicaciones aquí mencionadas, centrándome en el esquema de la base de datos, punto central de este trabajo.

La parte central de este trabajo es la carga de la base de datos y la medición de tiempos en diferentes condiciones de ciertas consultas que realizan habitualmente las aplicaciones del GPLSI. En primera instancia se hará en MySQL con las tablas normalizadas y desnormalizadas. Después se efectuará la migración de esos mismos datos a MongoDB, una base de datos NoSQL orientada a documentos JSON donde esas consultas SQL, debidamente traducidas, se ejecutarán igualmente midiendo tiempos y, finalmente, comparando todos ellos. La presente memoria la finalizaré con las conclusiones pertinentes, referencias bibliográficas y anexos con información adicional.

Marco teórico

Las bases de datos relacionales se llevan usando desde hace mucho tiempo pero en sistemas de datos realmente grandes, cuando se ejecutan millones de transacciones en periodos de tiempo suficientemente cortos, es muy fácil encontrar consultas SQL que saturan por completo el servidor durante segundos, minutos o incluso horas.

Es el caso de aplicaciones como Facebook o Twitter que no encontraban una solución transaccional a la ingente cantidad de datos que manejan. Estos ejemplos y muchos otros se encontraron con dificultades para dimensionar y predecir el comportamiento de las bases de datos relacionales, y también con un enorme problema de escalabilidad y disponibilidad.

La problemática generada por el rápido y gran crecimiento de los volúmenes de datos comenzó a requerir de soluciones escalables que lo soporten, es decir mayores recursos de procesamiento. Esto podía resolverse de dos maneras: mediante el escalamiento vertical (máquinas más poderosas, más memoria, procesadores, almacenamiento de disco, etc) o con el escalamiento horizontal (utilizar más máquinas pequeñas procesando en conjunto, mediante lo que se denomina clústeres). Esta segunda opción resulta más económica, y sus límites son más flexibles para el crecimiento que el escalamiento vertical.

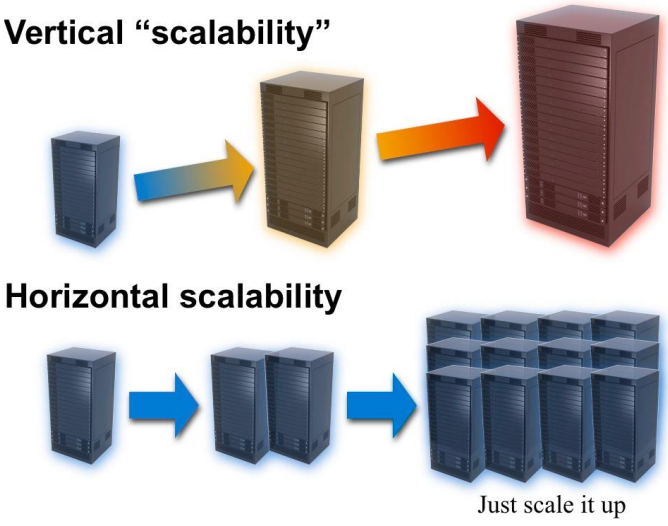


Fig. 1. Ejemplo ilustrativo de Escalabilidad Vertical VS. Horizontal [0]

Dado el auge de la informática en la Nube surge la necesidad de bases de datos altamente escalables, por lo que se tiene que optar por sistemas distribuidos. Y puesto que las bases de datos relacionales son complicadas de escalar horizontalmente, se pasa a usar en estos casos bases de datos no relacionales o NoSQL.

Los sistemas de almacenamiento no relacional proporcionaban un rendimiento en lectura y escritura de datos que los SGBDR (Sistemas Gestores de Bases de Datos Relacionales) no podían alcanzar, pero fue de las diferentes necesidades y retos técnicos con los que se fueron topando los sitios web más utilizados, sobre todo en cuanto a disponibilidad y escalabilidad, donde más se notó su impacto.

Ésta nueva tecnología cambiaba por completo la forma de trabajar con los datos, no era sólo un cambio de imagen y unos comandos diferentes para las consultas. Surgieron varios tipos de soluciones NoSQL, todas con un modelo de datos diferente (o directamente sin modelo, o de modelo libre); no sólo se modifica la forma de consultar o manejar los datos, se plantea una nueva filosofía para con las propiedades de una BD. Si el modelo clásico de Base de Datos Relacional debe cumplir las propiedades ACID (en inglés, Atomicidad, Consistencia, Aislamiento, Durabilidad); los defensores de las NoSQL plantean que, para sistemas distribuidos, es más indicado seguir el llamado Teorema de Brewer o CAP (en inglés, Consistencia, Disponibilidad, Tolerancia) y surge el acrónimo BASE (en inglés, Básicamente disponible, Estado flexible, Eventualmente consistencia), como opuesto del ACID (analogía a la relación entre los ácidos y las bases en química), para describir las propiedades de las NoSQL, donde prima la disponibilidad frente a la consistencia.

No obstante, lejos de llegarse a un consenso, empiezan las dudas sobre si realmente merecen la pena estas nuevas ‘soluciones’ o se podría conseguir los mismos resultados (o mejores, según lo que busques) con un sistema Relacional de toda la vida. En este trabajo vamos a explicar más en profundidad qué significa cada uno de estos términos y realizar una comprobación de las ventajas y críticas hacia las tecnologías no relacionales.

1. Taxonomía

Propiedades ACID

Las transacciones, como concepto y herramienta de las técnicas de bases de datos, se diseñaron para garantizar el correcto funcionamiento y la calidad de los datos almacenados. Una transacción es un conjunto de instrucciones que deben ejecutarse como si fueran una sola de tal forma que, si falla alguna y existe peligro de que la información en la base de datos quede en un estado inconsistente, se pueda revertir y volver justo al estado en el que se inició la transacción.

El problema reside fundamentalmente en la posibilidad de realizar transacciones simultáneas sobre un mismo dato. Supongamos, en el ejemplo clásico, un dato X al que se le incrementa en 1 unidad en una transacción y en otra se le decrementa en 2. En ambos casos se lee primero el valor de X, después se realiza la operación y, finalmente, se almacena el resultado. Es evidente que todos esperaríamos que el resultado final fuera, ejecutadas las dos transacciones, X-1. Pero si no hay un diseño y control adecuado de transacciones podría darse el caso de que ambas transacciones leyeran X sin tener en cuenta a la otra: el resultado sería X+1 o X-2, dependiendo de qué transacción almacenara la última.

Para evitar estos problemas de simultaneidad o, con más precisión, de concurrencia entre transacciones de bases de datos se definieron las propiedades ACID como un conjunto de leyes o reglas que cualquier sistema de base de datos debería garantizar:

- **Atomicidad**: Todas las operaciones en la transacción serán completadas o ninguna lo será.
- **Consistencia**: La base de datos estará en un estado válido 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.

Sin entrar en más detalles, aunque el resultado de observar todas estas garantías es, finalmente, que los datos almacenados son fiables después de operar con ellos, introduce una complejidad en la gestión de los sistemas de base de datos que, simplificando, incide en el tiempo de proceso y de respuesta. En sistemas de tamaño medio esta complejidad es inapreciable para el usuario. Sin embargo, en bases de datos grandes exige un muy cuidadoso diseño de transacciones, aún más en entornos distribuidos.

Precisamente, la tecnología NoSQL parte de la premisa de que no todas las bases de datos necesitan cumplir con las propiedades ACID, de ahí la ganancia en simplificación y, consecuentemente, en tiempo de respuesta y escalabilidad. También es evidente que para ciertos sistemas de información es crítico garantizar el cumplimiento de estas propiedades.

La tecnología NoSQL, sin embargo, establece unas propiedades centradas más en la disponibilidad que en la consistencia de los datos, las propiedades BASE.

Propiedades BASE

BASE son las siglas (en inglés) de:

- Básicamente Disponible (BA)
- Estado Flexible (S)
- Eventualmente Consistente (E)

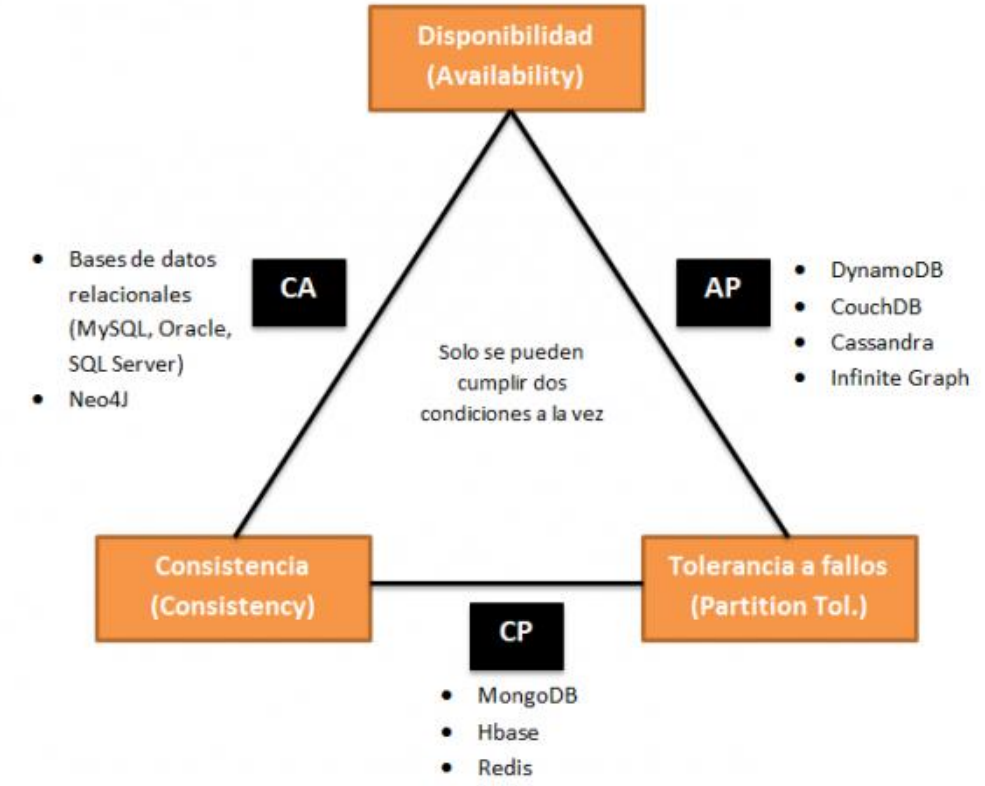
Las propiedades BASE son opuestas a las ACID: mientras que ACID es pesimista y fuerza la consistencia al finalizar cada operación, BASE es optimista y acepta que la consistencia de la base de datos esté en un estado flexible, por ejemplo, en una base de datos replicada, las actualizaciones podrían ir a un nodo quien replicará la última versión al resto de los nodos que eventualmente tendrán la última versión del set de datos. Permite niveles de escalabilidad que no pueden ser alcanzados con ACID. 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.

Siguiendo otras propiedades de tecnologías para el almacenamiento de datos, las diferentes tecnologías en BBDD, tanto relacionales como NoSQL, se pueden categorizar siguiendo el teorema de Brewer o CAP para sistemas distribuidos. Éste explica que los sistemas de bases de datos sólo pueden cumplir 2 propiedades al mismo tiempo de entre 3, Consistencia, disponibilidad y tolerancia a particiones. Los sistemas relacionales cumplirían Consistencia y disponibilidad; sin embargo, puesto que hay muchos tipos de almacenamiento no relacional, cada uno cumple otras propiedades según su objetivo.

Teorema CAP

Según Eric Brewer, profesor de la Universidad de Berkeley, California, los sistemas distribuidos no pueden asegurar en forma conjunta las siguientes propiedades: Consistencia (C), Disponibilidad (A) y Tolerancia a particiones (P), y sólo pueden cumplirse 2 de las 3 propiedades al mismo tiempo, utilizándose como criterio de selección, los requerimientos que se consideren más críticos para el negocio, optando entre propiedades ACID y BASE. Posteriormente, Seth Gilbert y Nancy Lynch de MIT publicaron una demostración formal de la conjetura de Brewer, convirtiéndola en un teorema [5]. El teorema de CAP ha sido

ampliamente adoptado por la comunidad NoSQL aunque desde varios y muy diferentes enfoques, como veremos a continuación.



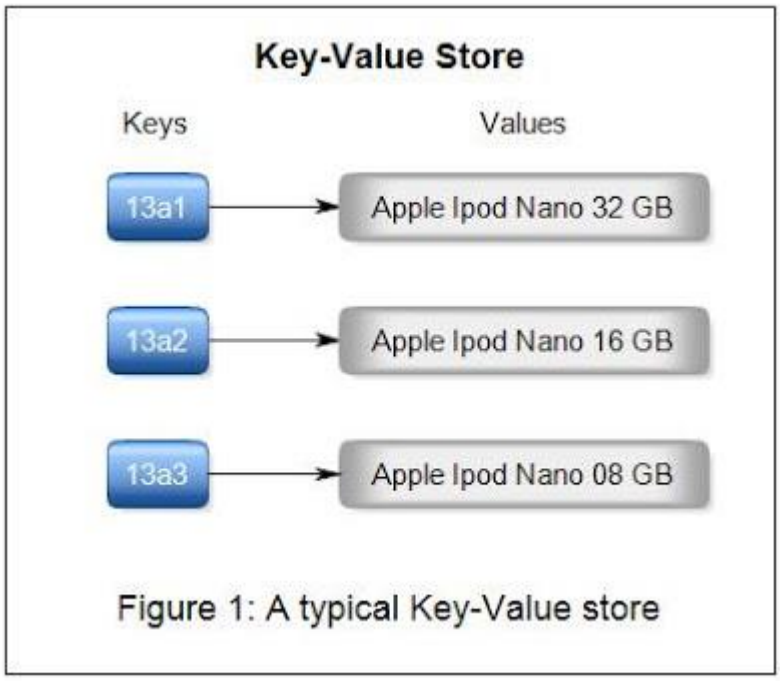
Clasificación de las BBDD según el teorema CAP [6]

Tipos de almacenamiento no relacional

Existen varios tipos de BBDD NoSQL, cada uno con sus particularidades y ventajas. Destacan tres, siendo los más utilizados en estos días:

- **Almacenamiento 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 DynamoDB de Amazon.

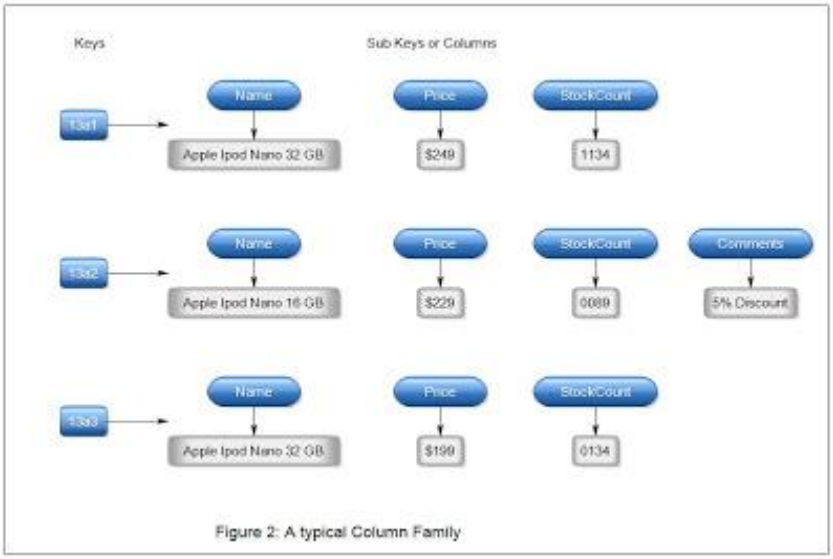


Esquema Clave/Valor [1]

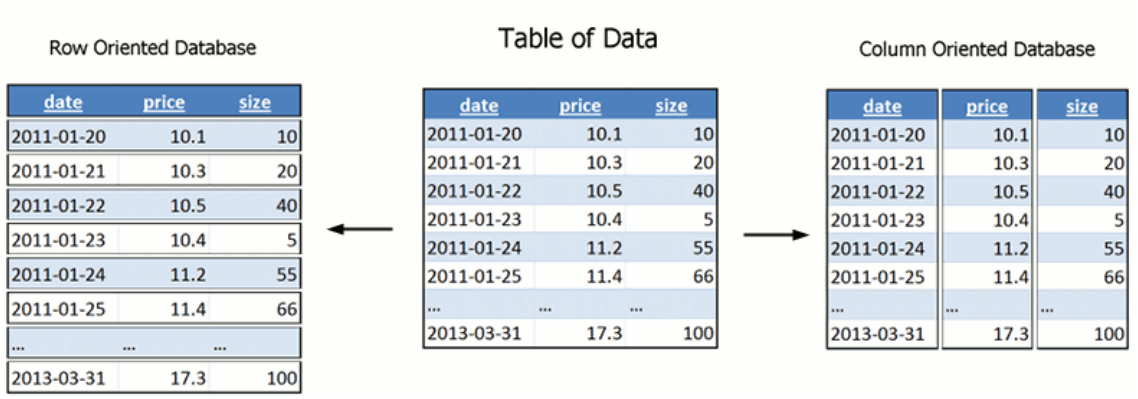
En 2009 surge Redis, desarrollado por Salvatore Sanfilippo para mejorar los tiempos de respuesta de un producto llamado LLOGG. Fue ganando popularidad sobre todo gracias a que en marzo del 2012 la empresa VMWare contrató a sus desarrolladores y patrocinó dicho sistema.

- **Orientadas a columnas / BigTable**

Estas almacenan la información por columnas, cada clave única apuntará a un conjunto de subclaves, que pueden ser tratadas como columnas. Se utilizan con más frecuencia para la **lectura de grandes volúmenes** de datos que para la escritura. **Añade cierta flexibilidad**, ya que permite añadir columnas a las filas necesarias sin alterar el esquema completo.



Esquema de Almacenamiento por columnas [1]



Comparación entre Alm. por Filas y Columnas [2]

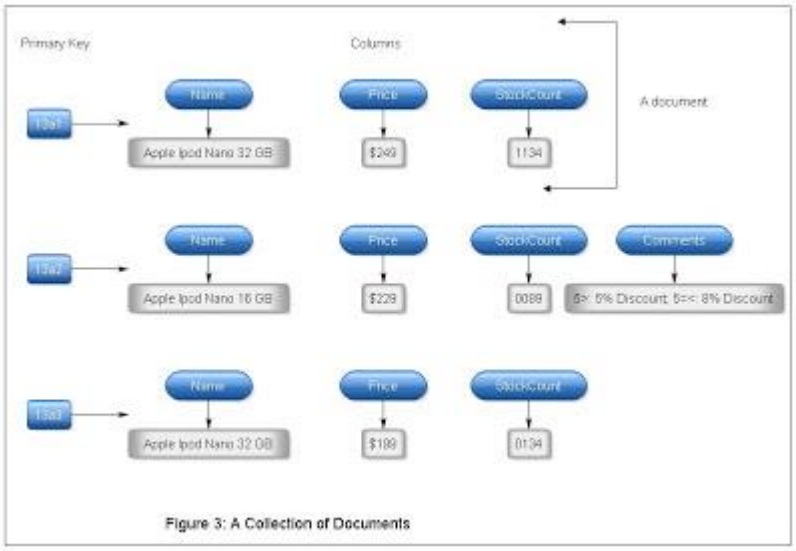
Este tipo de almacenamiento también es conocido como BigTable (de Google), que modela los valores como una terna (familias de columnas, columnas y versiones con timestamps); también es descrito como "mapa ordenado, multidimensional, persistente, distribuido y disperso.

En el año 2008, Facebook libera como proyecto open source Cassandra. Desarrollada por Avinash Lakshman (uno de los autores de DynamoDB) y Prashant Malik para darle mayor poder a su funcionalidad de búsqueda en la bandeja de entrada. En 2009 se transformó en un proyecto de Apache y termina siendo utilizada en Netflix, Twitter o Instagram.

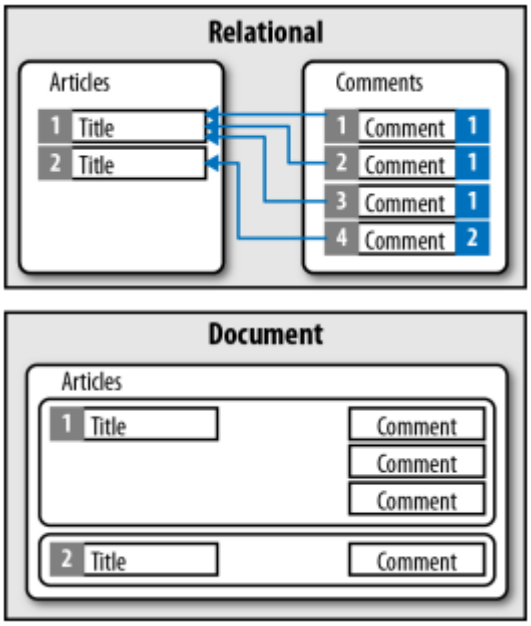
En el mismo año surge un clon a partir del BigTable de Google dentro del marco del proyecto Hadoop, que se denominó HBase. HBase comenzó como un proyecto de la compañía Powerset y luego fue integrado al proyecto Apache. Facebook eligió HBase en reemplazo de Cassandra para su nueva plataforma Facebook Messaging en el año 2010

- **Documentales / Basadas en documentos**

Los datos se almacenan en forma de documentos de tipo XML, JSON, etc, encapsulando los pares clave-valor en documentos y utilizando etiquetas para los valores de las claves. Ofrecen dos mejoras significativas respecto al modelo de columnas, la primera es que permiten estructuras de datos más complejas; la segunda mejora es el sistema de indexado a través de árboles B. Permiten realizar **búsquedas más potentes**, incluso con algunos motores se puede conseguir consultas parecidas a los JOIN de las BDs relacionales (usando referencias), por lo que son muy recomendables cuando necesitas **realizar consultas específicas**.



Esquema de almacenamiento documental [1]



Comparación entre modelo relacional y documental [3]

En 2007 la compañía 10gen desarrolla MongoDB, proyecto open-source basado en cloud, con una filosofía similar a la de CouchDB (alto rendimiento, disponibilidad y escalabilidad) y que utiliza documentos basados en JSON con esquemas dinámicos. Utilizada por eBay o Foursquare.

2. Ranking de los Sistemas de BBDD

280 systems in ranking, August 2015

Rank Aug 2015	Rank Jul 2015	Rank Aug 2014	DBMS	Database Model	Score Aug 2015	Score Jul 2015	Score Aug 2014
1.	1.	1.	Oracle	Relational DBMS	1453.02	-3.70	-17.83
2.	2.	2.	MySQL	Relational DBMS	1292.03	+8.69	+10.81
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1108.66	+5.60	-133.84
4.	4.	↑ 5.	MongoDB	Document store	294.65	+7.26	+57.30
5.	5.	↓ 4.	PostgreSQL	Relational DBMS	281.86	+9.04	+32.01
6.	6.	6.	DB2	Relational DBMS	201.23	+3.12	-5.19
7.	7.	7.	Microsoft Access	Relational DBMS	144.20	-0.10	+4.58
8.	8.	↑ 10.	Cassandra	Wide column store	113.99	+1.28	+32.09
9.	9.	↓ 8.	SQLite	Relational DBMS	105.82	-0.05	+16.95
10.	10.	↑ 11.	Redis	Key-value store	98.81	+3.73	+28.01
11.	11.	↓ 9.	SAP Adaptive Server	Relational DBMS	85.11	-2.10	-1.06
12.	12.	12.	Solr	Search engine	81.90	+2.61	+12.87
13.	13.	13.	Teradata	Relational DBMS	73.59	+1.28	+8.21
14.	14.	↑ 16.	Elasticsearch	Search engine	69.64	-0.52	+30.84
15.	15.	15.	HBase	Wide column store	59.95	-0.97	+18.03

Ranking de los sistemas para BBDD (agosto 2015) [4]

Resulta significativo que en apenas 5-6 años de vida que tienen algunos motores NoSQL se han llegado a posicionar hasta 3 sistemas, completamente diferentes en su modelo, entre los 10 sistemas de almacenamiento más utilizados. Seguramente sea debido a la evolución de las necesidades de ayer y las de hoy. Las primeras BBDD surgieron como una solución para el almacenamiento masivo de datos que se convirtió en necesidad para muchas empresas, conforme aumentaba el volumen de datos, éstas evolucionaron hasta surgir las BBDD Distribuidas, pero, o bien porque no son una solución suficiente, o bien por su complejidad, no fueron tan aceptadas como lo están siendo las soluciones NoSQL. Podríamos concentrar las posibles causas de la aceptación y popularización de los sistemas No relacionales en 3 grandes aspectos:

- Antes las bases de datos se diseñaban para ejecutarse en grandes y costosas máquinas aisladas. En cambio, hoy día, se opta por utilizar hardware más económico con una probabilidad de fallo predecible, y diseñar las aplicaciones para que manejen tales fallos que se consideran parte del "modo normal de operación"
- Los RDBMS son adecuados para datos relacionados rígidamente estructurados, permitiendo consultas dinámicas utilizando un lenguaje sofisticado. Sin embargo, hoy día, se desarrollan nuevas aplicaciones que se basan precisamente en datos con poca o ninguna estructura, dificultando su consulta por medios tradicionales.

3. Puntos de vista de las diferentes tecnologías

Lo cierto es que, a pesar del crecimiento que experimentan las tecnologías no relacionales en los últimos años, seguramente impulsado por la todavía más creciente tendencia a utilizar aplicaciones móviles, que no pueden utilizar un sistema de almacenamiento tradicional por problemas de coste o rendimiento; siguen existiendo diferentes opiniones sobre el uso de estas tecnologías en el entorno empresarial, siendo Oracle la crítica más dura hacia las nuevas tecnologías.

Ventajas (Punto de vista NoSQL)

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 (aquellas que necesiten una mayor disponibilidad y flexibilidad, en detrimento de la consistencia de los datos) éste set podría ser excesivo y el cumplimiento estricto de las propiedades ACID innecesario.

Alto rendimiento: Gracias a "sacrificar" la consistencia y centrarse en la disponibilidad de los datos podremos conseguir un mejor rendimiento.

Esto ha sido clave para empresas relacionadas con Web o aplicaciones móviles, donde reciben millones de consultas al mismo tiempo y no es demasiado problema la pérdida puntual de información, ya que, en el caso de una web, se vuelve a cargar la página y solucionado. Sin embargo en otros servicios como los bancarios no se tolera ninguna inconsistencia en los datos, es mejor tener seguridad en las transacciones que el que éstas tarden milisegundos en vez de minutos, al fin y al cabo, estamos hablando del dinero de las personas o empresas, algo que a nadie le gusta 'perder'.

Ejemplo de rendimiento: 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 *[7]*

Empleo de hardware más económico: Las máquinas pueden ser mucho menos complejas (y baratas), y en caso de necesitar más potencia, pueden ser agregadas o quitadas sin el esfuerzo operacional que implica realizar sharding en soluciones de clúster de RDBMS

Evitar el costoso mapeo objeto-relacional: SQL es un lenguaje de especificación o declarativo, se dice qué se quiere obtener, no cómo obtenerlo. Sin embargo, la mayoría de los lenguajes de programación actuales son procedurales u orientados a objetos donde lo fundamental es diseñar el cómo se obtienen los datos. Para ciertos programas de aplicación, adaptar los datos y las acciones a un formato cómodo para SQL supone un coste de tiempo innecesario. Según el bloguero y analista de BBDD Curt Monash, *"SQL es incómodo de adaptar para código procedural [...] cuando la estructura de tu base de datos es muy simple, puede no ser muy beneficioso"* [8]

Las NoSQL son diseñadas para almacenar estructuras de datos más simples o más similares a las utilizadas en los lenguajes de programación orientados a objetos beneficiando principalmente a aplicaciones de baja complejidad

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.

Según el bloguero Dennis Forbes, nótese que en este apartado el título es 'Las necesidades de un banco no son universales':

"El mundo de las firmas financieras, retailers y otros usuarios de SGBDR es muy diferente del escenario de las redes sociales

Si tuvieras que describir tus necesidades de datos en la red social en 3 aspectos serían:

- *Enormes islas de datos sin relación entre ellas*
- *Valor de las transacciones de usuario muy bajo*
- *La integridad de los datos no es crítica. (Si pierdes una actualización de estado o varios miles de ellas, o bien no se notará, o al menos no provocará nada importante)"*

Considerando estas 3 características, Forbes estipula lo siguiente en lo que concierne a las redes sociales y grandes aplicaciones web:

"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…" [9]

Particionar y distribuir un modelo de datos centralizado es caro y difícil: Los modelos de datos diseñados con una sola base de datos en mente, luego son difíciles (y caros) de particionar y distribuir. Bien porque al aplicar sharding (Fragmentación) se debe trabajar en otros sistemas de administración, o bien invertir una cantidad significativa de dinero en proveedores de SGBD para que operen la base de datos. A estos costes hay que sumarles que esto puede acarrear problemas o dificultades a largo plazo con las abstracciones de la aplicación, perdiendo información importante para la toma de decisiones como la referida a la latencia, fallas distribuidas, etc

Por ello, se sugiere diseñar el modelo de datos para que se adapte a un sistema particionado, aunque se empiece por un solo nodo.

En una entrevista a Ryan King, ex-Jefe Junior de Twitter encargado del uso de Cassandra:

"Tenemos actualmente un sistema basado en MySQL compartido + memcached, pero se está convirtiendo rápidamente prohibitivamente costoso (en términos de recursos humanos) de operar. Necesitamos un sistema que pueda crecer de un modo más automatizado y tenga alta disponibilidad" [11]

Críticas (punto de vista SQL)

Muchas de las críticas hacia las supuestas ventajas de las tecnologías NoSQL las plasmó Oracle en el documento *Debunking the NoSQL Hype* [12]. Irónicamente, meses después presentó su alternativa NoSQL, Oracle NoSQL.

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. Están condenadas a obtener cero ganancias económicas a largo plazo

No existe un líder claro: Al menos en lo que a modelo de sistema NoSQL utilizado. En el ranking mostrado anteriormente se puede observar que los modelos NoSQL más utilizados son todos diferentes.

El concepto "one-size-fits-all" ha sido criticado por los partidarios del software de código abierto 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. Cuando no tienes la herramienta adecuada para la funcionalidad adecuada, dices que no la necesitas [...] Sin embargo, los desarrolladores de herramientas se ocupan de añadir funcionalidades tan rápido como pueden porque, en la actualidad, ninguna de estas herramientas es suficiente para implementaciones de propósito general. Puedes usar un destornillador Phillips como una llave Allen, pero no es recomendable. Frecuentemente, se requiere de múltiples soluciones NoSQL porque cada una se adapta mejor para un caso de uso específico. [...] Incluso con la abundancia de bases de datos NoSQL, puede que no encuentres la que se adapte a tus necesidades y necesites implementar la tuya propia.

Escalabilidad no tan simple: Con frecuencia es más fácil decirlo que hacerlo, tal como lo demostraron los problemas de sharding que generaron el apagado en el Foursquare

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 —Capa de Presentación (Web), Lógica de Negocio (Aplicativa), Datos (Base de datos)— reestructurar los sistemas para que no ejecuten consultas con join o no poder confiar en el modelo de consistencia read-after-write

Se simplifica el teorema CAP a "Elige 2 de 3: Consistencia, disponibilidad, tolerancia a particiones": Alimenta la filosofía de desarrollo de aplicaciones antes mencionadas, alentando la inconsistencia y "disculpas". La elección sobre cuándo utilizar consistencia parcial y cuándo utilizar consistencia total no siempre está tan claro, y en ciertos casos de consistencia parcial no se observan beneficios reales 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

Modelos de datos sin esquema podría ser una mala decisión de diseño: Uno de los supuestos beneficios de usar soluciones NoSQL es la posibilidad de no tener un esquema definido y constante en el modelo de datos, como el modelo relacional o el entidad-relación, que determine cómo se almacenarán, organizarán y manipularán los datos de la BD. Esto les daría a las soluciones NoSQL mayor flexibilidad a la hora de trabajar con los datos, por

ejemplo, en el almacenamiento BigTable mostrado anteriormente, permite que los objetos que pertenezcan a un mismo conjunto de claves puedan tener diferente número de columnas, dando lugar a objetos que tengan una columna de Descripción y otros no la tengan; esto en un modelo relacional no podría pasar, todos los objetos tendrían una columna Descripción (por ejemplo), pero unos la tendrían vacía y otros no.

Si bien los modelos de datos sin esquema son flexibles desde el punto de vista del diseñador, son difíciles para consultar sus datos.

El problema de tratar con datos sin esquema recae en la capa de aplicación. Necesita saber qué información hay, dónde se guarda y cómo. Al tiempo que los datos evolucionan, la aplicación debe tratar con todos los diferentes formatos. [...] Modelos de datos desnormalizados son más caros de actualizar y mantener lógicamente consistente, y funciona mejor si los datos son de Solo-Lectura.

Tú no eres Google: Las NoSQL requieren de los mejores especialistas, que sólo compañías enormes como Google pueden invertir la cantidad de dinero y riesgo que supone utilizar o desarrollar una nueva tecnología para las bases de datos; una empresa menos poderosa no debería correr el riesgo, ni podría contratar al personal necesario.

4. Características y usos

Como hemos dicho, cada solución NoSQL se puede utilizar según diferentes criterios u objetivos. No es sólo seleccionar el tipo de NoSQL que mejor se adapte a tus necesidades para manejar los datos, en cada tipo, además, cada solución se ha desarrollado pensando en una forma de trabajar y tiene diferentes funcionalidades y se adaptan mejor a X caso de uso.

Éstas son las características de las soluciones NoSQL más utilizadas a día de hoy [13]:

MongoDB

Lenguaje: C++

Motivos para usarlo: Mantiene propiedades familiares de SQL

Licencia: GNU Affero General Public License (AGPL) -Drivers: Apache-

Protocolo: Propio, binario (BSON)

- Replicación Maestro/Esclavo (“failover” automático con conjuntos de réplicas)

- Fragmentación integrada
- Consultas como expresiones de JavaScript
- Ejecuta funciones de servidor aleatorias
- Mejora la ‘actualización en el sitio’ de CouchDB
- Usa archivos de memoria mapeada para el almacenamiento
- Rendimiento sobre funcionalidades
- Journaling (--journal) se activa mejor
- Limitación a 2.5GB en sistemas de 32 bits
- Búsqueda de texto integrada
- GridFS para el almacenamiento de grandes cantidades de datos + metadatos
- Indexado geoespacial
- Conciencia de centro de datos

Usar: Si necesitas consultas dinámicas. Si prefieres usar índices a reducir/mapear funciones. Para buen rendimiento en grandes BBDD. Si ibas a usar CouchDB, pero tus datos cambian mucho y llenan discos.

Ejemplo: Muchos de los usos que podrías darle a MySQL o PostgreSQL, pero sin el inconveniente de columnas predefinidas.

Cassandra

Lenguaje: Java

Motivos: Almacenar conjuntos de datos enormes “casi” en SQL

Licencia: Apache

Protocolo: CQL3 (como SQL, pero sin JOIN ni funciones agregadas)

- Consultas por clave o rango de claves (también por índices secundarios)
- intercambios personalizables para la distribución y replicación
- Los datos pueden expirar (usar en el INSERT)
- Escrituras más rápidas que lecturas
- Reducir/Mapear posible usando Apache Hadoop
- Todos los nodos son similares
- Replicación de centros de datos buena y fiable
- Contador de tipos de dato distribuido Distributed counter datatype.
- Permite escribir triggers en Java

Usar: Para almacenar tal cantidad de datos que no quepa en el servidor, pero usando una interfaz amigable.

Ejemplo: Análisis Web, para el conteo de visitas por hora, por navegador, por IP, etc. Colecciones de datos de grandes vectores de sensores. Logging de transacciones.

Redis

Lenguaje: C

Motivos: Tremendamente rápido

Licencia: BSD

Protocolo: parecido a Telnet, Binario seguro

- BD en memoria con respaldo en disco. Disk-backed in-memory database,
- Tamaño del conjunto de datos limitado por RAM (pero puede utilizar la RAM de varios nodos en clúster)
- Replicación Maestro-Esclavo, failover automático
- Estructuras simples de datos o valores por claves
- Operaciones complejas como ZREVRANGEBYSCORE.INCR & co (para estadísticas y puntuaciones)
- Operaciones de bits (por ejemplo para implementar filtros de floración)
- Tiene sets, union/diff/inter
- Tiene listas (también una cola; bloqueo pop)
- Tiene hashes (Objetos con múltiples campos)
- Conjuntos ordenados (tablas de alta puntuación, para consultas de rangos)
- Scripting Lua
- Tiene Transacciones
- Valores expirables (en caché)
- Pub/Sub permite implementar mensajería

Usar: Para intercambio rápido de datos con una BD de tamaño previsible

Ejemplo: Almacenamiento de precios en tiempo real. Clasificaciones. Comunicación en tiempo real. Cualquier sitio donde se usara memcached.

HBase

Lenguaje: Java

Motivos: Billones de filas X millones de columnas

Licencia: Apache

Protocolo: HTTP/REST (también Thrift)

- Usa el almacenamiento HDFS de Hadoop
- Mapeo/reducción con Hadoop
- Los predicados de las consultas vía filtros de escaneo y gets del lado del servidor Query predicate push down via server side scan and get filters
- Optimizaciones para consultas en tiempo real
- Una puerta de enlace de alto rendimiento a Thrift
- HTTP soporta XML, Protobuf, y binario
- Consola basada en Jruby (JIRB)
- Reinicio para actualizaciones menores y cambios en la configuración del balanceo
- Rendimiento de acceso aleatorio como MySQL
- Un clúster consiste en diferentes tipos de nodos

Usar: Ejecutar Mapeos/Reducciones en grandes conjuntos de datos

Ejemplo: Motores de búsqueda. Análisis de datos grandes. Sitios donde sea un requerimiento escanear tablas bidimensionales muy grandes.

Las aplicaciones GPLSI y el esquema de base de datos

En el Grupo de investigación en Procesamiento del Lenguaje y Sistemas de Información, cuyos miembros pertenecen en su mayoría al Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Alicante, se lleva ya varios años desarrollando herramientas que explotan la información publicada en redes sociales. Concretamente, Twitter está siendo la base de las líneas de investigación que trabajan sobre lo que se conoce como análisis de sentimientos y análisis de opiniones. Básicamente, este tipo de aplicaciones extraen información de textos escritos en lenguaje natural que permite determinar si una opinión o sentimiento es positivo o negativo en relación a un concepto o entidad en particular.

Uno de los usos más claros es la medida de la buena o mala reputación de empresas, especialmente para aquellas cuya actividad es principalmente la transacción comercial, la venta de productos.

Otro muy obvio también, tiene que ver con los procesos electorales periódicos en España o en cualquier otro país. Predecir o, al menos, detectar el sentimiento del potencial votante es lo que hacen las encuestas telefónicas o a pie de calle. Este mismo sentimiento se refleja en las redes sociales y puede ser identificado.

El GPLSI ha desarrollado y publicado dos aplicaciones que procesan los tuits de Twitter para mostrar tendencias de opinión a partir de etiquetas que se suministran como parámetro, esto es, etiquetas que identifican a personas u organizaciones.

A continuación se describen someramente las aplicaciones concretas que ofrece el GPLSI para, por último, dar información más detallada sobre la base de datos que las sustenta.

Social Observer

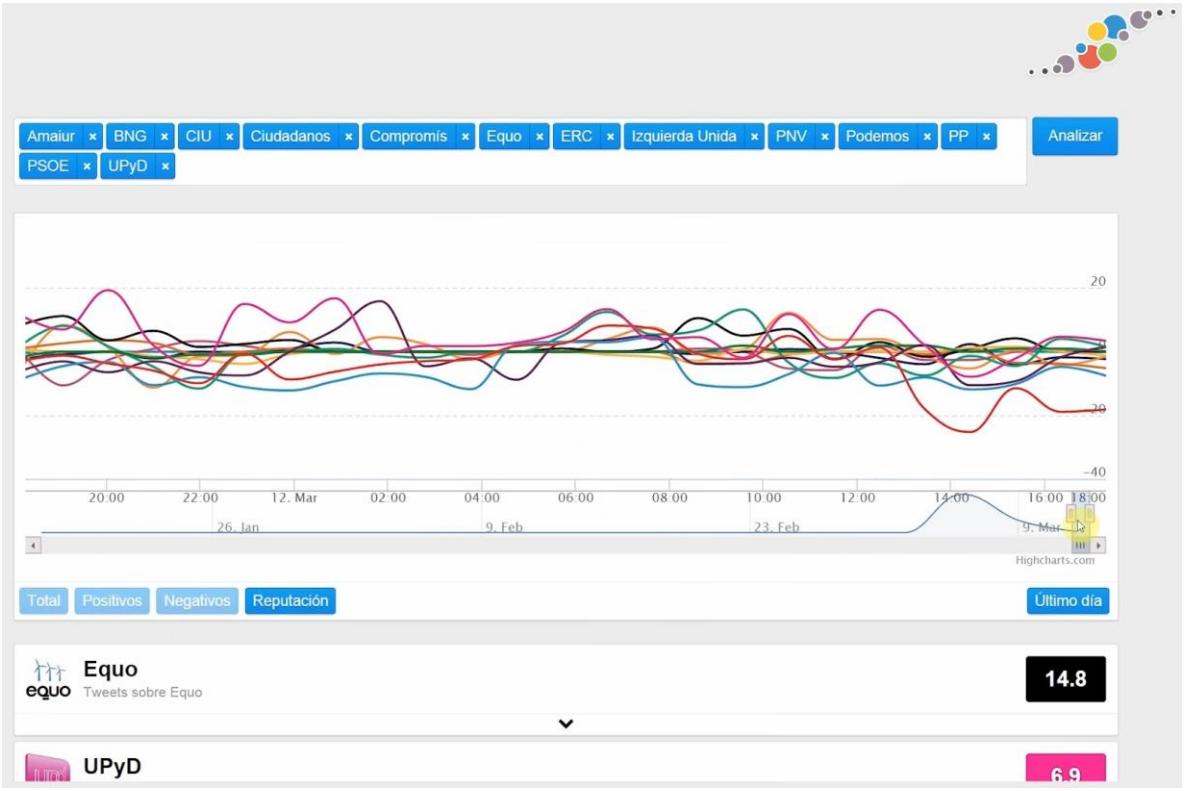
En la página de descripción del programa de demostración de GPLSI Social Observer[1] se nos explica que es una aplicación que recupera tweets de la red social de Twitter sobre un tema en concreto y, de forma automática, valora las opiniones expresadas en los mensajes. Esto permite realizar un seguimiento de las opiniones de la gente sobre un famoso, político, equipo de fútbol, producto, inversión en bolsa o, incluso, unas elecciones. De esta forma, y

[1] https://gplsi.dlsi.ua.es/gplsi13/es/node/249

gracias a esta herramienta se puede llegar a un análisis y predicción de opiniones y tendencias. Este análisis puede representarse tanto geográficamente en un mapa como con gráficos circulares, lineales y de barras.

El análisis de tendencias es un área de investigación relativamente reciente y de gran interés para medios de comunicación, empresas, agencias bursátiles y gobiernos entre otros. Permite predecir ciertos aspectos socioeconómicos y tomar las medidas oportunas en cada caso antes de que se produzcan. También permite que las empresas mejoren sus productos en aquellos aspectos que más desagrada a sus clientes y que conserven aquellos que más gustan. Incluso se puede perfilar estas herramientas para que se adapten a ciertos grupos de interés. Un ejemplo que puede ilustrar bastante bien el párrafo anterior es, por ejemplo, cuando alguien busca un hotel para sus vacaciones. Normalmente nos guiamos por ciertos factores como el número de estrellas y la opinión de los usuarios de ese hotel. El problema es que desconocemos cómo es la persona que da las opiniones y puede que un padre o una madre de familia busquen algo muy distinto a lo que buscaría un joven que quiere divertirse. Con esta tecnología de análisis de tendencias un sistema puede valorar que una discoteca cercana es bueno para un joven pero no es tan interesante para otras personas, o que el valor de un producto bursátil va a caer inminentemente.

Pero toda esta tecnología se basa en otra que analiza las opiniones de la gente y valora los sentimientos expresados en ellas. Así, la minería de opiniones (también llamada análisis de sentimientos) nos permite rastrear a través de Internet y de las redes sociales los comentarios que hace la gente sobre cualquier tema y valorar qué emoción expresa, si es positiva o negativa e, incluso, la intensidad.



El GPLSI Social Observer es una demo totalmente funcional de un sistema de minería de opiniones que trabaja sobre los tweets de Twitter para valorar cada uno de ellos y resumirlos en una valoración final de subjetividad. Es decir, Social Observer monitoriza los temas definidos en el sistema y, para cada tweet relacionado con dichos temas, valora si la opinión de la gente es positiva, negativa y en qué grado.

Aunque la demo trabaja únicamente con tweets, esta puede expandirse para trabajar con otras fuentes y añadirle más operatividad, como el análisis de tendencias mencionado o que el sistema sea capaz de aprender de sus errores para ir mejorando con el tiempo.

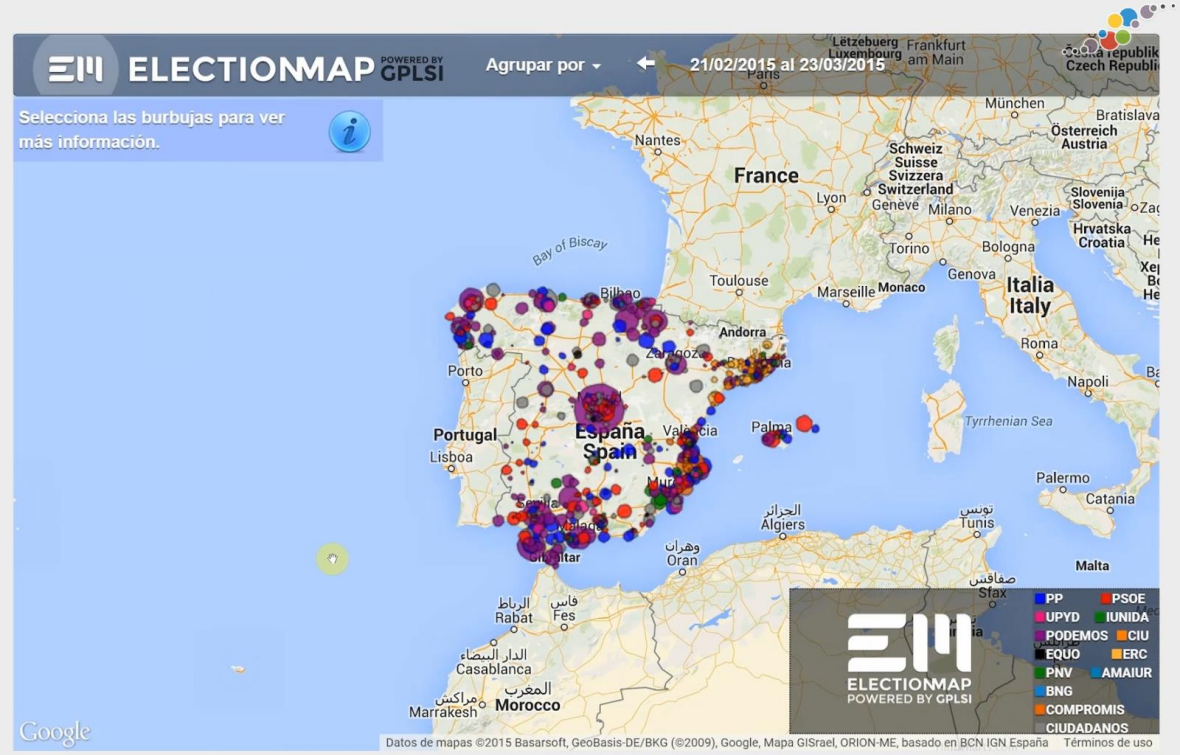
Election Map

"Election Map[2]: una representación geo localizada de intenciones de voto hacia partidos políticos sobre la base de comentarios de usuarios de Twitter."[22]

En la actualidad las redes sociales se han convertido en uno de los principales medios de distribución de información. Una de las redes más utilizadas en la actualidad es Twitter, y la cantidad de información que proporciona esta red social no ha pasado desapercibida para las grandes empresas y gobiernos. Tanto empresas como gobiernos han comenzado a guiar sus

[2] https://gplsi.dlsi.ua.es/gplsi11/content/mapa-de-tendencias-electorales

campañas de promoción fijándose en las opiniones que los usuarios de las redes sociales valoran en sus perfiles. Otro de los problemas de las campañas de promoción es que no tienen el mismo impacto en toda la geografía. Saber en qué regiones una campaña obtiene una mayor aceptación social.



Para cubrir estas necesidades surge Election Map. Esta herramienta muestra de forma gráfica la opinión de los usuarios de Twitter sobre temas relacionados con la política. No todas las opiniones pueden tenerse en cuenta, sólo sirven las opiniones positivas y localizadas, por lo que el conjunto de datos utilizado no es el 100 % de la información recogida sino una muestra representativa del total. Finalmente tras procesar este conjunto de datos se muestra una representación gráfica de intención de voto político agrupada por áreas geográficas representando así el apoyo de los usuarios de los medios sociales.

El esquema de base de datos

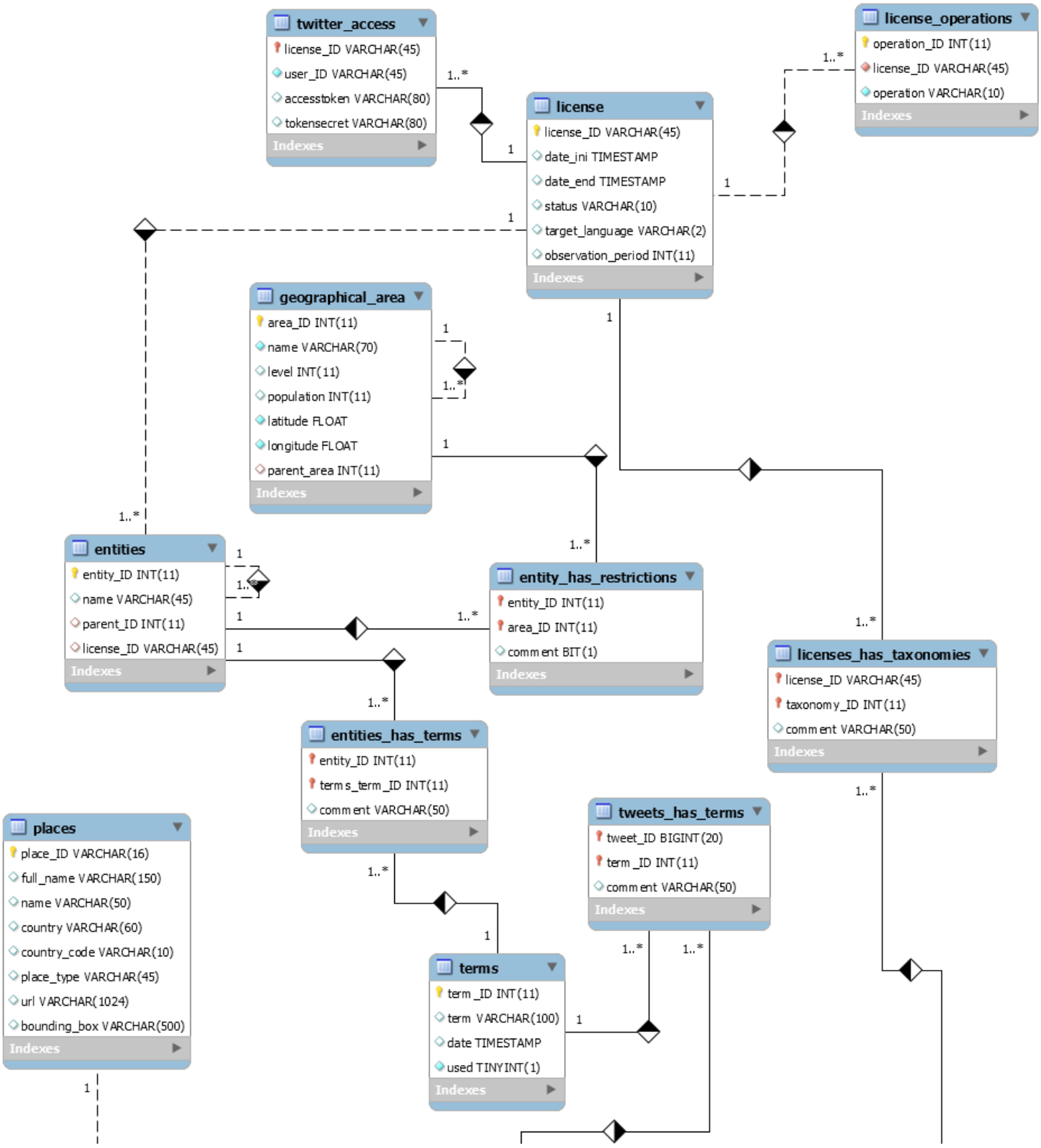
La BD que nutre a las dos aplicaciones anteriores se puebla mediante una descarga directa de los últimos tuits de la red Twitter. Estos tuits contienen información diversa y de utilidad para establecer datos demográficos y de lugar y tiempo importantes, aparte de la propia naturaleza del comentario analizado que puede ser relevante o no y de carácter positivo o negativo.

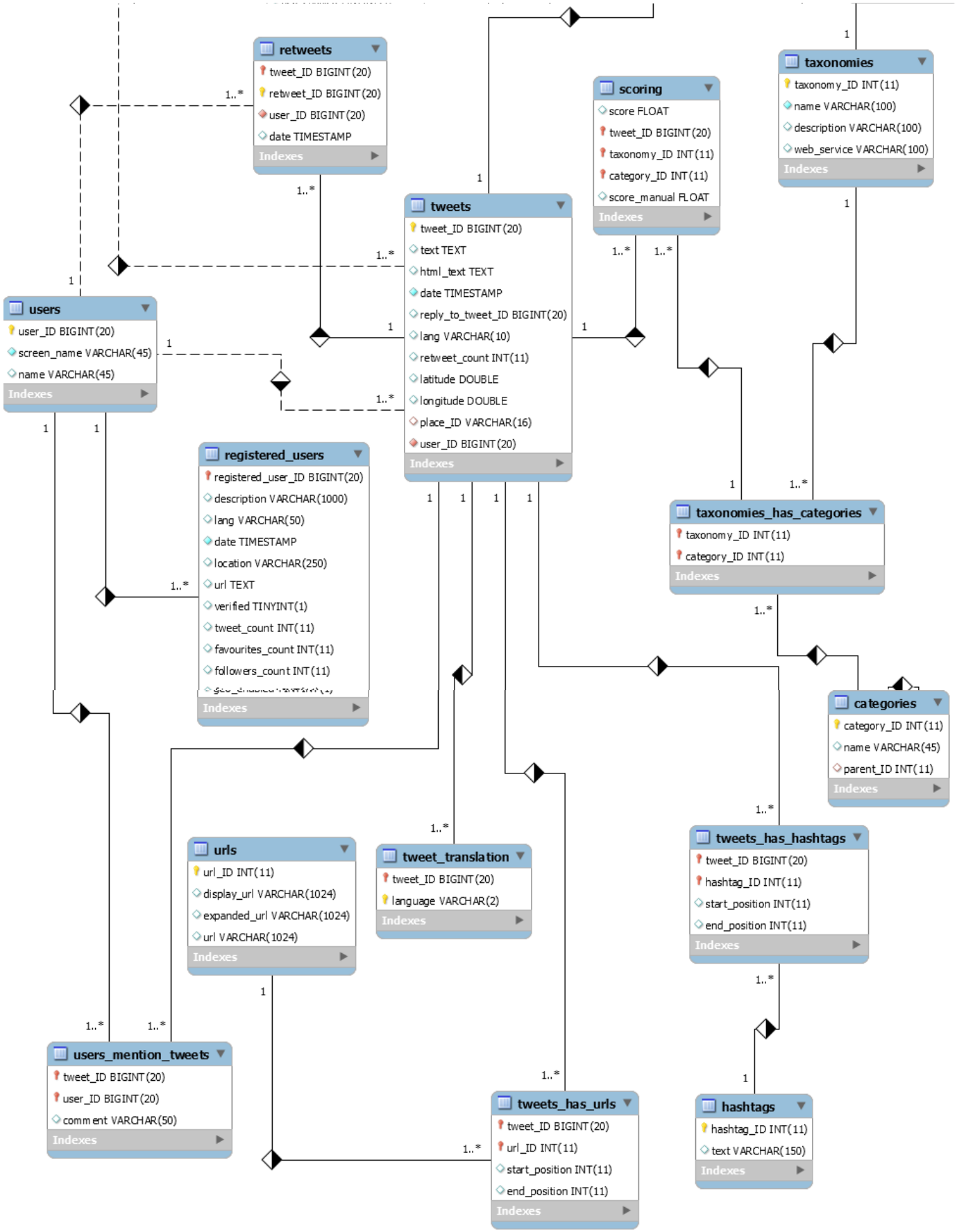
Una vez recuperada una cierta cantidad de información mediante la API que proporciona la propia empresa, los tuits son procesados para ser almacenados en la base de datos. De hecho, la mayoría de conceptos que se reflejan en el esquema de BD son útiles para definir las poblaciones de la muestra de la que se dará información en forma de gráficos de tendencia. Podemos ver este proceso como la extracción previa de toda la información potencialmente útil para que, cuando se haga una determinada consulta, pueda obtenerse la polaridad de la opinión y mostrar una evolución en el tiempo.

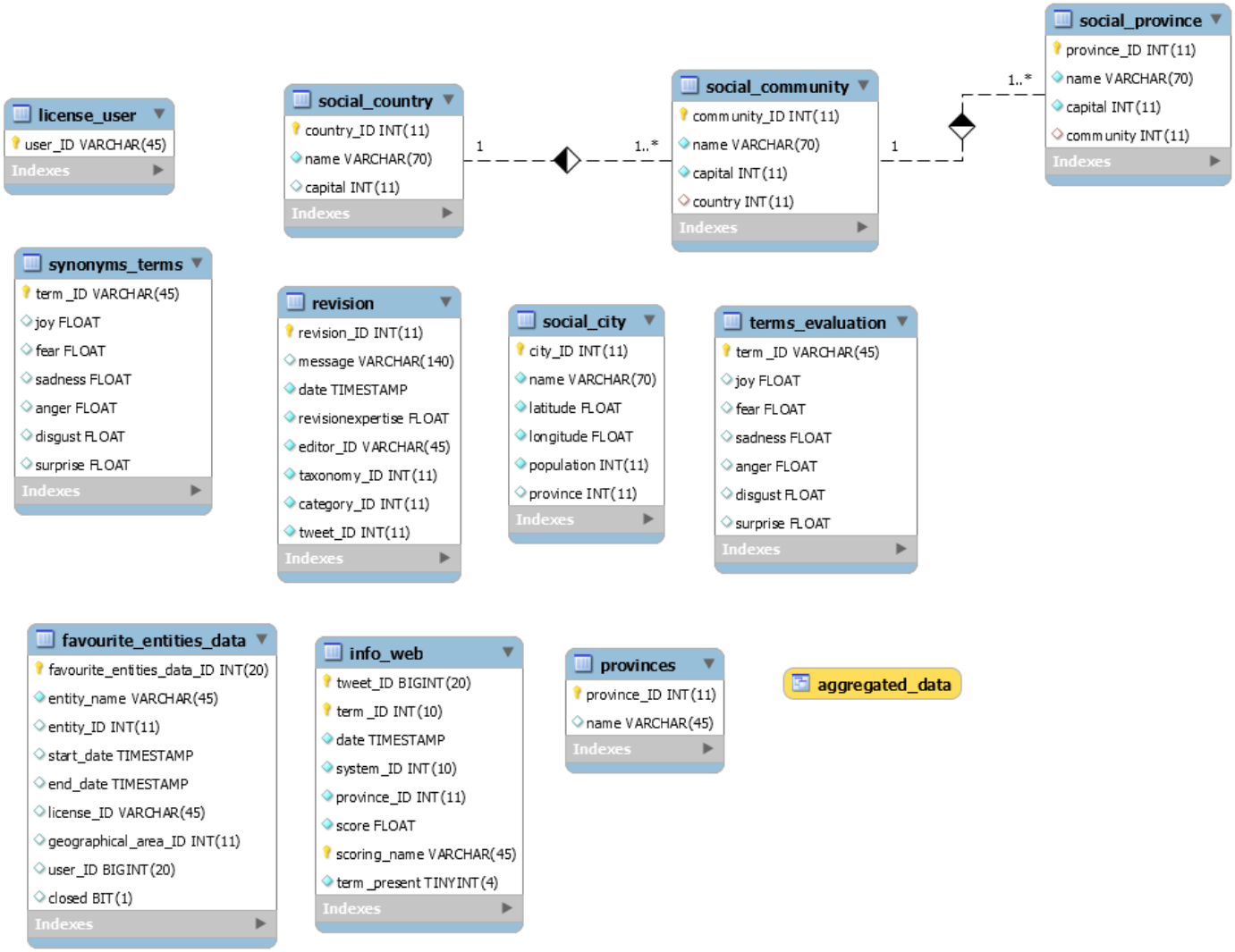
Evidentemente estamos hablando de una base de datos con un volumen importante —de hecho, se puede hacer tan grande como se quiera, todo depende del periodo de tiempo almacenado y del filtro previo que se haya podido hacer— y de muchas tablas relacionadas entre sí. Esto hace que el "*join*" necesario involucre a la mayoría de estas tablas, con filtros en las consultas habitualmente complejos. Es esta complejidad en las consultas la que se intuye como causa principal de la necesidad de optimización de la base de datos buscando la mayor rapidez de respuesta.

De especial importancia por el objetivo y desarrollo posterior de este trabajo son la vista "aggregated_data" y la tabla "aggregated_data_store". Precisamente el problema de rendimiento en la obtención de resultados hizo que los desarrolladores originales de las aplicaciones aquí mencionadas intentasen una desnormalización de las tablas buscando evitar la mayoría de *JOINs*. Curiosamente, "aggregated_data_store", una tabla que tiene en sus columnas toda la información a extraer de Twitter y que sería el resultado de desnormalizar todas las demás tablas, nunca ha sido usada. Probablemente por la dificultad de adaptar los programas para trabajar con esa única tabla —recordemos que el proceso de extraer la información de los tuits no es trivial y está fuertemente parametrizada por la imposibilidad de almacenar absolutamente todos los tuits—.

En definitiva, llegamos a la meta principal de este trabajo: evaluar las posibilidades de mejora sobre el esquema relacional actual de BD y las ventajas de migrarlo a sistemas que se suponen más adecuados para estos usos, una base de datos NoSQL.


twitter_access
license_ID VARCHAR(45)
user_ID VARCHAR(45)
accesstoken VARCHAR(80)
tokensecret VARCHAR(80)
Indexes
license_operations
operation_ID INT(11)
license_ID VARCHAR(45)
operation VARCHAR(10)
Indexes
license
license_ID VARCHAR(45)
date_ini TIMESTAMP
date_end TIMESTAMP
status VARCHAR(10)
target_language VARCHAR(2)
observation_period INT(11)
Indexes
geographical_area
area_ID INT(11)
name VARCHAR(70)
level INT(11)
population INT(11)
latitude FLOAT
longitude FLOAT
parent_area INT(11)
Indexes
entities
entity_ID INT(11)
name VARCHAR(45)
parent_ID INT(11)
license_ID VARCHAR(45)
Indexes
entity_has_restrictions
entity_ID INT(11)
area_ID INT(11)
comment BIT(1)
Indexes
licenses_has_taxonomies
license_ID VARCHAR(45)
taxonomy_ID INT(11)
comment VARCHAR(50)
Indexes
entities_has_terms
entity_ID INT(11)
terms_term_ID INT(11)
comment VARCHAR(50)
Indexes
places
place_ID VARCHAR(16)
full_name VARCHAR(150)
name VARCHAR(50)
country VARCHAR(60)
country_code VARCHAR(10)
place_type VARCHAR(45)
url VARCHAR(1024)
bounding_box VARCHAR(500)
Indexes
tweets_has_terms
tweet_ID BIGINT(20)
term_ID INT(11)
comment VARCHAR(50)
Indexes
terms
term_ID INT(11)
term VARCHAR(100)
date TIMESTAMP
used TINYINT(1)
Indexes


retweets
tweet_ID BIGINT(20)
retweet_ID BIGINT(20)
user_ID BIGINT(20)
date TIMESTAMP
Indexes
scoring
score FLOAT
tweet_ID BIGINT(20)
taxonomy_ID INT(11)
category_ID INT(11)
score_manual FLOAT
Indexes
taxonomies
taxonomy_ID INT(11)
name VARCHAR(100)
description VARCHAR(100)
web_service VARCHAR(100)
Indexes
tweets
tweet_ID BIGINT(20)
text TEXT
html_text TEXT
date TIMESTAMP
reply_to_tweet_ID BIGINT(20)
lang VARCHAR(10)
retweet_count INT(11)
latitude DOUBLE
longitude DOUBLE
place_ID VARCHAR(16)
user_ID BIGINT(20)
Indexes
users
user_ID BIGINT(20)
screen_name VARCHAR(45)
name VARCHAR(45)
Indexes
registered_users
registered_user_ID BIGINT(20)
description VARCHAR(1000)
lang VARCHAR(50)
date TIMESTAMP
location VARCHAR(250)
url TEXT
verified TINYINT(1)
tweet_count INT(11)
favourites_count INT(11)
followers_count INT(11)
Indexes
taxonomies_has_categories
taxonomy_ID INT(11)
category_ID INT(11)
Indexes
categories
category_ID INT(11)
name VARCHAR(45)
parent_ID INT(11)
Indexes
urls
url_ID INT(11)
display_url VARCHAR(1024)
expanded_url VARCHAR(1024)
url VARCHAR(1024)
Indexes
tweet_translation
tweet_ID BIGINT(20)
language VARCHAR(2)
Indexes
tweets_has_hashtags
tweet_ID BIGINT(20)
hashtag_ID INT(11)
start_position INT(11)
end_position INT(11)
Indexes
users_mention_tweets
tweet_ID BIGINT(20)
user_ID BIGINT(20)
comment VARCHAR(50)
Indexes
tweets_has_urls
tweet_ID BIGINT(20)
url_ID INT(11)
start_position INT(11)
end_position INT(11)
Indexes
hashtags
hashtag_ID INT(11)
text VARCHAR(150)
Indexes


license_user
user_ID VARCHAR(45)
Indexes
social_country
country_ID INT(11)
name VARCHAR(70)
capital INT(11)
Indexes
1
1..*
social_community
community_ID INT(11)
name VARCHAR(70)
capital INT(11)
country INT(11)
Indexes
1
1..*
social_province
province_ID INT(11)
name VARCHAR(70)
capital INT(11)
community INT(11)
Indexes
synonyms_terms
term _ID VARCHAR(45)
joy FLOAT
fear FLOAT
sadness FLOAT
anger FLOAT
disgust FLOAT
surprise FLOAT
Indexes
revision
revision_ID INT(11)
message VARCHAR(140)
date TIMESTAMP
revisionexpertise FLOAT
editor_ID VARCHAR(45)
taxonomy_ID INT(11)
category_ID INT(11)
tweet_ID INT(11)
Indexes
social_city
city_ID INT(11)
name VARCHAR(70)
latitude FLOAT
longitude FLOAT
population INT(11)
province INT(11)
Indexes
terms_evaluation
term _ID VARCHAR(45)
joy FLOAT
fear FLOAT
sadness FLOAT
anger FLOAT
disgust FLOAT
surprise FLOAT
Indexes
favourite_entities_data
favourite_entities_data_ID INT(20)
entity_name VARCHAR(45)
entity_ID INT(11)
start_date TIMESTAMP
end_date TIMESTAMP
license_ID VARCHAR(45)
geographical_area_ID INT(11)
user_ID BIGINT(20)
closed BIT(1)
Indexes
info_web
tweet_ID BIGINT(20)
term _ID INT(10)
date TIMESTAMP
system_ID INT(10)
province_ID INT(11)
score FLOAT
scoring_name VARCHAR(45)
term _present TINYINT(4)
Indexes
provinces
province_ID INT(11)
name VARCHAR(45)
Indexes
aggregated_data

Funcionamiento de las aplicaciones

Ambas aplicaciones tiene como origen de sus datos los tuits de Twitter. La compañía ofrece ua API mediante el que, con restricciones de volumen y tiempo, se pueden recuperar esos mensajes de la nube y almacenarlos para su tratamiento posterior.

Es por lo tanto, un proceso offline, primero se descargan los tweet, se analizan, se extrae toda la información que se considera relevante —usuario, localización, fecha y hora, hashtags, términos, retweets, etc.— y se almacena en la base de datos. Con esa información ya "desmenuzada" se procede al filtrado y la exposición de resultados bien de forma gráfica, bien de forma tabular.

Es evidente que no se puede descargar "todo Twitter", los tweet recuperados son aquellos en los que se detecta cierta información clave relacionada con el área de aplicación de los resultados esperados. Por ejemplo, los que tienen que ver con votar o no a determinados partidos políticos.

Aun así, la cantidad de información recuperada y almacenada es considerable. Tanto que los propios responsables de las aplicaciones han tenido que idear formas de limitar la descarga para no hacer inviable su proceso.

En definitiva, el coste realmente reside en la adquisición de datos y su almacenamiento en sus correspondientes tablas. Por eso se aspira a que el usuario obtenga lo que quiere de forma instantánea; el pre proceso ya está hecho, el usuario solo filtra datos. Por algún proceso incremental se puede ir añadiendo información dando la ilusión de "tiempo real". Eso no quita para que, si añadimos tiempo innecesario al realizar las consultas sobre los datos analizados, la percepción del usuario sea de excesiva lentitud en obtener resultados.

Aquí es donde entra de lleno mi trabajo. Esas últimas consultas de usuario deben ser muy eficientes aún trabajando en un entorno con múltiples accesos simultáneamente y con filtros aleatoriamente complicados.

Objetivos y Metodología

La intención de la elaboración de éste trabajo es responder a las cuestiones:

¿Es realmente un sistema no relacional como MongoDB, más rápido para la obtención de información, en un entorno con un volumen inmenso de datos, que uno relacional como MySQL? O por el contrario podría mantenerse el desarrollo en un SGBDR, pero mejor optimizado, de forma que las consultas a los datos puedan mantener los tiempos exigidos por las aplicaciones web en las que van a aplicarse.

Para responder dichas cuestiones se realizará una monitorización del rendimiento de las consultas de datos en mi propia máquina, apuntando los tiempos necesarios para la obtención de los resultados en ambos sistemas, MySQL y MongoDB, pero antes de evaluar el rendimiento del sistema no relacional, he de evaluar si la optimización del SGBDR es la adecuada para su uso en las aplicaciones que lo precisan.

Tras comprobar la optimización de la base de datos MySQL y apuntar los tiempos de las consultas realizadas se probará una solución parcial, basada en los sistemas NoSQL, para la mejora del rendimiento de las consultas. Esto consistirá en desnormalizar la BD en una única tabla donde almacenar la información utilizada en las aplicaciones de GPLSI y comprobar si existe una mejora en la obtención de la información.

Una vez completada la evaluación de las soluciones en MySQL se procederá al traslado de la información a un entorno no relacional como MongoDB, en la misma máquina. Con la información necesaria ya incorporada, se realizarán las mismas pruebas de monitorización de las consultas en el nuevo esquema y se compararán con los resultados obtenidos en MySQL.

La elaboración de las conclusiones utilizará estos datos para responder a las cuestiones mencionadas anteriormente.

Debe remarcarse que trabajaré estrictamente en el escenario que se me ofrece, esto es, el estado de la base de datos en un momento concreto —cuando hicieron el *dump*, el respaldo— y sin modificaciones sustanciales del esquema. No es objetivo de mi trabajo todo el proceso de análisis que lleva a la población de la BD sino el rendimiento de las consultas posteriores que muestran los resultados de ese análisis.

Preparación del entorno

Las pruebas las realizaré en mi propio ordenador personal:

- Modelo: Portátil de 15,6” Asus A53SD
- Procesador: Intel Core i5 4 Cores a 2,50 GHz
- RAM: 8 GB
- Almacenamiento 1: SSD de 128 GB. 2 particiones:
    - Windows 10 Pro 64 bit
    - Linux Mint 17.3 XFCE Rosa 64 bit con el kernel 3.19.0-32-generic
- Almacenamiento 2: HDD de 500 GB para datos

La instalación de los sistemas y posteriormente las pruebas las realizaré en el SO Linux Mint, por motivos de eficiencia y espacio.

Los sistemas utilizados son:

- MySQL Community Server 5.7.9 y MySQL Workbench 6.3.5, ambos x64 bit
- MongoDB 3.2.7 x64 bit

Para información adicional sobre la instalación de los sistemas consultar Anexo 0: Instalación en Linux Mint

Importación de la BD

Una vez instalado MySQL procedo a la importación de la BD de Observer con la que trabajaré. Para la importación se me ha enviado un enlace para descargar un archivo comprimido con una copia (dump) de la Base de Datos en un momento concreto.

Para ello utilizaré la herramienta MySQL Workbench y su utilidad Data Import Wizard, que me permite importar un dump de una BD contenido en un archivo *‘socialobserver-15-10-16.sql’*. También crearé un esquema nuevo llamado *sobserver* donde almacenar la estructura y datos de la BD.

El tiempo de importación en Linux Mint ha sido de unas 2 horas y 28 min. Aprox.

Para información detallada paso a paso del proceso de importación ver Anexo 1: Importación en Linux Mint.

Consultas y primeras observaciones

Para tener una idea de los tiempos que se tarda en consultar los datos de la BD que se utilizarán en la aplicación voy a necesitar basarme en las consultas de que hayan utilizado para su sitio web, para ello me han enviado dos archivos:

- *info-socialobserver-consultas-usuario-.java*
- *info-socialobserver-consultas-gestion.java*

Con los comandos utilizados tanto en consultas de usuario, es decir las SELECT de los datos deseados, como consultas de gestión o administración de la base de datos, es decir eliminación e inserción y modificación de filas en la BD.

Para información detallada de las consultas extraídas del código Java consultar Anexo 2: Consultas de aplicación

Basándome en esto he creado dos scripts para consultas de usuario:

- *sql-lenta.sql*. Que consulta toda la BD para obtener las columnas deseadas
- *sql-vista.sql*. Que consulta la Vista (View) llamada *aggregated_data* con las columnas deseadas

Para ver el código de los script mencionados consultar Anexo 3: Consultas en MySQL

Y he corrido los scripts 1 vez para hacerme una idea de los tiempos que tardará cada una*. El tiempo de la consulta a la BD ha sido de aproximadamente 25 minutos, mientras que el de la Vista ha sido más o menos de 30 min. En ambos casos se han devuelto un total de 8443578 filas de resultados.

*Posteriormente, en el apartado Benchmarking las correré varias veces y haré una media del tiempo para tener una aproximación más realista.

Optimización de la BD

Para descartar un posible fallo en el diseño o la administración de la base de datos he de comprobar la optimización de la misma siguiendo los consejos de la documentación oficial[14]. Compruebo qué motores están utilizando en las tablas, si son dinámicas, comprimidas, etc; también los índices creados y utilizados en las consultas tanto a la BD como a la Vista.

Información detallada en Anexo 4: Optimización de la BD

Viendo la estructura de la BD puedo observar que está todo lo optimizada posible en cuanto al diseño.

Hay que mencionar que en la propia documentación de MySQL se indica que con cada versión el propio proceso interno del servidor intenta utilizar los valores más adecuados a nuestro trabajo.

Esto también afecta a las consultas, ya que las consultas de usuario que se utilizan en la aplicación de Sobserver son, de alguna manera, simples; nada más que SELECT FROM [WHERE] [ORDER BY]. El propio motor se encarga de buscar y utilizar el plan más óptimo para mostrar los datos de la consulta, en la documentación oficial, el apartado de optimización de consultas habla de operaciones aritméticas o complejas que no deberían utilizarse o que podrían sustituirse por operaciones más simples.

Por tanto, en lo que respecta al diseño y administración de ésta base de datos, se podría decir que está correctamente optimizada.

Monitorización de rendimiento (Benchmarking)

Viendo que la optimización a la BD es la correcta me dispongo a medir los tiempos medios de las consultas de usuario, que por ahora es el tema importante a comprobar, la obtención de los datos de la BD de la forma más rápida posible.

Voy a medir los tiempos de forma que quiero ver los valores más grandes posibles. Es importante limpiar los datos temporales que genera MySQL tras cada consulta para acelerar las repeticiones de la misma petición porque, para mi caso, no interesa esta aceleración, puesto que pretendo observar la diferencia de velocidad en los peores casos, cuando el propio servidor no acelere el proceso de obtención de la información.

Para medir el rendimiento de las consultas MySQL ofrece una serie de herramientas interesantes para realizar pruebas de carga en la BD como *MySQLSglap*, pero también pueden medirse con el comando *time* de Unix.

MySQLslap

La utilidad MySQLSlap sirve para simular consultas simples y mostrar el tiempo necesario para la obtención de los datos, así como cargas pesadas producidas por consultas de varios clientes simultáneamente o repeticiones de la misma consulta. Tiene 3 estados de funcionamiento[15]:

1. Crea/Usa el schema deseado, la tabla y, opcionalmente, añade datos necesarios para la prueba. (1 cliente)
2. Corre la prueba (1/Varios clientes)
3. Desconecta y borra la tabla si se ha especificado (1 cliente)

Opciones interesantes:

- --delimiter=”separador_usado”
- --create=”Sentencia_creacion/insercion_tabla/datos”
- --create-schema=”Esquema_donde_probar”
- --user
- --password
- --query=”Consulta/archivo_con_consultas”
- --concurrency=numero_clientes(entero)
- --iterations=numero_consultas_cada_cliente

Un ejemplo de ejecución para la BD sobserver, el script *sql-lenta.sql* con 1 cliente:

```
# mysqlslap -u mysql –password=mysql --create-schema
sobserver --delimiter ";" -q SQL-lenta.sql --no-drop
```

Comando time

El comando time de Unix cronometra el tiempo utilizado para ejecutar la orden que se le esté dando a la consola (Shell) además de otros tiempos. Mostrará 3 tiempos en total[16]:

- real: Tiempo transcurrido para la ejecución (cronómetro)
- user: Tiempo de CPU de Usuario, tiempo que tarda la CPU en realizar la acción
- sys: Tiempo de CPU de Sistema, llamadas al kernel en nombre del programa

El tiempo de CPU total (tiempo de usuario + tiempo de sistema) suele ser menor que el real, pero puede ser mayor si un programa se bifurca en hijos, ya que el tiempo de éstos se sumará al tiempo total sin contar en el real. En mi caso, me interesa observar el tiempo real de la ejecución del script, que es lo que tardará en obtener los datos y mostrarlos.

Sin embargo surge un problema con MySQL y es que no puedo obtener los datos sin imprimirlos, bien en consola o bien en un archivo externo. El mostrar los resultados por consola aumenta el tiempo de ejecución, ya que dicha tarea cuesta mucho trabajo. Para no mostrar los resultados de la consulta utilizaré una escritura a un archivo null, de forma que no escribirá nada realmente y no mostrará por consola las filas obtenidas.

Un ejemplo de ejecución para la BD sobserver, el script *sql-lenta.sql*:

```
# time mysql -u mysql –password=mysql < SQL-lenta.sql >
/dev/null
```

Primeras pruebas

Para estas primeras pruebas haré una comprobación del tiempo que le cuesta al sistema realizar la consulta desde mi máquina recién arrancada.

Utilizaré los scripts *sql-lenta.sql* y *sql-vista.sql* para la realización de las consultas.

Después de cada consulta reiniciaré el servicio de MySQL para borrar los datos de caché almacenados en disco, como he mencionado anteriormente.

Éste es el gráfico obtenido de los tiempos medios de consultar la Vista y la BD:

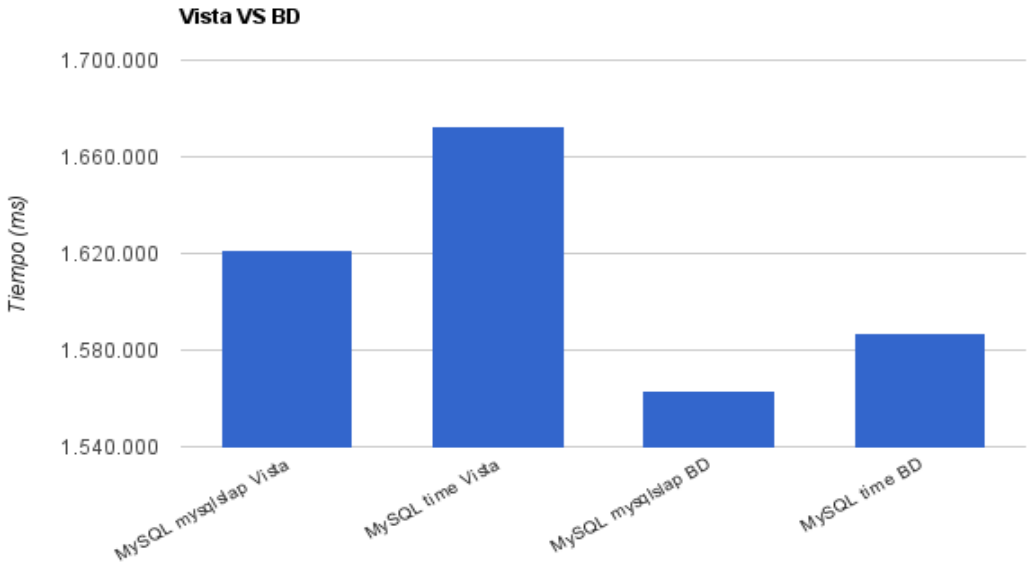


Gráfico de tiempos 1. Vista Vs BD completa

Obtenemos unos valores muy aproximados a la toma de contacto, unos

- 27 min aprox. (1.621.472 ms de media) para el comando mysqlslap hacia la Vista
- 28 min aprox. (1.672.299 ms de media) para el comando time hacia la Vista
- 26 min aprox. (1.563.317 ms de media) para el comando mysqlslap hacia la BD
- 27 min aprox. (1.586.889 ms de media) para el comando time hacia la BD

La diferencia de tiempo entre *mysqlslap* y *time* posiblemente se deba a que mysqlslap se ciñe expresamente al tiempo que se tarda en realizar la consulta al servidor, sin tener en cuenta la ralentización debida a la carga del sistema en ese momento y, también porque time técnicamente está "escribiendo la salida" a un archivo (en realidad nulo), mientras que el primero no mide dicha carga.

Para ver los valores en detalle consultar Anexo 5: Tablas de tiempos

Filtrado de las consultas

Ahora probemos a consultar la BD filtrando los resultados. Para estas pruebas he elegido 3 filtros, cada uno guardado en diferentes scripts .sql:

- term = "Javier Echenique". Que devolverá 43 filas

- term = “#Alicante”. Que devolverá 340.007 filas
- term like “%pp%” or term like “%ppopular%” or term like “%psoe%” or term like “%ciuda%” or term like “%compromis%” or term like “%UpyD%”. Que devolverá 3.852.943 filas

Para más detalles sobre las consultas Anexo 3: Consultas de MySQL

Estos filtros me ayudarán a observar cómo se comportan las consultas con parámetros típicos de las aplicaciones web en las que se utiliza. También, de forma intencionada, se han utilizado parámetros que devuelvan un número creciente de filas para observar cómo afecta el total de registros obtenidos a los tiempos de carga.

Las pruebas se realizan de la misma manera que en el caso anterior, únicamente cambian los scripts a ejecutar. Un ejemplo de ejecución sería:

```
# time mysql -u mysql –password=mysql < SQL-lenta-where.sql > /dev/null
```

```
# mysqlslap -u mysql –password=mysql --create-schema sobserver --delimiter ";" -q SQL-lenta-where.sql --no-drop
```

Utilizaré los scripts *sql-lenta-where.sql*, *sql-lenta-where-alicante.sql*, *sql-lenta-where-compuesto.sql*, *sql-vista.sql*, *sql-vista-where.sql* y *sql-vista-where-compuesto.sql* para la realización de las consultas.

Estos son los gráficos obtenidos de los tiempos medios de consultar la BD y la Vista con los diferentes filtros:

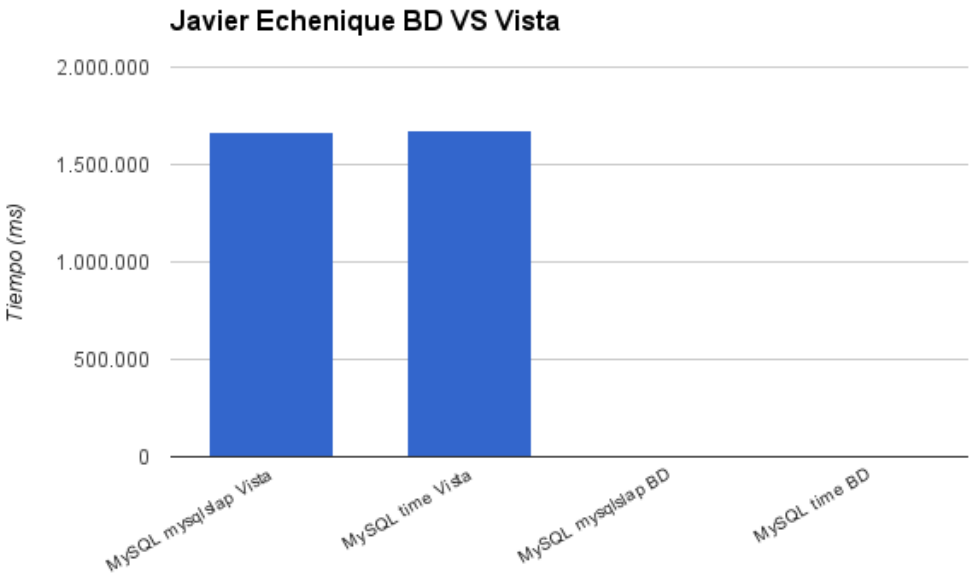


Gráfico de tiempos 2. Javier Echenique Vista Vs BD

Obtenemos los valores:

- 27 min aprox. (1.666.916 ms de media) para el comando mysqlslap hacia la Vista
- 28 min aprox. (1.679.298 ms de media) para el comando time hacia la Vista
- 94 ms aprox. para el comando mysqlslap hacia la BD
- 109 ms de media para el comando time hacia la BD

Este resultado es excepcionalmente anómalo. Cabe pensar que la vista ha de dar tiempos muy parecidos a la consulta directa, la que llamo "BD". Es cierto que MySQL arrastra desde hace mucho tiempo problemas de rendimiento en la ejecución de las vistas[23] pero no parece este un escenario en el que deban darse tales diferencias de tiempos.

Revisando la definición original de la vista "aggregated_data" se han detectado errores en su definición, concretamente el uso del modificador DISTINCT en la consulta base. Redefinida la vista, los tiempos se hacen "normales", con valores que uno podría esperar.

Información detallada en Anexo 4: Optimización de la BD

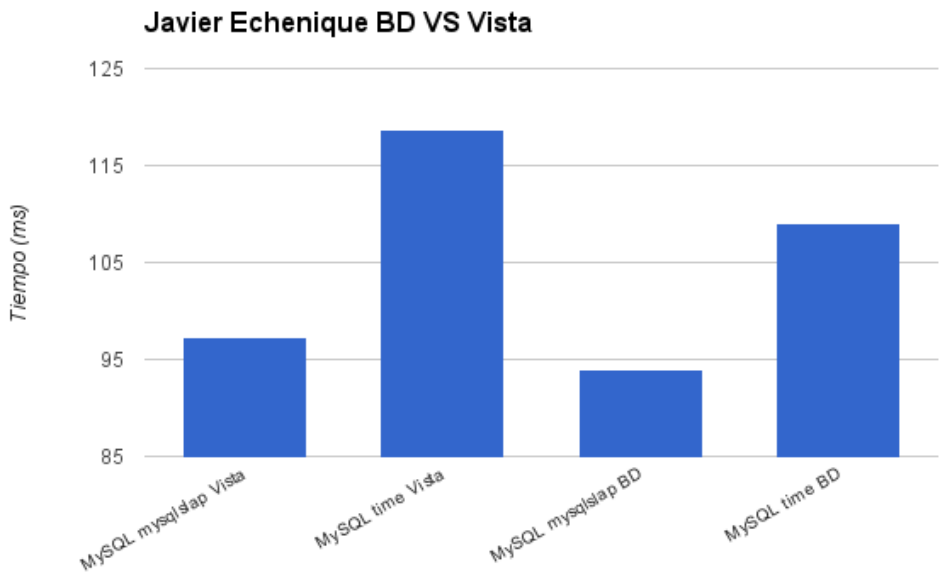


Gráfico de tiempos 3. Javier Echenique Vista (redefinida) Vs BD

Obtenemos los valores:

- 97 ms de media para el comando mysqlslap hacia la Vista
- 119 ms de media para el comando time hacia la Vista
- 94 ms aprox. para el comando mysqlslap hacia la BD
- 109 ms de media para el comando time hacia la BD

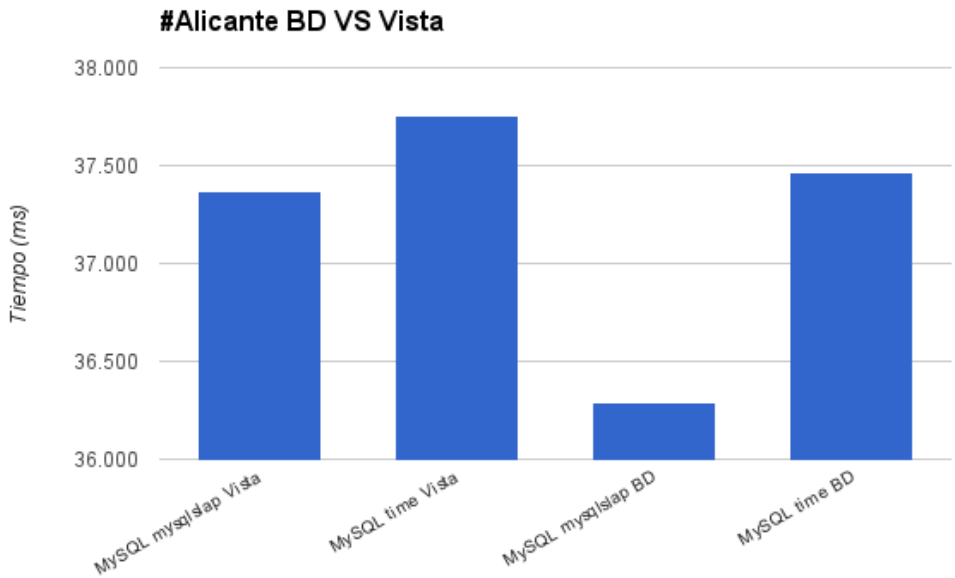


Gráfico de tiempos 4. Alicante Vista Vs BD

Obtenemos los valores:

- 37 seg aprox. (37.365 ms de media) para el comando mysqlslap hacia la Vista
- 38 seg aprox. (37.756 ms de media) para el comando time hacia la Vista
- 36 seg aprox. (36.292 ms de media) para el comando mysqlslap hacia la BD
- 37 seg aprox. (37.462 ms de media) para el comando time hacia la BD

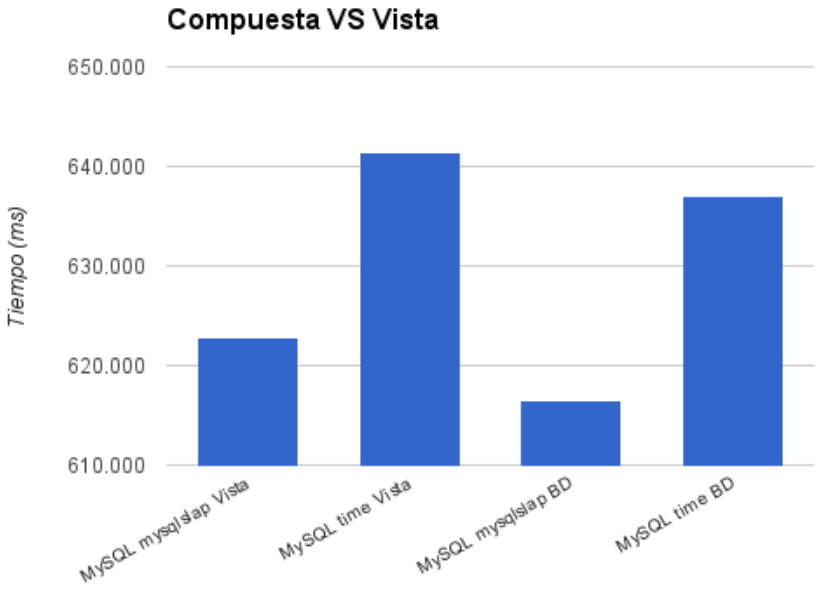


Gráfico de tiempos 5. Compuesta Vista Vs BD

.

Obtenemos los valores:

- 10 min 23 seg aprox. (622.846 ms de media) para el comando mysqlslap hacia la Vista
- 10 min 41 seg aprox. (641.395 ms de media) para el comando time hacia la Vista
- 10 min 17 seg aprox. (616.578 ms de media) para el comando mysqlslap hacia la BD
- 10 min 37 seg aprox. (636.972 ms de media) para el comando time hacia la BD

Se puede observar que, cuantas menos filas devuelva el resultado, la consulta será más rápida, tal es así que, cuando utilizamos el filtro de menor número de registros (Javier Echenique), es prácticamente inmediata en ambos casos.

Esto ocurre porque cuando se utilizan filtros en las consultas, tanto la BD como la Vista, consulta una tabla temporal en disco donde ha almacenado la información más utilizada en las consultas para acelerar la obtención de los resultados.

Pero puestos a ser exigentes sí es cierto que en estos ejemplos las consultas directas a la BD son ligeramente más rápidas que usando la Vista.

Desnormalización de la BD en MySQL

Los sistemas NoSQL, en su mayoría, publicitan como una de sus características más deseadas el carecer de esquema de base de datos. En principio, esto da total libertad al diseñador (más bien lo elimina) y al programador, dejando todos los aspectos de organización y recuperación de los datos en manos del propio servidor de base de datos. Obviamente, esto tiene sus aplicaciones, pero no ha de tomarse como la solución a todos los sistemas de información posibles, más bien al contrario, y es que NoSQL tiene unos usos concretos.

No obstante, la minería de datos es un ejemplo de previo de desnormalización de bases de datos relacionales buscando manejar grandes volúmenes de datos para análisis más sofisticados que extraigan información a partir de los datos almacenados. En realidad, una aproximación cercana a NoSQL o directamente un punto de partida. Me planteo si una adaptación simple del esquema como es la desnormalización total del esquema de la base de datos, o lo que es lo mismo, trasladar a una única tabla todos los datos de la BD, daría un rendimiento mejor, peor o similar que otras soluciones.

A ésta tabla la he denominado *tabla_parche* ya que considero que sería un apaño rápido frente a un problema que podría o no resolverse, no lo sabré hasta la ejecución de las consultas. Además podría un cambio de pensamiento en la Administración de la BD que podría o no afectar a la integridad de los datos y a la seguridad del sistema.

Para la creación de la misma he utilizado un script *create.sql* que utilizará el comando CREATE TABLE... SELECT ... para obtener las columnas necesarias de la Vista de la BD y exportarlas a una tabla nueva.

Me interesa conocer también el tiempo necesario para realizar esta operación, podría ser posible que para actualizar los datos en esta tabla los desarrolladores decidan crearla de nuevo cada cierto tiempo en lugar de insertar los valores nuevos cada vez que se introduzcan en el esquema original. Utilizaré para ello el comando time, principalmente porque no trato de medir el tiempo de la obtención de los datos, sino el de la creación de la tabla que contendrá los mismos, por lo que se sumarán el tiempo de consulta y el de creación.

El tiempo necesario ha sido de 33 min. y 43 seg. y la tabla contiene, efectivamente 8443578 filas, por lo que se puede afirmar que contiene los mismos registros que la Vista utilizada.

Para información detallada del script y los tiempos consultar Anexo 6: Desnormalización

Benchmarking en la tabla desnormalizada

Del mismo modo que hice para el caso anterior, haré una comprobación del tiempo que le cuesta al sistema realizar la consulta desde mi máquina recién arrancada.

Utilizaré el script *sql-parche.sql* para la realización de las consultas.

Para detalle de la consulta a la tabla parche consultar Anexo 3: Consultas de MySQL

Éste es el gráfico obtenido de los tiempos medios de consultar la Tabla 'parche' contra la BD, que fue la que mejores tiempos tenía en pruebas anteriores:

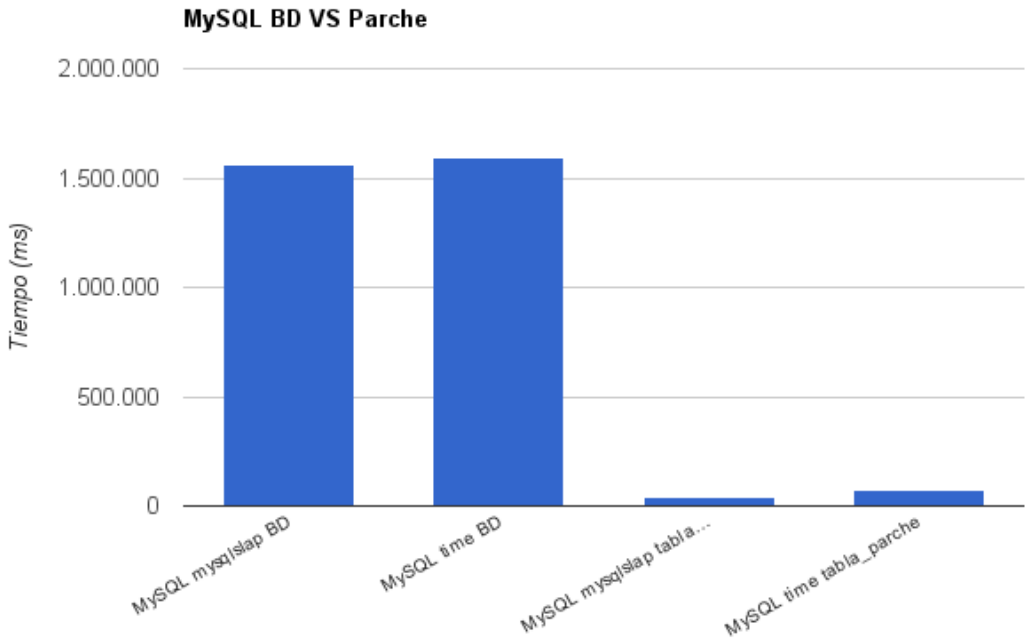


Gráfico de tiempos 6: BD VS Parche

Obtenemos los siguientes valores:

- 26 min aprox. (1.563.317 ms de media) para el comando mysqlslap hacia la BD
- 26 min 35 seg aprox. (1.595.280 ms de media) para el comando time hacia la BD
- 42 seg aprox. (41.968 ms de media) para el comando mysqlslap hacia la tabla parche
- 1 min y 14 seg aprox. (74.588 ms de media) para el comando time hacia la tabla parche

Teniendo en cuenta que en ambos casos se han devuelto 8443578 filas con los mismos datos. Se puede observar una diferencia más que considerable en los tiempos conseguidos al utilizar una tabla desnormalizada.

Para información más detallada consultar Anexo 5: Tablas de tiempos

Filtrado de las consultas

Ahora realizaré las consultas con filtros en la tabla parche. Ejecutaré las pruebas de la misma manera que en la BD, pero cambiaré los scripts para que consulten la tabla 'parche'.

Utilizaré los scripts *sql-parche-where.sql, sql-parche-where-alicante.sql* y *sql-parche-where-compuesto.sql* para la realización de las consultas.

- La primera filtra sobre un término de poca utilización en la base de y develve 43 filas.
- La segunda, también sobre un único término, pero este con mayor presentica en la BD, que devuelve unas 340007 filas.
- La última pretende simular una consulta habitual en los programas de demostración de las aplicaciones GPLSI, que filtra por diversos partidos políticos españoles y devolverá un total de 3.852.943 filas.

Para ver el código de las consultas sobre las consultas Anexo 3: Consultas de MySQL

Estos son los gráficos obtenidos de los tiempos medios de consultar la BD y la tabla Parche con los diferentes filtros:

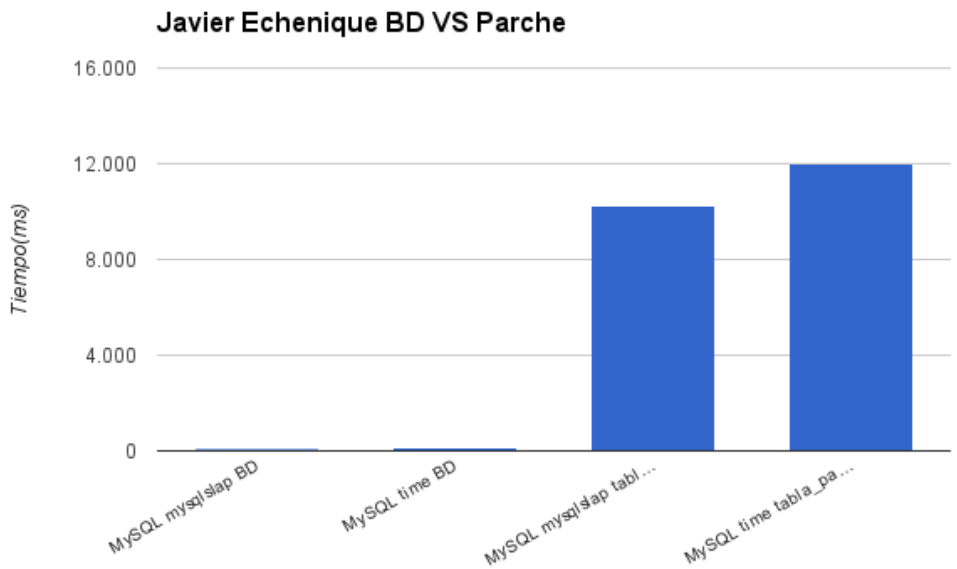


Gráfico de tiempos 7. Javier Echenique BD VS Parche

Obtenemos los valores

- 94 ms aprox. para el comando mysqlslap hacia la BD
- 109 ms de media para el comando time hacia la BD
- 11 seg aprox. (10.281 ms de media) para el comando mysqlslap hacia la tabla Parche
- 12 seg aprox. (12.023 ms de media) para el comando time hacia la tabla Parche

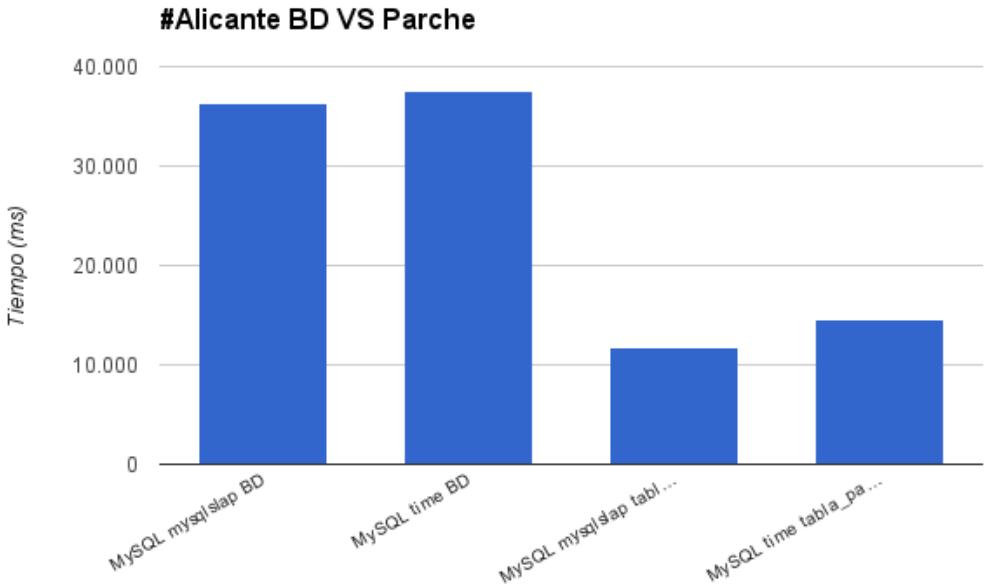


Gráfico de tiempos 8. Alicante BD VS Parche

Obtenemos los valores

- 36 seg aprox.(36.292 ms de media) para el comando mysqlslap hacia la BD
- 37 seg aprox. (37.462 ms de media) para el comando time hacia la BD
- 12 seg aprox. (11.782 ms de media) para el comando mysqlslap hacia la tabla Parche
- 15 seg aprox. (14.615 ms de media) para el comando time hacia la tabla Parche

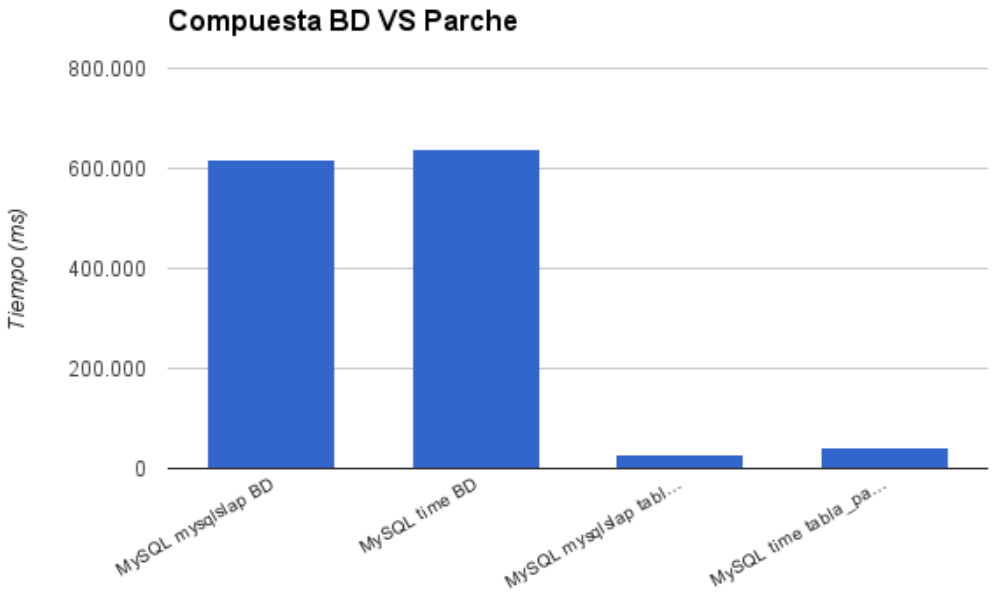


Gráfico de tiempos 9. Compuesta BD VS Parche

Obtenemos los valores:

- 10 min 16 seg aprox. (616.578 ms de media) para el comando mysqlslap hacia la BD
- 10 min 36 seg aprox. (636.972 ms de media) para el comando time hacia la BD
- 30 seg aprox. (29.749 ms de media) para el comando mysqlslap hacia la tabla Parche
- 43 seg aprox. (43.373 ms de media) para el comando time hacia la tabla Parche

Parece que la BD va perdiendo en rendimiento contra la tabla parche conforme aumenta el número de filas devueltas. Sí es cierto que, para el caso de las 43 filas, la tabla temporal donde consulta la BD sigue siendo más rápida, con unos 10 seg. de diferencia frente a la tabla desnormalizada.

Traslado a MongoDB

Ahora que he comprobado los tiempos de consulta en MySQL para los casos en los que se consulte la BD, una Vista o incluso se opte por desnormalizar los datos requeridos en una tabla aparte, me dispongo a comprobar si uno de los sistemas NoSQL más utilizados en la actualidad, MongoDB, es realmente más rápido que cualquier optimización/apaño que utilicemos en nuestro sistema original MySQL.

Puesto que MongoDB no es relacional y no utiliza JOINs en sus consultas, no veo necesario (o posible) exportar toda la BD a una serie de colecciones (equivalentes a tablas en MySQL) del mismo. Es más, puesto que ha quedado comprobado que el uso de la misma desnormalización de estos motores en el propio MySQL supone una ganancia considerable del rendimiento para las consultas, considero que el comparar los tiempos de la tabla parche en MySQL frente a los tiempos que podría tardar una tabla equivalente en MongoDB me respondería mejor a la pregunta '¿Es necesario cambiar el servidor de BD para obtener un rendimiento óptimo?', ya que la tabla parche supone una solución de buenos resultados sin necesidad de ello.

Exportación de los datos

Para la exportación de los datos utilizaré el entorno MySQLWorkbench, que me ofrece la utilidad Table Data Export Wizard para exportar una consulta a un archivo tipo .csv o .json. En este caso utilizaré el formato .json, que es con el que trabaja MongoDB.

Al finalizar indica 8443578 registros, que coincide con las filas que contiene la tabla, ha tardado unos 37 minutos aprox. y el archivo ocupa unos 4,5 GB.

Importación a MongoDB (mongoimport)

Para la importación a MongoDB utilizaré el comando *mongoimport*, que permite insertar contenido de archivos JSON, CSV o TSV de forma rápida, aunque no se recomienda su utilización junto con *mongoexport* para crear/importar backups de una colección[17].

Algunas opciones interesantes:

- --db <database>, -d <database>: Selecciona la BD donde importar, si no existe la crea
- --collection <collection>, -c <collection>: Selecciona la colección (tabla) donde importar. si no existe la crea

- --stopOnError: Fuerza la parada del proceso en caso de error en algún documento
- --jsonArray: Acepta el uso de un Array de JSONs, así como si en alguno de los documentos contenidos existe un array. Limitado a 16 MB por documento.
- --file: Especifica el nombre del archivo que contiene los documentos a insertar

Un ejemplo de comando sería:

```
# mongoimport --stopOnError --jsonArray --db sobserver --
collection aggregated_data --file TablaParcheJson.json
```

Sin embargo al intentar realizar la importación. Ésta se para a los 2.0 GB de datos importados y no introduce ningún dato en la colección el archivo JSON en pedazos más pequeños de 2.0GB utilizando el comando *split* de Unix.

Una vez obtenidos los archivos, hay que formatear los archivos para que estén en un formato correcto para JSON, eso significa añadirle '[' o ']' al principio y/o al final, según convenga y quitar las ',' finales de los primeros archivos.

Tras corregir esta dificultad ya me deja importar cada uno de los archivos anteriores. El comando de importación queda así:

```
# mongoimport --stopOnError --jsonArray --db sobserver --
collection aggregated_data --file sub-parcheaX.json
```

*Siendo X la letra que corresponde a cada archivo

Ha tardado alrededor de 8 minutos en importar todos los documentos. Una vez importados, para comprobar que coinciden el total de documentos importados a la colección con el número de registros exportados por MySQLWorkbench utilizo un count de MongoDB:

```
# mongo sobserver --eval "db.runCommand({count:
'aggregated_data'})"
```

Coincide el número de documentos con los registros exportados con MySQLWorkbench y las filas de la tabla parche, 8443578.

Para información detallada del proceso de importación consultar Anexo 7: Importación a MongoDB

Benchmarking en MongoDB

Siguiendo la documentación oficial de MongoDB[18], para medir el rendimiento de las consultas se puede utilizar el comando *.explain("executionStats")*, del cual me interesa el tiempo mostrado en la clave *executionTimeMillis*, que me dice los ms necesarios para ejecutar la consulta[19]. Además de éste, usaré el comando time, igual que en los casos anteriores para obtener los tiempos reales de la ejecución de las consultas.

Una cosa a tener en cuenta es que MongoDB no necesita obligatoriamente devolver una columna de datos, por lo que puedo mostrar/escribir únicamente la información que precise, en el caso de *explain()*, el *executionTimeMillis*, y en el caso de time no me es necesario imprimir nada, por lo que no preciso de escribir en un archivo nulo,

MongoDB permite la ejecución de scripts desde consola de Unix, al igual que MySQL, pero se debe usar un archivo .js escrito con lenguaje JavaScript válido.

Del mismo modo que hice anteriormente, haré una comprobación del tiempo que le cuesta al sistema realizar la consulta desde mi máquina recién arrancada.

Utilizaré los scripts *script-SQL-parche.js* y *script-stats.js* para la realización de las consultas. Para la ejecución de los scripts los comandos quedarían:

```
# mongo sobserver script-stats.js
```

```
# time mongo sobserver script-SQL-Parche.js
```

Para detalle de las consultas a MongoDB consultar Anexo 8: Consultas de MongoDB

MongoDB tiene la capacidad de mantener archivos temporales en memoria virtual incluso después de reiniciar el servicio (demonio) *mongod*. Para mantener la misma política que con MySQL de no aprovechar la optimización propia del sistema con archivos temporales debería tener que reiniciar el sistema por completo, pero para evitar tener que reiniciar el ordenador en cada consulta, puedo utilizar un programa compatible con Linux llamado Bleachbit, que me permite limpiar la memoria virtual de procesos sin utilizar si se ejecuta con permisos de superusuario.

Después de cada consulta detendré el proceso *mongod*, ejecutaré Bleachbit con opción de limpiar la memoria y, por último iniciaré de nuevo el servidor de MongoDB. De ésta manera me aseguro que no se utilicen archivos temporales.

Éste es el gráfico obtenido de los tiempos medios de consultar la Tabla parche contra MongoDB. No es necesario compararlo con la consulta a toda la BD, ya que hemos visto que, para consultas sin filtro, existe una diferencia considerable entre la tabla parche y la BD:

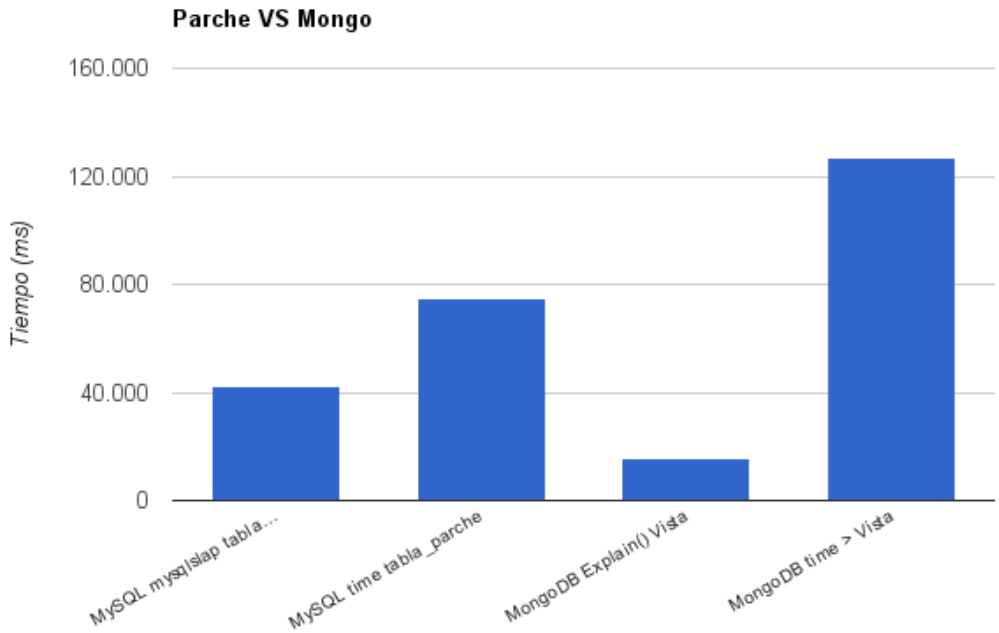


Gráfico de tiempos 10. Tabla parche VS MongoDB

Obtenemos los siguientes valores:

- 42 seg aprox. (41.968 ms de media) para el comando mysqlslap hacia la tabla parche
- 1 min y 14 seg aprox. (74.588 ms de media) para el comando time hacia la tabla parche
- 16 seg aprox. (15.915 ms de media) para el comando explain() en MongoDB
- 2 min y 7 seg aprox. (126.582 ms de media) para el comando time en MongoDB

Para información más detallada consultar Anexo 5: Tablas de tiempos

Es interesante ver que, como en MySQL, los tiempos representados por la herramienta *explain()* propia de MongoDB son menores que los de *time*, pero más interesante aún es que el tiempo obtenido por *explain()* es menor que el de *mysqlslap*, mientras que el *time* de MongoDB es mayor que en MySQL.

Esto generaría una duda ¿A qué valor hago caso? Personalmente, me decanto por el valor de *time*, que muestra el tiempo que ha llevado la ejecución de la consulta en el sistema, teniendo en cuenta las cargas que tenga en ese momento.

Filtrado de consultas

Ahora mediré los tiempos que le lleva a MongoDB realizar una consulta filtrada para después compararlos con los de la tabla parche y la BD completa.

Utilizaré los scripts *script-SQL-parche-where.js, script-stats-where.js, script-SQL-parche-where-compuesto.js y script-stats-where-compuesto.js,* para la realización de las consultas. Hay que tener en cuenta que en MongoDB hay que usar una escritura compatible con JavaScript para filtrar los resultados.

Para ver código de los scripts consultar Anexo 8: Consultas en MongoDB

Estos son los gráficos obtenidos de los tiempos medios de consultar la BD completa, la Tabla parche y MongoDB con las consultas filtradas:

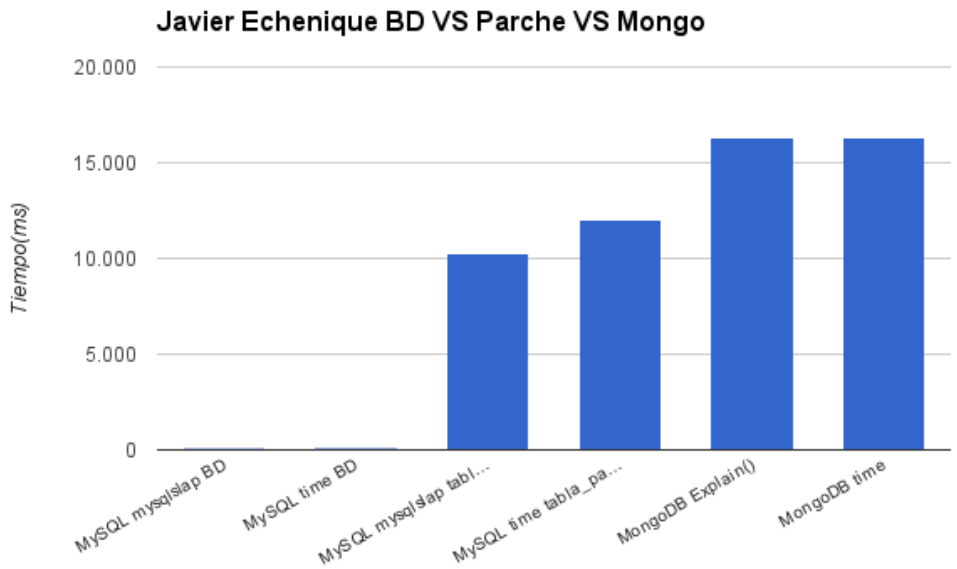


Gráfico de tiempos 11. Javier Echenique BD VS Tabla parche VS MongoDB

Obtenemos los siguientes valores:

- 94 ms de media para el comando mysqlslap hacia la BD completa
- 109 ms de media para el comando time hacia la BD completa
- 10 seg aprox. (10.281 ms de media) para el comando mysqlslap hacia la tabla parche
- 12 seg aprox. (12.023 ms de media) para el comando time hacia la tabla parche
- 16 seg aprox. (16.337 ms de media) para el comando explain() en MongoDB
- 16 seg aprox. (16.347 ms de media) para el comando time en MongoDB

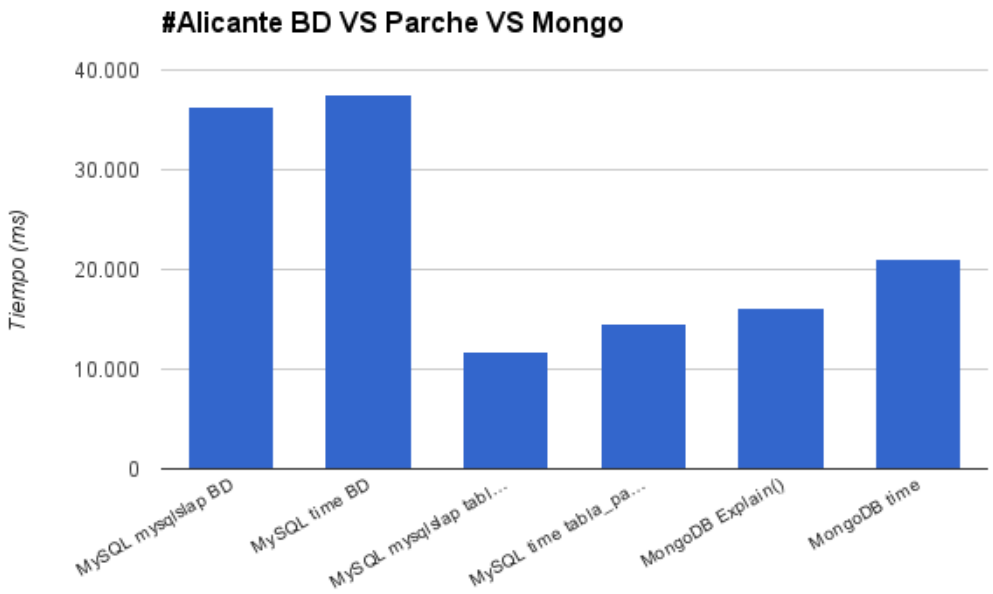


Gráfico de tiempos 12. Alicante BD VS Tabla parche VS MongoDB

Obtenemos los siguientes valores:

- 36 seg aprox. (36.292 ms de media) para el comando mysqlslap hacia la BD completa
- 37 seg aprox. (37.462 ms de media) para el comando time hacia la BD completa
- 12 seg aprox. (11.782 ms de media) para el comando mysqlslap hacia la tabla parche
- 15 seg aprox. (14.615 ms de media) para el comando time hacia la tabla parche
- 16 seg aprox. (16.080 ms de media) para el comando explain() en MongoDB
- 20 seg aprox. (20.968 ms de media) para el comando time en MongoDB

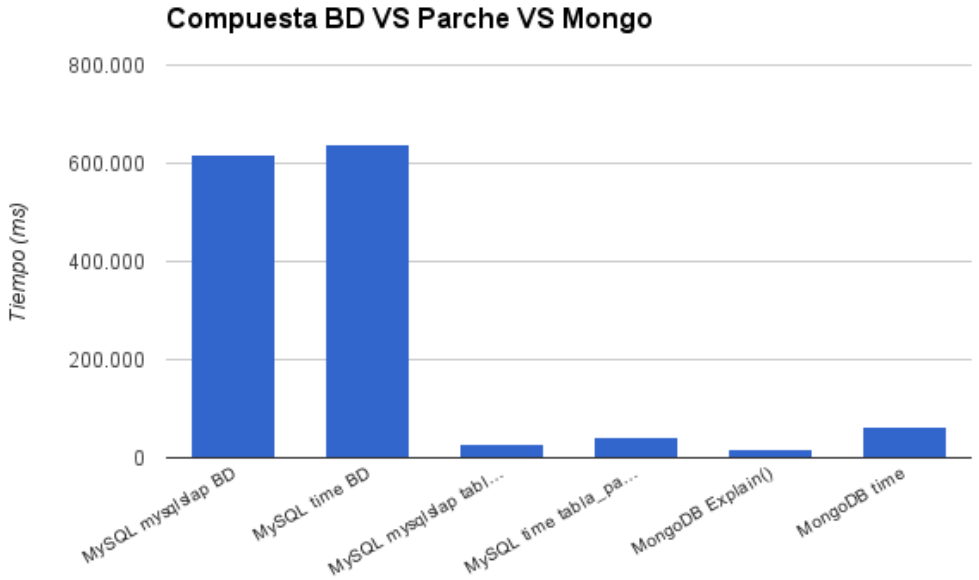


Gráfico de tiempos 13. Compuesta BD VS Tabla parche VS MongoDB

Obtenemos los siguientes valores:

- 10 min 17 seg aprox. (616.578 ms de media) para el comando mysqlslap hacia la BD completa
- 10 min 37 seg aprox. (636.972 ms de media) para el comando time hacia la BD completa
- 30 seg aprox. (29.749 ms de media) para el comando mysqlslap hacia la tabla parche
- 43 seg aprox. (43.373 ms de media) para el comando time hacia la tabla parche
- 16 seg aprox. (16.906 ms de media) para el comando explain() en MongoDB
- 1 min 2 seg aprox. (62.295 ms de media) para el comando time en MongoDB

Para información más detallada consultar Anexo 5: Tablas de tiempos

Al igual que cuando se comparaban la tabla parche y la BD completa, cuando se devuelven pocos registros es la tabla temporal de la BD la que ofrece los resultados más rápidos, sin embargo en los siguientes filtros pasa a ser la consulta más lenta de las 3. Respecto a la tabla parche frente a MongoDB ocurre lo mismo que cuando se realiza una consulta sin filtrar, que los tiempos de MongoDB son menores si hacemos caso del comando *explain(),* pero son peores con el comando *time*. Reitero mi opinión de que me parece más fiable o correcta la interpretación de *time*, por lo que, en mi opinión, la tabla parche sigue teniendo mejores tiempos de carga.

Pero hay un apartado que todavía no he comprobado y es la posible optimización tanto de la tabla parche en MySQL como de la colección en MongoDB utilizando índices en las columnas necesarias. Para comprobarlo crearé índices de las columnas que los utilizaban en las consultas a la BD y a la Vista, un total de 15 índices.

Para información detallada de los índices utilizados en la Vista y en la BD consultar Anexo 4: Optimización de la BD

Pero una cosa a tener en cuenta con el uso de índices en las bases de datos es que pueden ralentizar tareas de administración como el borrado o inserción de un número masivo de datos, por lo que debo tener en cuenta una comparación de los tiempos que generan consultas de administración tanto antes de crear los índices como después.

Benchmarking de consultas de Administración

Puesto que el uso de índices podría o no ralentizar operaciones de administración de la BD me dispongo a medir los tiempos actuales de eliminar/añadir datos en ambos sistemas.

He considerado suficiente la eliminación e inserción de 10.000 filas de la tabla parche en MySQL y de la colección de MongoDB. Para ello lo que haré será primero hacer una consulta ordenada de los primeros 10.000 registros de cada sistema para después exportarlos a un archivo válido para su inserción.

Para medir el tiempo utilizado utilizaré el comando *time*, las otras órdenes no me permiten medir estas operaciones.

MySQL

En MySQL he generado un script llamado *sql-parche-export.sql* que ejecutaré en el MySQL Workbench. Dicha herramienta me permite guardar los resultados de una consulta a varios tipos de archivo, entre ellos un archivo .sql con comandos INSERT INTO en cada registro para facilitar su inserción.

Una vez obtenido el archivo puedo proceder al borrado de las 10.000 primeras filas ordenadas; para ello utilizo el script *borrar.sql* en el siguiente comando:

```
# time mysql -u mysql --password=mysql sobserver < borrar.sql
```

La inserción del archivo en MySQL se podría realizar de la siguiente manera:

```
# time mysql -u mysql --password=mysql sobserver < insercion-
nfilas.sql
```

Para más detalle del proceso de exportación y de los scripts .sql consultar Anexo 9: Consultas de Administración

MongoDB

En MongoDB es más simple, ya que puedo utilizar el comando *mongoimport* igual que antes para insertar los documentos contenidos en un archivo JSON y para generar este archivo puedo utilizar un script llamado *script-salida.js* que imprimirá los 10.000 primeros

documentos ordenados de la misma manera que en MySQL a un archivo de texto. La orden de la exportación queda:

```
# time mongo sobserver --quiet script-salida.js >
../insercion-mongo-nfilas.json
```

*Nota: La ruta del archivo debe ser absoluta, aquí se ha simplificado

Una vez obtenido el archivo, al igual que en MySQL procedo a borrar primero los 10.000 primeros documentos y después insertarlos con *mongoimport*. Los comandos quedarían:

```
# time mongo sobserver --quiet script-borrar.js
```

```
# time mongoimport --stopOnError --jsonArray --db sobserver -
-collection aggregated_data --file insercion-mongo-
nfilas.json
```

Para más detalle del proceso de exportación y de los scripts .sql consultar Anexo 9: Consultas de Administración

Después de cada consulta reiniciaré el servicio *mysql* y detendré el proceso *mongod*, ejecutaré Bleachbit con opción de limpiar la memoria y, por último iniciaré de nuevo el servidor de MongoDB. De ésta manera me aseguro que no se utilicen archivos temporales.

Éste es el gráfico obtenido de los tiempos medios de consultar la Tabla parche contra MongoDB:

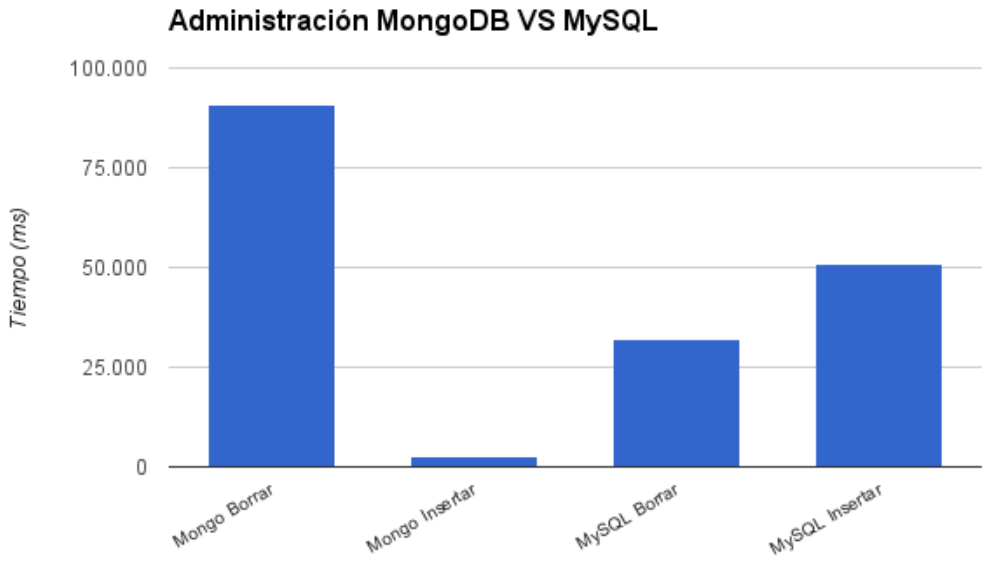


Gráfico de tiempos 14. Administración MongoDB VS MySQL

Obtenemos los siguientes valores:

- 1 min y 30 seg aprox. (90.792 ms de media) para borrar en MongoDB
- 3 seg aprox. (2.827 ms de media) para insertar en MongoDB
- 32 seg aprox. (32.064 ms de media) para borrar en MySQL
- 50 seg aprox. (50.671 ms de media) para insertar en MySQL

Nótese que la inserción en MongoDB es tremendamente rápida en comparación con MySQL, sin embargo el borrado de filas es más lento

Para información más detallada consultar Anexo 5: Tablas de tiempos

Índices de datos

Ahora que tengo los tiempos de administración puedo comenzar a crear los índices necesarios en cada sistema. El uso de éstos índices podría ayudar a mejorar los tiempos de consulta, sobre todo en aquellas con filtros específicos, ya que por defecto no se usan índices en consultas generales.

En el caso de MySQL utilizaré el script *indices.sql* para la creación de los 15 índices necesarios. El comando de Shell sería:

```
# time mysql -u mysql --password=mysql sobserver < 
indices.sql
```

Para ver el código del script consultar Anexo 10: Índices

A la hora de realizar las consultas, como se ha mencionado antes, a no ser que se indique un filtro, no se utilizarán estos índices en la consulta general a la tabla parche, por lo que debo usar consultas filtradas.

Utilizaré los scripts *sql-parche-where.sql, sql-parche-where-alicante.sql* y *sql-parche-where-compuesto.sql* para la realización de las consultas.

Con la consulta compuesta pasa una cosa, y es que por defecto o usa el índice que han usado las anteriores candidatas. Esto ocurre porque con tal cantidad de registros a devolver, el índice se hace pesado y el propio motor de MySQL detecta que es mejor no utilizarlo. Por tanto, utilizaré los resultados que dio anteriormente dicha consulta cuando no estaba el índice presente.

En MongoDB utilizaré para la creación del índice *script-indices.js*. El comando queda:

```
# time mongo sobserver script-indices.js
```

Para información detallada y código de los scripts consultar Anexo 10: Índices

Utilizaré los scripts *script-sql-parche-where.js, script-sql-parche-where-alicante.js* y *script-sql-parche-where-compuesto.js* para la realización de las consultas.

Estos son los gráficos obtenidos de realizar las consultas filtradas en la BD completa frente a la tabla Parche y MongoDB, ahora con Índices:

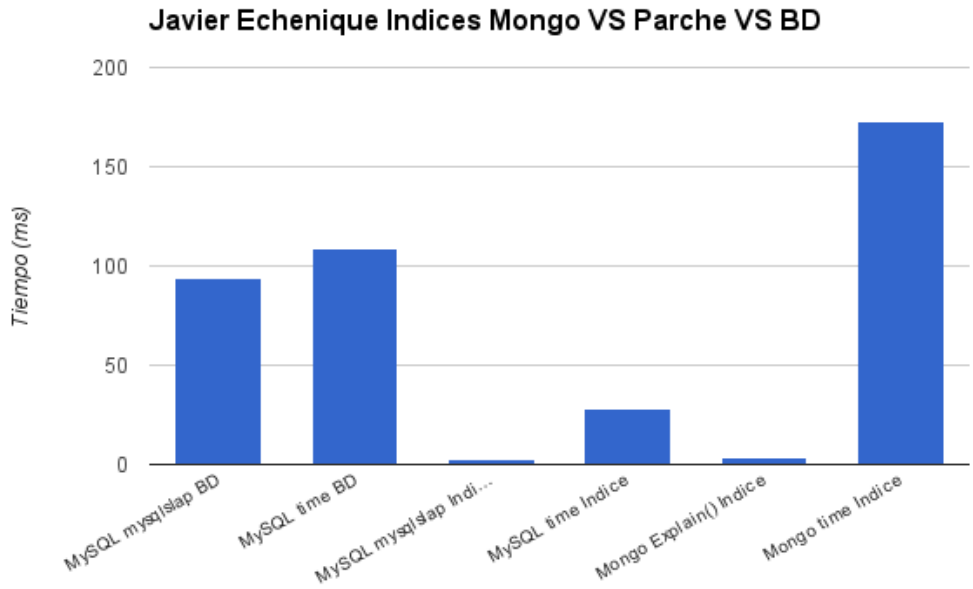


Gráfico de tiempos 15. Javier Echenique Índices BD VS Tabla parche VS MongoDB

Obtenemos los siguientes valores:

- 94 ms de media para el comando mysqlslap hacia la BD completa
- 109 ms de media para el comando time hacia la BD completa
- 3 ms aprox. de media para el comando mysqlslap hacia la tabla parche
- 28 ms aprox. de media para el comando time hacia la tabla parche
- 3 ms aprox. de media para el comando explain() en MongoDB
- 173 ms aprox. de media para el comando time en MongoDB

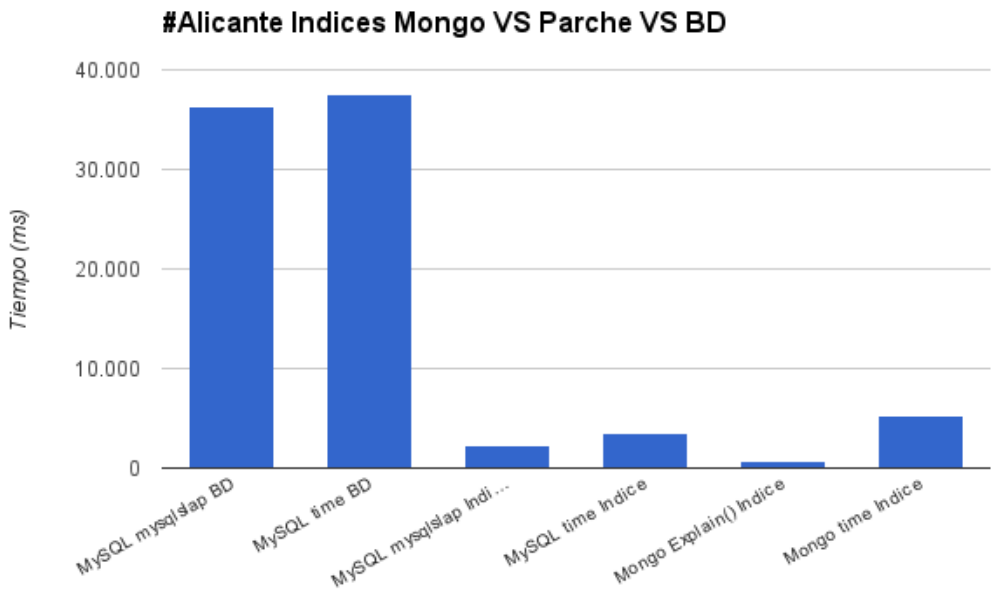


Gráfico de tiempos 16. Alicante Índices BD VS Tabla parche VS MongoDB

Obtenemos los siguientes valores:

- 36 seg aprox. (36.292 ms de media) para el comando mysqlslap hacia la BD completa
- 37 seg aprox. (37.462 ms de media) para el comando time hacia la BD completa
- 2 seg aprox. (2.359 ms de media) para el comando mysqlslap hacia la tabla parche
- 4 seg aprox. (3.588 ms de media) para el comando time hacia la tabla parche
- 753 ms de media para el comando explain() en MongoDB
- 5 seg aprox. (5.227 ms de media) para el comando time en MongoDB

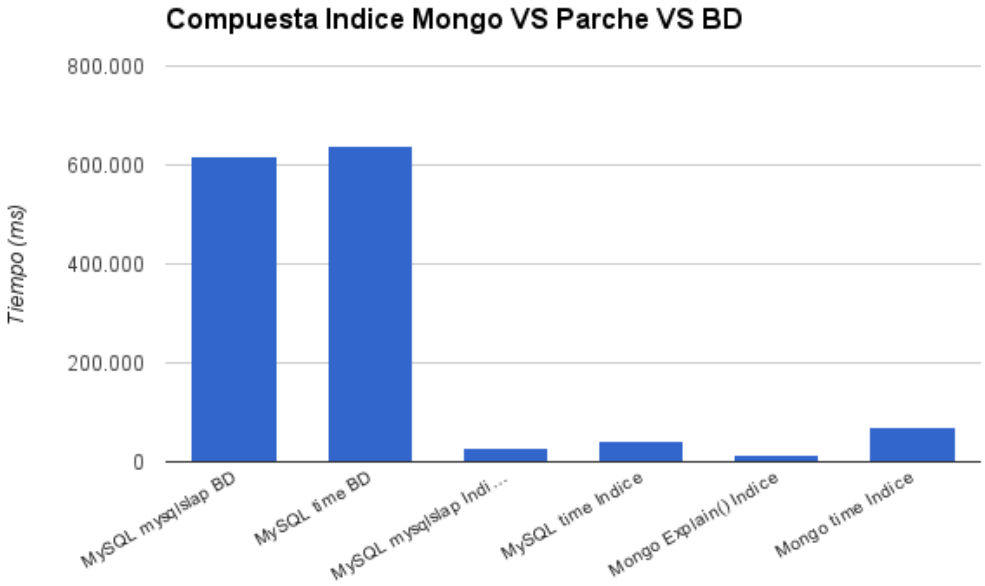


Gráfico de tiempos 17. Compuesta Índices BD VS Tabla parche VS MongoDB

Obtenemos los siguientes valores:

- 10 min 17 seg aprox. (616.578 ms de media) para el comando mysqlslap hacia la BD completa
- 10 min 37 seg aprox. (636.972 ms de media) para el comando time hacia la BD completa
- 30 seg aprox. (29.749 ms de media) para el comando mysqlslap hacia la tabla parche
- 43 seg aprox. (43.373 ms de media) para el comando time hacia la tabla parche
- 15 seg aprox. (14.606 ms de media) para el comando explain() en MongoDB
- 1 min 9 seg aprox. (69.308 ms de media) para el comando time en MongoDB

Para información más detallada consultar Anexo 5: Tablas de tiempos

Con estas últimas pruebas, la consulta a la BD completa queda definitivamente como la más lenta para devolver los registros deseados (salvo al devolver 43 filas según el comando time, que MongoDB es ligeramente más lento).

En lo que respecta a la comparación de los tiempos obtenidos por MongoDB y la tabla Parche, ocurre lo mismo que en los resultados anteriores, y es que según el comando *time* la tabla Parche es más rápida, incluso en la última consulta sin usar el índice correspondiente. Mantengo lo dicho y concibo que los tiempos obtenidos por el comando *time* son más fiables y, por tanto, la tabla Parche es la más rápida realizando consultas de datos.

Consultas de administración

Ahora faltaría comprobar las operaciones de administración después de crear el índice. En este caso sí puedo reutilizar los scripts anteriores, ya que me interesa el tiempo que se tarda en realizar el borrado y la inserción, no la consulta de los 10.000 primeros registros.

Después de cada consulta reiniciaré el servicio *mysql* y detendré el proceso *mongod*, ejecutaré Bleachbit con opción de limpiar la memoria y, por último iniciaré de nuevo el servidor de MongoDB. De ésta manera me aseguro que no se utilicen archivos temporales.

Éste es el gráfico obtenido de los tiempos medios de borrar en MongoDB y MySQL utilizando o no el índice compuesto:

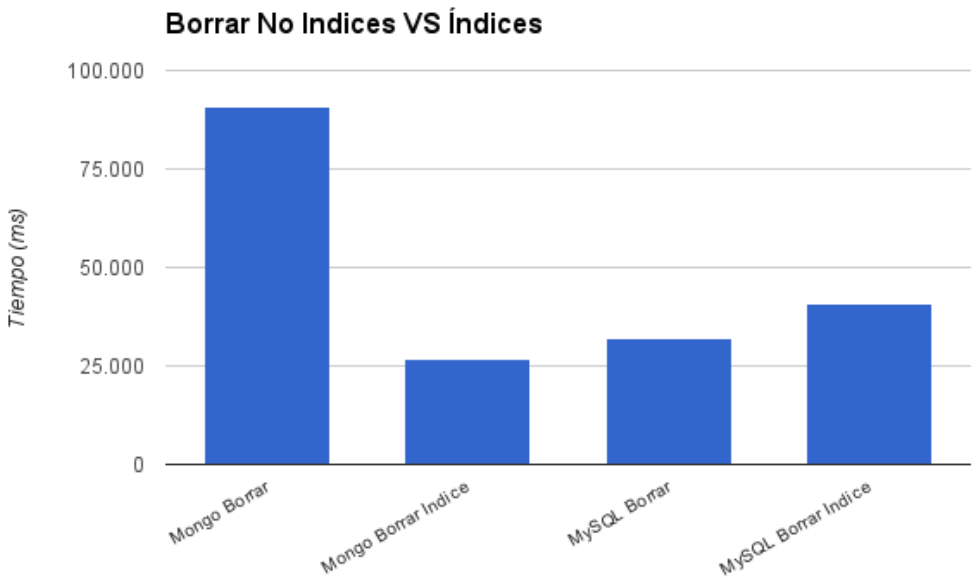


Gráfico de tiempos 18. Borrar No Índices VS Índices

Obtenemos los siguientes valores:

- 1 min y 31 seg aprox. (90.792 ms de media) para borrar en MongoDB
- 27 seg aprox. (26.781 ms de media) para borrar en MongoDB usando el índice
- 32 seg aprox. (32.064 ms de media) para borrar en MySQL
- 41 seg aprox. (40.754 ms de media) para borrar en MySQL usando el índice

Curiosamente, sólo el borrado en MongoDB se ha acelerado usando el índice.

Éste es el gráfico obtenido de los tiempos medios de insertar en MongoDB y MySQL utilizando o no el índice compuesto:

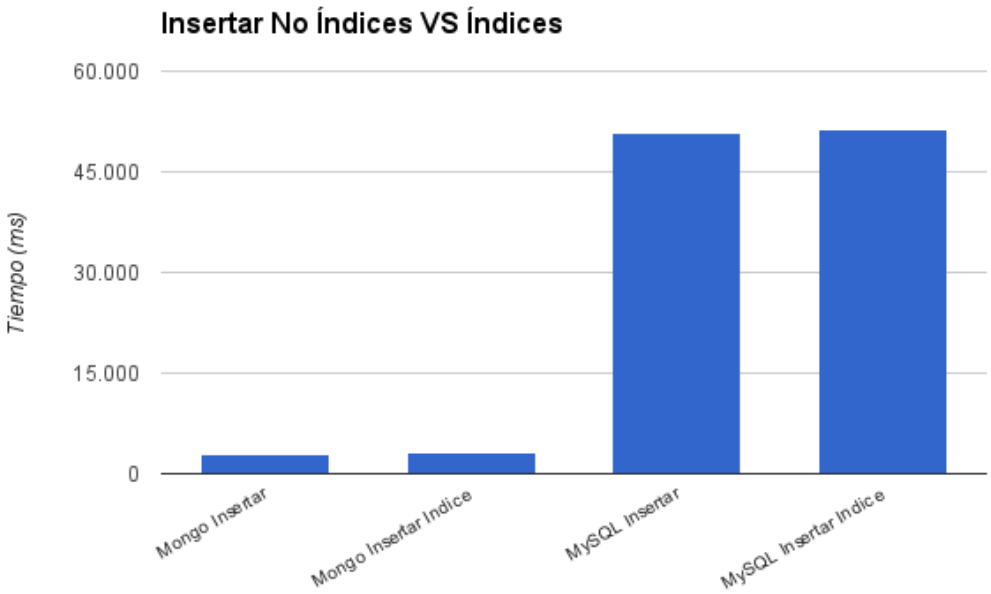


Gráfico de tiempos 19. Insertar No Índices VS Índices

Obtenemos los siguientes valores:

- 3 seg aprox. (2.827 ms de media) para insertar en MongoDB
- 3 seg aprox. (3.057 ms de media) para insertar en MongoDB usando el índice
- 51 seg aprox. (50.671 ms de media) para insertar en MySQL
- 51 seg aprox. (51.343 ms de media) para insertar en MySQL usando el índice

En el caso de la inserción no existe una diferencia notable para ninguno de los casos, el uso del índice ralentiza muy ligeramente la operación como se esperaba.

Para información más detallada consultar Anexo 5: Tablas de tiempos

Conclusiones

Este trabajo realiza un análisis de las actuales consultas tipo que se realizan en dos aplicaciones de minería de opiniones y sentimientos. Se enmarcan dentro de las conocidas como Procesamiento del Lenguaje Natural (PLN) y Tecnologías del Lenguaje Humano (TLH) grandes áreas de investigación que tratan de analizar información textual —también hablada pero realmente se trabaja con las transcripciones automáticas— para adquirir información sintáctica, semántica y hasta discursiva que permitan, finalmente, "que una máquina comprenda lo escrito por un humano".

Estas dos aplicaciones, Social Observer y Election Map, son similares en cuanto que su origen de datos es el mismo, Twitter, y su objetivo parecido, buscar tendencias de opinión o sentimiento, más generales en el primer caso y sobre intención de voto en elecciones el segundo.

Ambas aplicaciones recuperan tweet de la red social mediante la API publicada por Twitter, los analizan, y almacenan las piezas de información relevante en una estructura de base de datos relacional. Es importante este esquema ya que las consultas, que finalmente obtienen los resultados a mostrar al usuario, son muy variadas en cuanto los filtros de filas y columnas necesarios. Dicho de otra forma, dependiendo de la información objetivo, el concepto central de la búsqueda difiere: localización geográfica, cantidad de usuarios, cantidad de retweets, hashtags,… todo ello da lugar a datos elaborados diferentes y de distinto uso.

El origen del trabajo es el grupo de investigación GPLSI de la Universidad de Alicante, responsable del desarrollo y producción de estas dos aplicaciones. Hasta que obtuvieron una solución más o menos aceptable, el rendimiento de las consultas de usuario y de administración dejaba bastante que desear. Las posibilidades de optimización y de escalado del sistema eran, y son, limitadas. Se trabaja con servidores que no se pueden ampliar o mejorar por cuestiones presupuestarias y alternativas de hosting no ofrecen más garantías por el mismo motivo: dinero. Al mismo tiempo, la prioridad de los investigadores necesariamente deriva en otros asuntos y la búsqueda de mejores configuraciones o diseños ya no es la más urgente.

El trabajo se centra en establecer unas consultas tipo de tal forma que se puedan comparar los tiempos de ejecución de diversas aproximaciones:

- Consultas normales (*join*)

- Vista de agregación
- Tabla desnormalizada
- Datos migrados a MongoDB (NoSQL)

El hecho de diferenciar la vista de una consulta tal cual es que MySQL tiene conocidos problemas de rendimiento en la ejecución de vistas. No obstante, tal afirmación ha de comprobarse ya que no todos los contextos generan este problema. Ha de decirse que la vista, en este caso, consiste en un *join* de prácticamente todas las tablas del esquema sin más complicaciones, no hay subconsultas, *group by* u ordenaciones o complejas, ni tan siquiera filtros de fila excepto los necesarios para el *join*. Evidentemente, la consulta de la vista establecerá todos los filtros que se necesiten.

Precisamente la propia definición de la vista, que a priori obtiene todos los datos de la BD, hace que la primera consulta tipo que se plantee sea la recuperación de toda la información almacenada. Más tarde se probaron consultas más afines a las aplicaciones, es decir, filtrando por determinados valores.

Cabe destacar que uno de los argumentos de los desarrolladores de las aplicaciones, el pobre desempeño de la vista de agregación, rápidamente se vio desmontado porque se detectó un error en la definición de la misma.

La tabla desnormalizada es un intento de trasladar los datos que se obtienen de la vista a una única tabla. No se ha entrado en cómo incorporar esta forma de trabajar en las aplicaciones, simplemente se plantea como una posible adaptación que alguien decidirá si es posible y recomendable o no.

Finalmente, la migración a una base de datos NoSQL, MongoDB, parte de los propios investigadores dueños de las aplicaciones que, por los motivos aducidos anteriormente, no pudieron comprobar si realmente conseguirían mejoras significativas de rendimiento.

A partir de estos planteamientos mi trabajo se ha centrado en medir los tiempos de cada una de las alternativas, añadiendo índices en el caso de consultas filtradas por valores. Ha de decirse que el trabajo de poner en marcha la base de datos en mi propia máquina no ha sido trivial. También que para asegurarme que los tiempos sean correctos he repetido todas las consultas mencionadas en ambos motores después de haber apuntado los primeros valores y en la segunda pasada se han obtenido valores semejantes.

A la luz de los tiempos obtenidos para cada consulta, tanto de usuario como de administración, he podido observar los siguientes puntos:

- Se puede desnormalizar un sistema relacional como MySQL para obtener mejor rendimiento, pero el impacto en la recogida de datos de esta forma de trabajar en las aplicaciones no es conocido.
- El uso de índices en tablas mejora los tiempos de consulta cuando se utilizan filtros, pero no los de administración, ni a toda la información sin filtrar
- El uso de sistemas NoSQL como MongoDB seguramente no garantice un mejor rendimiento de las consultas, teniendo en cuenta el parámetro del tiempo real de ejecución de una consulta
- Las inserciones en MongoDB sí son mucho más rápidas

Hablando de MongoDB, es evidente que su ámbito de aplicación es otro y que aquí no se montado un sistema en producción como sería esperable. Sin ir más lejos, no se ha establecido un esquema de particionamiento (*sharding*) que, en principio, debería aportar paralelismo y cierta aceleración de las consultas. No obstante, a esto habría que añadir los tiempos de replicación (esquema típico de alta disponibilidad del producto) y de comunicación por la red. De hecho, las soluciones NoSQL no son más rápidas por definición, simplemente se aplican cuando la solución relacional se hace inviable, generalmente por esquemas de datos atípicos y problemas económicos de escalado horizontal.

Este trabajo, y sus conclusiones en particular, debe servir para que los investigadores del GPLSI dispongan de alternativas de mejora en sus aplicaciones, como así me lo pidieron y espero que para ello sirva.

Conocimientos adquiridos

Este trabajo ha supuesto un reto para mis conocimientos en bases de datos, no sólo por haber tenido que reutilizar conceptos aprendidos a lo largo de la realización del Grado en Ingeniería Informática y sus asignaturas relativas a las BBDD, sino que también me ha ayudado a comprender mejor entornos ya conocidos como MySQL y aprender el uso y funcionamiento de sistemas novedosos como MongoDB.

Más allá de lo que se ha concluido en este estudio comparativo, considero que el aprendizaje de un modelo no relacional como MongoDB me puede servir en mi futuro profesional de cara

a la comprensión del funcionamiento de otros sistemas NoSQL, o bien para optimizar el funcionamiento de SGBDR, como se ha realizado en el apartado desnormalización de la base de datos.

Durante la realización de este trabajo, sobretodo en sus inicios, la tarea de documentación ha sido la que más tiempo he dedicado (y he necesitado). Información sobre los sistemas no relacionales, estudios sobre su rendimiento y debates sobre la veracidad de los mismos fue simplemente una introducción para el comienzo de este trabajo. Más adelante tendría que utilizar recursos más especializados, como los manuales de referencia de MySQL para comprobar la optimización de las consultas o la propia base de datos; y para MongoDB la realización parcial de un curso en línea sobre el uso de esta herramienta en la administración de BBDD.

Aunque sí es cierto que he ampliado varios de los conceptos aprendidos en la carrera y, posiblemente, posea más competencias de las que se me solicitaba en la titulación, no podría presumir de haber adquirido los suficientes conocimientos o capacidades como para llamarme, en estos momentos, Administrador, ni mucho menos Experto en bases de datos en general, o MySQL y MongoDB en particular.

Sin embargo lo considero un primer paso para mi realización como profesional Ingeniero Informático y más concretamente gestor de una BBDD y es posible que continúe mi formación en esta vía para solicitar un empleo relacionado con esta tarea y dedicarme plenamente al perfeccionamiento de mis habilidades en dicho campo, así como la adquisición de nuevos conceptos y competencias.

Anexos

Anexo 0: Instalación En Linux Mint

Utilizaré las instrucciones para instalar MySQL Server desde las fuentes usando el repositorio, tal y como se explica en la documentación oficial

1. Lo primero será añadir el repositorio a nuestra máquina. Para ello seguimos las instrucciones[20]:
   1.1. Descargamos el paquete .deb que contendrá nuestro repositorio
   1.2. Utilizamos el comando: **sudo dpkg -i /Ruta_Descarga/paquete.deb**
   1.3. Ahora tendremos el repositorio, pero no tenemos las claves públicas. Para importarlas[21]: **gpg --recv-keys 5072E1F5**
   1.4. Ahora deberíamos poder hacer el apt-get update, pero no encuentra paquetes en el repositorio instalado, eso es porque nos ha instalado un repositorio con el formato:
   deb http://repo.mysql.com/apt/**linuxmint/ rosa** mysql-5.7
   Y debemos usar el repositorio de Ubuntu Trusty, para ello modificaremos las URLs de los repositorios para que apunten a esta distribución, de forma que nuestro **/etc/apt/sources.list.d/mysql.list** quede:
   deb http://repo.mysql.com/apt/ubuntu/ trusty connector-python-2.0
   deb http://repo.mysql.com/apt/ubuntu/ trusty mysql-5.7
   deb http://repo.mysql.com/apt/ubuntu/ trusty mysql-apt-config
   deb http://repo.mysql.com/apt/ubuntu/ trusty router-2.0
   deb http://repo.mysql.com/apt/ubuntu/ trusty mysql-utilities-1.5
   deb http://repo.mysql.com/apt/ubuntu/ trusty workbench-6.3
   deb-src http://repo.mysql.com/apt/ubuntu/ trusty mysql-5.7
2. Actualizamos nuestra lista de paquetes con: **sudo apt-get update**
3. Si instalamos pre compilados:
   3.1. Primero instalamos la base: **sudo apt-get install mysql-server**
   3.2. Pedirá una contraseña para el root de MySQL (2 veces)
   3.3. Luego podemos instalar los extras que queramos añadir **sudo apt-get install mysql-client mysql-utilities mysql-community-workbench**
   Nota: mysql-workbench (sin community) no puede convivir con mysql-utilities por dependencias del conector python

Anexo 1: Importación en Linux Mint

Una vez instalado MySQL en nuestra máquina, procedemos a importar la BD de SObserver con la que trabajaremos. Primero deberemos preparar el entorno.

Nos conectamos a la conexión por defecto, que utiliza el usuario *root* con el pass *mysql*. Ahora, en el editor de consultas escribimos el siguiente script:

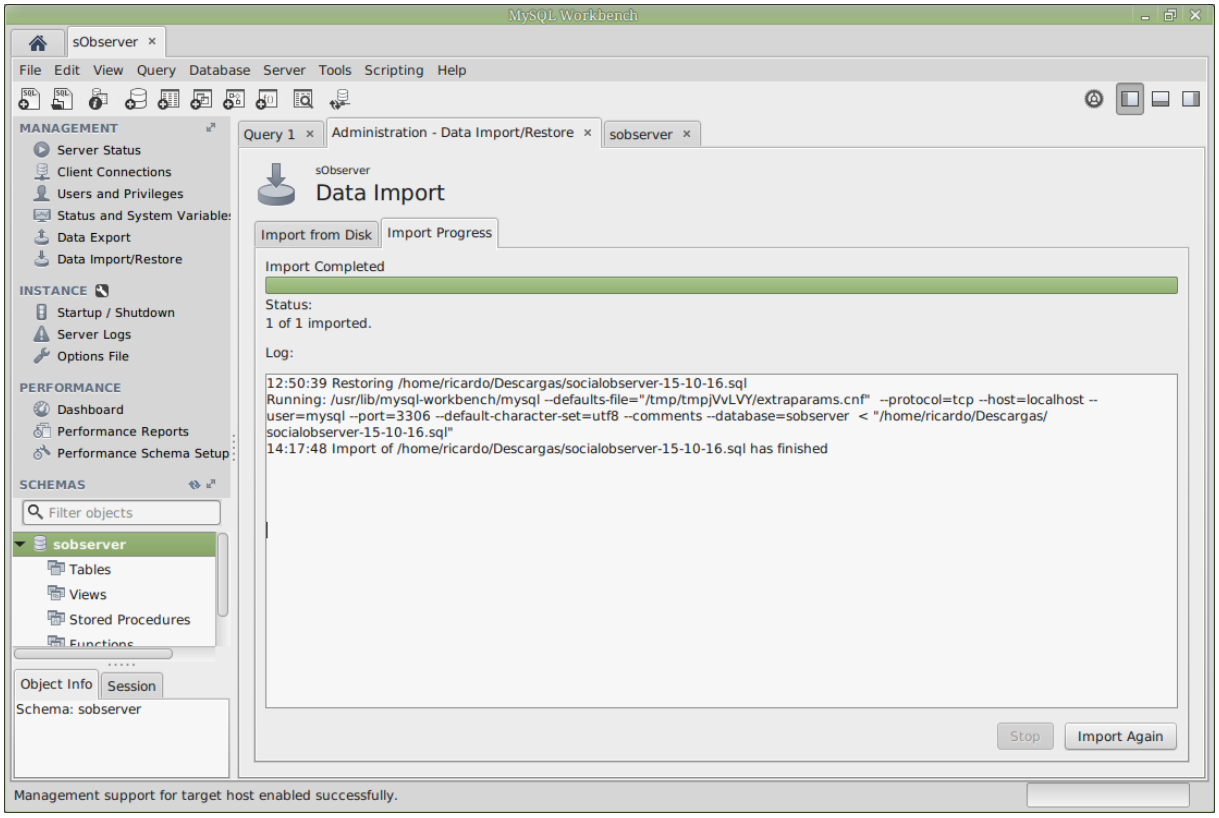
```
-- Creamos un usuario  mysql con todos los privilegios
CREATE USER 'mysql'@'%' IDENTIFIED BY 'mysql';
GRANT ALL PRIVILEGES ON *.* TO 'mysql'@'%' WITH GRANT OPTION;

-- Creamos un esquema  sobserver en charset utf-8
CREATE SCHEMA sobserver
  DEFAULT CHARACTER SET utf8
  DEFAULT COLLATE utf8_general_ci;
```

Ahora creamos una nueva conexión de mysql que apunte al esquema *sobserver* y use el usuario *mysql*. Procedemos a importar:

1. Abrimos MySQL Workbench e iniciamos la sesión local por defecto:
2. En el menú superior seleccionamos **Server > Data Import**. Se abrirá una pestaña de Data Import
3. En **Import from Self-Contained File** seleccionamos la carpeta que contenga el .sql extraído del Dump de la BD
4. Seleccionamos en **Default Target Schema** el esquema **sobserver**
5. Seleccionamos **Dump structure and Data** y pulsamos sobre **Start Import** para comenzar la importación
    a. Nota: En la pestaña **Administration data Import** mostrará el progreso de la importación, pero sólo cuando complete ciertas tareas, por lo que para asegurarnos que se están escribiendo datos podemos consultar:
        i. **Server Status** y ver que el Buffer y escrituras de InnoDB se están utilizando
        ii. **Client Connections** y buscar una Query update que, en detalle, está haciendo los INSERT
        iii. **Dashboard** Al igual que Server Status, nos dice el Buffer y escrituras de InnoDB

El proceso de importación en Linux Mint termina así:



Captura 1. Importación de la BD a MySQL

Han sido 2h 28min. aprox. el tiempo necesario para la importación de la BD. Desde MySQLWorkbench puedo observar algunos datos importantes de las tablas importadas:

Tabla	**Filas**	**Tamaño**
categories	9	16 KiB
entities	34	16 KiB
entities_has_terms	168	16 KiB
entity_has_restrictions	6	16 KiB
favourite_entities_data	45657	3,5 MiB
geographical_area	8077	512 KiB
hashtags	224868	10,3 MiB
info_web	0	0

license	10	16 KiB
license_operations	0	16 KiB
license_user	2	16 KiB
licenses_has_taxonomies	10	16 KiB
places	13421	1,3 MiB
provinces	0	8 KiB
registered_users	1221423	111,9 MiB
retweets	3162826	92 MiB
revision	0	16 KiB
scoring	3463680	141,8 MiB
social_city	8120	480 KiB
social_community	19	16 KiB
social_country	0	16 KiB
social_province	52	16 KiB
synonyms_terms	1084	96 KiB
taxonomies	3	16 KiB
taxonomies_has_categories	18	16 KiB
terms	170	16 KiB
terms_evaluation	1827	128 KiB
tweets	3390489	427 MiB
tweets_has_hashtags	4284021	157,8 MiB
tweets_has_terms	4123809	134,7 MiB

tweets_has_urls	1224844	45,6 MiB
tweet_translation	15024	496 KiB
twitter_access	10	16 KiB
urls	1070842	62,3 MiB
users	1344115	46,8 MiB
users_mention_tweets	5041855	189,8 MiB

Tabla 1. Importación de la BD. Objetos importados, filas y tamaño

Anexo 2: Consultas de aplicación

Éstas son las consultas extraídas de los archivos .java que me enviaron los desarrolladores para hacerme una idea de las consultas que estaban realizando. Se dividen en consultas de usuario (SELECT... WHERE...) y de gestión (DROP, INSERT, etc).

De usuario

El método getAggregatedDataView genera una consulta del estilo:

```
SELECT DISTINCT columna1, columna2,... columnaN
FROM 'aggregated_data'
[ WHERE columnaX=valorY || columnaX is not null [ AND
columnaW=valorZ || columnaW is not null ]
 [ AND tweetdate >= fInicio ]
 [ AND tweetdate <= fFin ]
[ ORDER BY ordenX ] ← date, term, taxonomy, entity, category,
user
[ LIMIT [ limiteOff || 0 ], limiteCont ]
```

```
SELECT DISTINCT count(*)
FROM 'aggregated_data'
[ WHERE columnaX=valorY || columnaX is not null [ AND
columnaW=valorZ || columnaW is not null ]
 [ AND tweetdate >= fInicio ]
 [ AND tweetdate <= fFin ]
[ ORDER BY ordenX ] ← date, term, taxonomy, entity, category,
user
[ LIMIT [ limiteOff || 0 ], limiteCont ]
```

También existe el método getAggregatedDataStore, que utiliza casi la misma consulta, solo que permite excluir ciertas columnas

De gestión

El método addEntitiesTermsMap genera consultas:

```
SELECT *
FROM entities
WHERE name=nombreEntidad
 AND license_ID=licencia
```

```
INSERT INTO entities (name,license_ID)
VALUES (nombreEntidad,licencia)
```

```
SELECT term_ID
FROM terms
WHERE term=terminoX
```

```
INSERT INTO terms (term,used)
VALUES (terminoX,0)
```

```
INSERT INTO entities_has_terms (entity_ID,terms_term_ID)
VALUES (idEntidad,idUltimoTerminoUsado)
```

```
SELECT 1
FROM entities_has_terms
WHERE entity_ID=idEntidad
 AND terms_term_ID=terminoX
```

```
INSERT INTO entities_has_terms (entity_ID,terms_term_ID)
VALUES (idEntidad,terminoX)
```

El método addLicenseOperationPairList:

```
INSERT INTO license_operations (license_ID,operation)
VALUES (licencia,operacionX)
```

El método addTwitterLicense:

```
SELECT user_ID
FROM twitter_access
WHERE user_ID=usuario
```

```
INSERT INTO 'license'
('license_ID','date_ini','date_end','status','target_languaje
','observation_period')
VALUES (licencia,fInicio,fFin,"stopped","es","129600")
```

```
INSERT INTO 'twitter_access'
('license_ID','user_ID','accesstoken','tokensecret')
VALUES (licencia,usuario,acceso,secreto)
```

```
SELECT taxonomy_ID
FROM taxonomies
WHERE name=metodoX
```

```
INSERT INTO license_has_taxonomies (license_ID,taxonomy_ID)
VALUES (licencia,idTaxonomia)
```

El método addTaxonomy:

```
INSERT INTO 'taxonomies' ('name','description','web_service')
VALUES (nTaxonomia,descripcion,servicio)
```

```
SELECT taxonomy_ID
FROM taxonomies
WHERE name=nTaxonomia
```

```
INSERT INTO 'categories' ('name')
VALUES (nCategoria)
```

```
SELECT category_ID
FROM categories
WHERE name=nCategoria
```

```
INSERT INTO taxonomies_has_categories
('taxonomy_ID','category_ID')
VALUES (idTaxonomia,idCategoria)
```

El método deleteLicense:

```
SELECT license_ID
FROM license
WHERE license_ID=licencia
```

```
SELECT entity_ID
FROM entities
WHERE license_ID=licencia
```

```
DELETE FROM licenses_has_taxonomies
WHERE license_ID=licencia
```

```
DELETE FROM entities_has_terms
WHERE entity_ID=idEntidad
```

```
DELETE FROM entities
WHERE license_ID=licencia
```

```
DELETE FROM license_operations
WHERE license_ID=licencia
```

```
DELETE FROM twitter_access
WHERE license_ID=licencia
```

```
DELETE FROM 'license'
WHERE license_ID=licencia
```

El método getCategoriesFromTaxonomy:

```
SELECT c
FROM TaxonomiesHasCategories h, Categories c
WHERE h.taxonomiesHasCategoriesPK.categoryID=c.categoryID
 AND h.taxonomiesHasCategoriesPK.taxonomyID=idTaxonomia
```

El método getTermsFromEntity:

```
SELECT t
FROM EntitiesHasTerms h, Terms t
WHERE h.entitiesHasTermsPK.termstermID=t.termID
 AND h.entitiesHasTermsPK.entityID=idEntidad
```

El método setScoreManualOpinion:

```
UPDATE scoring
SET score_manual=puntuación
WHERE tweet_ID=idTweet
 AND taxonomy_ID = idTaxonomia
```

```
UPDATE aggregated_data_store
SET scoremanual=puntuación, score=puntuación
WHERE tweet_ID=idTweet
 AND taxonomy_ID=idTaxonomia
```

El método updateLicense:

```
SELECT license_ID
FROM license
WHERE license_ID=licencia
```

```
SELECT taxonomy_ID
FROM taxonomies
WHERE name=metodoX
```

```
UPDATE 'twitter_access'
SET 'user_ID'=usuario,
'accesstoken'=acceso,'tokensecret'=secreto
WHERE 'license_ID'=licencia
```

```
UPDATE 'license'
SET 'date_ini'=fInicio, 'date_end'=fFin,
WHERE 'license_ID'=licencia
```

```
DELETE FROM licenses_has_taxonomies
WHERE 'license_ID'=licencia
```

```
INSERT INTO licenses_has_taxonomies (license_ID, taxonomy_ID)
VALUES(licencia, idTaxonomia)
```

También existe los métodos getAggregatedDataView y getAggregatedDataStore, exactamente iguales que en las consultas de usuario.

Anexo 3: Consultas de MySQL

Consulta de la BD completa; script *sql-lenta.sql*:

```
SELECT DISTINCT
    l.license_ID AS licenseid,
    e.entity_ID AS entityid,
    e.name AS entity,
    t.term_ID AS termid,
    t.term AS term,
    tw.tweet_ID AS tweetid,
    rt.retweet_ID AS retweetid,
    tw.text AS tweet,
    tax.taxonomy_ID AS taxonomyid,
    tax.name AS taxonomy,
    c.category_ID AS categoryid,
    c.name AS category,
    s.score AS score,
    s.score_manual AS scoremanual,
    tw.date AS tweetdate,
    p.place_ID AS placeid,
    p.name AS place,
    u.user_ID AS userid,
    u.screen_name AS user,
    ru.location AS userlocation,
    p.country AS country,
    tw.lang AS language
FROM
    ((((((((((((sobserver.license l
    JOIN sobserver.entities e ON ((l.license_ID =
e.license_ID)))
    JOIN sobserver.entities_has_terms et ON ((e.entity_ID =
et.entity_ID)))
    JOIN sobserver.terms t ON ((et.terms_term_ID =
t.term_ID)))
    JOIN sobserver.tweets_has_terms tt ON ((t.term_ID =
tt.term_ID)))
    JOIN sobserver.tweets tw ON ((tt.tweet_ID =
tw.tweet_ID)))
    LEFT JOIN sobserver.retweets rt ON ((rt.tweet_ID =
tw.tweet_ID)))
    LEFT JOIN sobserver.places p ON ((tw.place_ID =
p.place_ID)))
    LEFT JOIN sobserver.users u ON ((tw.user_ID =
u.user_ID)))
    LEFT JOIN sobserver.registered_users ru ON ((u.user_ID =
ru.registered_user_ID)))
    LEFT JOIN sobserver.scoring s ON ((tw.tweet_ID =
```

```
s.tweet_ID)))
    LEFT JOIN sobserver.taxonomies_has_categories tc ON
(((s.taxonomy_ID = tc.taxonomy_ID)
        AND (s.category_ID = tc.category_ID))))
    LEFT JOIN sobserver.categories c ON ((tc.category_ID =
c.category_ID)))
    LEFT JOIN sobserver.taxonomies tax ON ((tc.taxonomy_ID =
tax.taxonomy_ID)))
    LEFT JOIN sobserver.licenses_has_taxonomies lt ON
((tax.taxonomy_ID = lt.taxonomy_ID)))
;
```

Consulta de la BD con filtros simples; script *sql-lenta-where.sql* y *sql-lenta-where-alicante.sql*:

```
SELECT DISTINCT
    l.license_ID AS licenseid,
    e.entity_ID AS entityid,
    e.name AS entity,
    t.term_ID AS termid,
    t.term AS term,
    tw.tweet_ID AS tweetid,
    rt.retweet_ID AS retweetid,
    tw.text AS tweet,
    tax.taxonomy_ID AS taxonomyid,
    tax.name AS taxonomy,
    c.category_ID AS categoryid,
    c.name AS category,
    s.score AS score,
    s.score_manual AS scoremanual,
    tw.date AS tweetdate,
    p.place_ID AS placeid,
    p.name AS place,
    u.user_ID AS userid,
    u.screen_name AS user,
    ru.location AS userlocation,
    p.country AS country,
    tw.lang AS language
FROM
    ((((((((((((sobserver.license l
    JOIN sobserver.entities e ON ((l.license_ID =
e.license_ID)))
    JOIN sobserver.entities_has_terms et ON ((e.entity_ID =
et.entity_ID)))
    JOIN sobserver.terms t ON ((et.terms_term_ID =
t.term_ID)))
    JOIN sobserver.tweets_has_terms tt ON ((t.term_ID =
tt.term_ID)))
    JOIN sobserver.tweets tw ON ((tt.tweet_ID =
```

```
tw.tweet_ID)))
    LEFT JOIN sobserver.retweets rt ON ((rt.tweet_ID =
tw.tweet_ID)))
    LEFT JOIN sobserver.places p ON ((tw.place_ID =
p.place_ID)))
    LEFT JOIN sobserver.users u ON ((tw.user_ID =
u.user_ID)))
    LEFT JOIN sobserver.registered_users ru ON ((u.user_ID =
ru.registered_user_ID)))
    LEFT JOIN sobserver.scoring s ON ((tw.tweet_ID =
s.tweet_ID)))
    LEFT JOIN sobserver.taxonomies_has_categories tc ON
(((s.taxonomy_ID = tc.taxonomy_ID)
        AND (s.category_ID = tc.category_ID))))
    LEFT JOIN sobserver.categories c ON ((tc.category_ID =
c.category_ID)))
    LEFT JOIN sobserver.taxonomies tax ON ((tc.taxonomy_ID =
tax.taxonomy_ID)))
    LEFT JOIN sobserver.licenses_has_taxonomies lt ON
((tax.taxonomy_ID = lt.taxonomy_ID)))
WHERE
    term = "Javier Echenique"
    -- term = "#Alicante"
;
```

Consulta de la BD con filtro compuesto; script *sql-lenta-where-compuesto.sql*:

```
SELECT DISTINCT
    l.license_ID AS licenseid,
    e.entity_ID AS entityid,
    e.name AS entity,
    t.term_ID AS termid,
    t.term AS term,
    tw.tweet_ID AS tweetid,
    rt.retweet_ID AS retweetid,
    tw.text AS tweet,
    tax.taxonomy_ID AS taxonomyid,
    tax.name AS taxonomy,
    c.category_ID AS categoryid,
    c.name AS category,
    s.score AS score,
    s.score_manual AS scoremanual,
    tw.date AS tweetdate,
    p.place_ID AS placeid,
    p.name AS place,
    u.user_ID AS userid,
    u.screen_name AS user,
    ru.location AS userlocation,
    p.country AS country,
```

```
    tw.lang AS language
FROM
    ((((((((((((sobserver.license l
    JOIN sobserver.entities e ON ((l.license_ID =
e.license_ID)))
    JOIN sobserver.entities_has_terms et ON ((e.entity_ID =
et.entity_ID)))
    JOIN sobserver.terms t ON ((et.terms_term_ID =
t.term_ID)))
    JOIN sobserver.tweets_has_terms tt ON ((t.term_ID =
tt.term_ID)))
    JOIN sobserver.tweets tw ON ((tt.tweet_ID =
tw.tweet_ID)))
    LEFT JOIN sobserver.retweets rt ON ((rt.tweet_ID =
tw.tweet_ID)))
    LEFT JOIN sobserver.places p ON ((tw.place_ID =
p.place_ID)))
    LEFT JOIN sobserver.users u ON ((tw.user_ID =
u.user_ID)))
    LEFT JOIN sobserver.registered_users ru ON ((u.user_ID =
ru.registered_user_ID)))
    LEFT JOIN sobserver.scoring s ON ((tw.tweet_ID =
s.tweet_ID)))
    LEFT JOIN sobserver.taxonomies_has_categories tc ON
(((s.taxonomy_ID = tc.taxonomy_ID)
        AND (s.category_ID = tc.category_ID))))
    LEFT JOIN sobserver.categories c ON ((tc.category_ID =
c.category_ID)))
    LEFT JOIN sobserver.taxonomies tax ON ((tc.taxonomy_ID =
tax.taxonomy_ID)))
    LEFT JOIN sobserver.licenses_has_taxonomies lt ON
((tax.taxonomy_ID = lt.taxonomy_ID)))
WHERE
    (
        term LIKE "%PP%"
    OR
        term LIKE "%ppopular%"
    OR
        term LIKE "%psoe%"
    OR
        term LIKE "%ciuda%"
    OR
        term LIKE "%compromis%"
    OR
        term LIKE "%UPyD%"
    )
;
```

Consulta a la Vista de la BD; script *sql-vista.sql*:

```
SELECT
    *
FROM
    sobserver.aggregated_data
;
```

Consulta a la Vista de la BD con filtros simples; scripts *sql-vista-where.sql* y *sql-vista-where-alicante.sql*:

```
SELECT
    *
FROM
    sobserver.aggregated_data
WHERE
    term = "Javier Echenique"
    -- term = "#Alicante"
;
```

Consulta a la Vista de la BD con filtro compuesto; script *sql-vista-where-compuesto.sql*:

```
SELECT
    *
FROM
    sobserver.aggregated_data
WHERE
    (
        term LIKE "%PP%"
    OR
        term LIKE "%ppopular%"
    OR
        term LIKE "%psoe%"
    OR
        term LIKE "%ciuda%"
    OR
        term LIKE "%compromis%"
    OR
        term LIKE "%UPyD%"
    )
;
```

Consulta a la tabla ‘Parche’, script *sql-parche.sql*:

```
SELECT
     *
FROM
    sobserver.tabla_parche
;
```

Consulta a la tabla ‘Parche’ con filtros simples, scripts *sql-parche-where.sql* y *sql-parche-where-alicante.sql*:

```
SELECT
     *
FROM
    sobserver.tabla_parche
WHERE
    term = "Javier Echenique"
    -- term = "#Alicante"
;
```

Consulta a la tabla ‘Parche’ con filtro compuesto, script *sql-parche-where-compuesto.sql*:

```
SELECT
     *
FROM
    sobserver.tabla_parche
WHERE
    (
        term LIKE "%PP%"
    OR
        term LIKE "%ppopular%"
    OR
        term LIKE "%psoe%"
    OR
        term LIKE "%ciuda%"
    OR
        term LIKE "%compromis%"
    OR
        term LIKE "%UPyD%"
    )
;
```

Anexo 4: Optimización de la BD

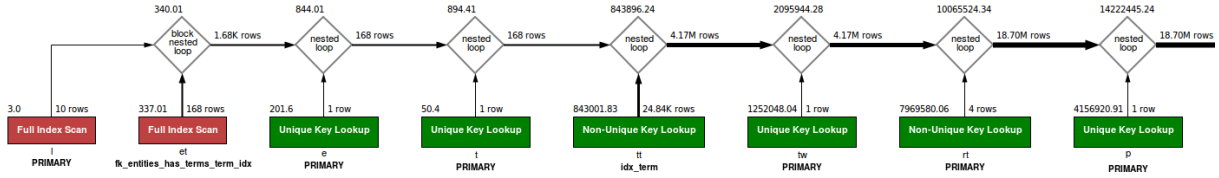
Necesito saber los motores usados por las tablas de la BD SObserver:

Nombre	Motor	Formato de filas
categories	InnoDB	Dinámico
entities	InnoDB	Dinámico
entities_has_terms	InnoDB	Dinámico
entity_has_restrictions	InnoDB	Dinámico
favourite_entities_data	InnoDB	Dinámico
geographical_area	InnoDB	Dinámico
hashtags	InnoDB	Comprimido
info_web	MyISAM	Dinámico
license	InnoDB	Dinámico
license_operations	InnoDB	Dinámico
license_user	InnoDB	Dinámico
licenses_has_taxonmies	InnoDB	Dinámico
places	InnoDB	Comprimido
provinces	InnoDB	Comprimido
registered_users	InnoDB	Comprimido
retweets	InnoDB	Comprimido
revision	InnoDB	Dinámico
scoring	InnoDB	Dinámico
social_city	InnoDB	Dinámico
social_community	InnoDB	Dinámico
social_country	InnoDB	Dinámico
social_province	InnoDB	Dinámico
synonyms_terms	InnoDB	Dinámico
taxonomies	InnoDB	Dinámico
taxonomies_has_categories	InnoDB	Dinámico
terms	InnoDB	Dinámico
terms_evaluation	InnoDB	Dinámico

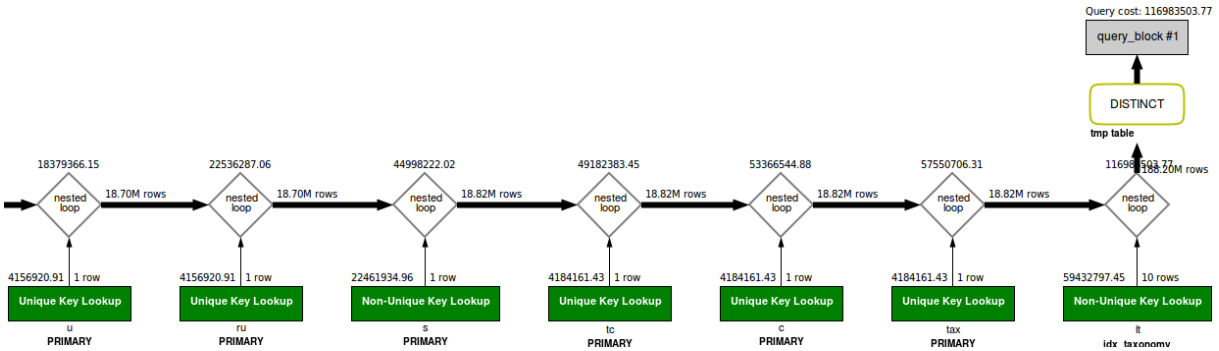
tweet_translation	InnoDB	Dinámico
tweets	InnoDB	Comprimido
tweets_has_hashtags	InnoDB	Dinámico
tweets_has_terms	InnoDB	Dinámico
tweets_has_urls	InnoDB	Dinámico
twitter_access	InnoDB	Dinámico
urls	InnoDB	Comprimido
users	InnoDB	Comprimido
users_mention_tweets	InnoDB	Dinámico

Tabla 2. Optimización de la BD. Motores y compresión de las tablas

Para mostrar los índices utilizados en las consultas podemos utilizar la herramienta MySQL Workbench, que mostrará tanto de forma gráfica como por texto el Explain Plan utilizado para la consulta. Primero los índices utilizados en la consulta a BD con filtros simples:



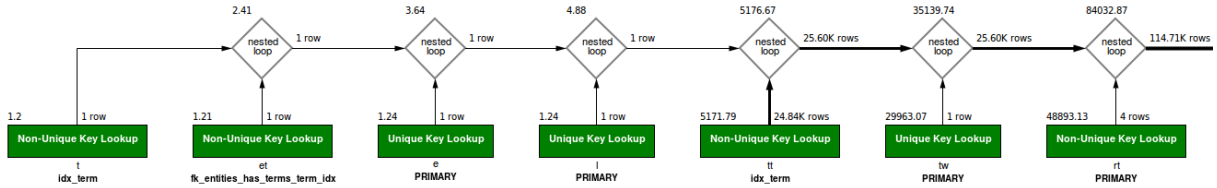
Explain Plan de la consulta a BD (parte 1)



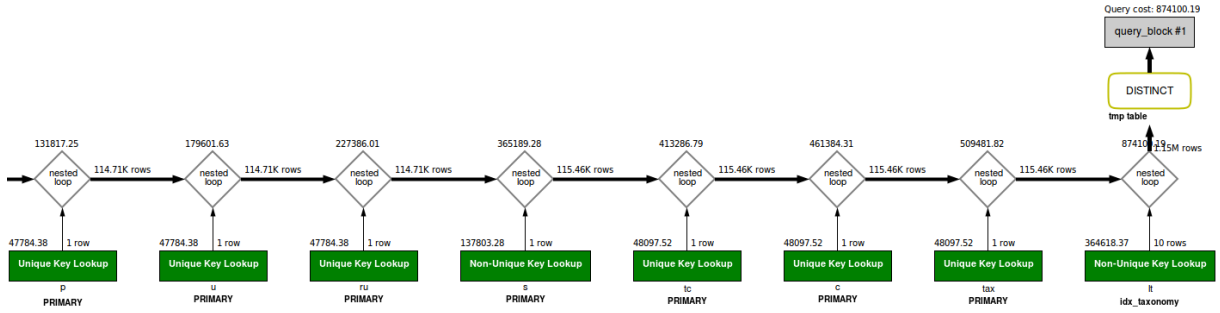
Explain Plan de la consulta a BD (parte 2)

Vemos que se utiliza un índice para casi cada columna que se quiere consultar, un total de 15 índices y que utiliza una tabla temporal *tmp table* para devolver los resultados que, primero deberán pasar por el DISTINCT.

Ahora la misma consulta filtrada contra la Vista:



Explain Plan de la consulta a Vista (parte 1)



Explain Plan de la consulta a Vista (parte 2)

Aquí muestra que la vista, efectivamente, utiliza los mismos índices que la consulta a BD, de hecho el Plan es prácticamente el mismo, ya que MySQL detecta automáticamente la consulta con mejor rendimiento para devolver los resultados.

Sobre la creación de la Vista ‘aggregated_data’ resulta que se usaba un DISTINC en el procedimiento que la generaba, por lo que hay que cambiar este comportamiento. El script de generación de la Vista *create-view.sql* queda así:

```
CREATE
    ALGORITHM = MERGE
VIEW `aggregated_data` AS
    SELECT
        `l`.`license_ID` AS `licenseid`,
        `e`.`entity_ID` AS `entityid`,
        `e`.`name` AS `entity`,
        `t`.`term_ID` AS `termid`,
        `t`.`term` AS `term`,
        `tw`.`tweet_ID` AS `tweetid`,
        `rt`.`retweet_ID` AS `retweetid`,
        `tw`.`text` AS `tweet`,
        `tax`.`taxonomy_ID` AS `taxonomyid`,
        `tax`.`name` AS `taxonomy`,
        `c`.`category_ID` AS `categoryid`,
        `c`.`name` AS `category`,
        `s`.`score` AS `score`,
        `s`.`score_manual` AS `scoremanual`,
        `tw`.`date` AS `tweetdate`,
        `p`.`place_ID` AS `placeid`,
        `p`.`name` AS `place`,
```

```
        `u`.`user_ID` AS `userid`,
        `u`.`screen_name` AS `user`,
        `ru`.`location` AS `userlocation`,
        `p`.`country` AS `country`,
        `tw`.`lang` AS `language`
    FROM
        ((((((((((((`license` `l`
        JOIN `entities` `e` ON ((`l`.`license_ID` =
`e`.`license_ID`)))
        JOIN `entities_has_terms` `et` ON ((`e`.`entity_ID` =
`et`.`entity_ID`)))
        JOIN `terms` `t` ON ((`et`.`terms_term_ID` =
`t`.`term_ID`)))
        JOIN `tweets_has_terms` `tt` ON ((`t`.`term_ID` =
`tt`.`term_ID`)))
        JOIN `tweets` `tw` ON ((`tt`.`tweet_ID` =
`tw`.`tweet_ID`)))
        LEFT JOIN `retweets` `rt` ON ((`rt`.`tweet_ID` =
`tw`.`tweet_ID`)))
        LEFT JOIN `places` `p` ON ((`tw`.`place_ID` =
`p`.`place_ID`)))
        LEFT JOIN `users` `u` ON ((`tw`.`user_ID` =
`u`.`user_ID`)))
        LEFT JOIN `registered_users` `ru` ON ((`u`.`user_ID`
= `ru`.`registered_user_ID`)))
        LEFT JOIN `scoring` `s` ON ((`tw`.`tweet_ID` =
`s`.`tweet_ID`)))
        LEFT JOIN `taxonomies_has_categories` `tc` ON
(((`s`.`taxonomy_ID` = `tc`.`taxonomy_ID`)
            AND (`s`.`category_ID` = `tc`.`category_ID`))))
        LEFT JOIN `categories` `c` ON ((`tc`.`category_ID` =
`c`.`category_ID`)))
        LEFT JOIN `taxonomies` `tax` ON ((`tc`.`taxonomy_ID`
= `tax`.`taxonomy_ID`)))
        LEFT JOIN `licenses_has_taxonomies` `lt` ON
((`tax`.`taxonomy_ID` = `lt`.`taxonomy_ID`)))
;
```

Anexo 5: Tablas de Tiempos

Sistema	MySQL			
Comando	Vista		BD	
Tabla	Mysqlslap	time	Mysqlslap	time
Tiempos (ms)	1.579.184	1.670.486	1.566.537	1.593.588
	1.610.185	1.675.677	1.560.180	1.586.163
	1.675.047	1.670.733	1.563.235	1.580.916
T. Medio (ms)	1.621.472	1.672.299	1.563.317	1.586.889

Tabla 3. Tiempos. Vista VS BD completa

Sistema	MySQL			
Término	Javier Echenique			
Comando	Vista		BD	
Tabla	Mysqlslap	time	Mysqlslap	time
Tiempos (ms)	99	138	91	113
	99	112	94	103
	94	106	97	111
T. Medio (ms)	97	119	94	109

Tabla 4. Tiempos. Javier Echenique Vista VS BD

Sistema	MySQL			
Término	#Alicante			
Comando	Vista		BD	
Tabla	Mysqlslap	time	Mysqlslap	time
Tiempos (ms)	38.371	37.456	36.464	37.503
	37.548	38.481	36.010	37.614
	36.176	37.331	36.403	37.269
T. Medio (ms)	37.365	37.756	36.292	37.462

Tabla 5. Tiempos. Alicante Vista VS BD

Sistema	MySQL			
Término	**Compuesta**			
Comando	**Vista**		**BD**	
Tabla	**Mysqlslap**	**time**	**Mysqlslap**	**time**
Tiempos (ms)	618.381	646.870	618.621	645.065
	622.319	637.675	611.162	632.407
	627.839	639.639	619.951	633.445
T. Medio (ms)	622.846	641.395	616.578	636.972

Tabla 6. Tiempos. Compuesta Vista VS BD

Sistema	MySQL			
Comando	**BD**		**Parche**	
Tabla	**Mysqlslap**	**time**	**Mysqlslap**	**time**
Tiempos (ms)	1.566.537	1.618.761	42.267	73.865
	1.560.180	1.586.163	40.638	74.816
	1.563.235	1.580.916	42.998	75.083
T. Medio (ms)	1.563.317	1.595.280	41.968	74.588

Tabla 7. Tiempos. BD VS Parche

Sistema	MySQL		MongoDB	
Tabla	**Parche**		**-**	
Comando	**Mysqlslap**	**time**	**Explain()**	**time**
Tiempos (ms)	42.267	73.865	15.761	120.538
	40.638	74.816	15.906	131.481
	42.998	75.083	16.079	127.726
T. Medio (ms)	41.968	74.588	15.915	126.582

Tabla 8. Tiempos. Parche VS Mongo

Sistema	MySQL				MongoDB	
Término	Javier Echenique					
Comando	BD		Parche		-	
Tabla	Mysqlslap	time	Mysqlslap	time	Explain()	time
Tiempos (ms)	91	113	9.956	11.361	16.337	16.408
	94	103	10.875	12.484	16.307	16.020
	97	111	10.011	12.223	16.368	16.612
T. Medio (ms)	94	109	10.281	12.023	16.337	16.347

Tabla 9. Tiempos. Javier Echenique BD VS Parche VS Mongo

Sistema	MySQL				MongoDB	
Término	#Alicante					
Comando	BD		Parche		-	
Tabla	Mysqlslap	time	Mysqlslap	time	Explain()	time
Tiempos (ms)	36.464	37.503	11.225	15.276	16.054	21.609
	36.010	37.614	12.098	13.781	16.085	20.681
	36.403	37.269	12.022	14.788	16.101	20.613
T. Medio (ms)	36.292	37.462	11.782	14.615	16.080	20.968

Tabla 10. Tiempos. Alicante BD VS Parche VS Mongo

Sistema	MySQL				MongoDB	
Término	Compuesta					
Comando	BD		Parche		-	
Tabla	Mysqlslap	time	Mysqlslap	time	Explain()	time
Tiempos (ms)	618.621	645.065	30.415	47.320	16.728	62.453
	611.162	632.407	28.783	41.517	16.932	62.334
	619.951	633.445	30.048	41.283	17.059	62.097
T. Medio (ms)	616.578	636.972	29.749	43.373	16.906	62.295

Tabla 11. Tiempos. Compuesta BD VS Parche VS Mongo

Sistema	MongoDB		MySQL	
Time >	**Borrar**	**Insertar**	**Borrar**	**Insertar**
Tiempos (ms)	90.625	3.753	31.078	51.001
	90.440	2.518	31.227	51.829
	91.311	2.211	33.886	49.182
T. Medio (ms)	90.792	2.827	32.064	50.671

Tabla 12. Tiempos. Administración Mongo VS Parche

Sistema	MySQL				MongoDB	
Término	**Javier Echenique**					
Comando	**BD**		**Parche**		**-**	
Tabla	**Mysqlslap**	**time**	**Mysqlslap**	**time**	**Explain()**	**time**
Tiempos (ms)	91	113	3	22	2	232
	94	103	2	23	6	165
	97	111	3	39	2	122
T. Medio (ms)	94	109	3	28	3	173

Tabla 13. Tiempos. Javier Echenique Índices BD VS Parche VS MongoDB

Sistema	MySQL				MongoDB	
Término	**#Alicante**					
Comando	**BD**		**Parche**		**-**	
Tabla	**Mysqlslap**	**time**	**Mysqlslap**	**time**	**Explain()**	**time**
Tiempos (ms)	36.464	37.503	2.426	3.795	804	5.100
	36.010	37.614	2.281	3.472	723	5.777
	36.403	37.269	2.370	3.498	733	4.803
T. Medio (ms)	36.292	37.462	2.359	3.588	753	5.227

Tabla 14. Tiempos. Alicante Índices BD VS Parche VS MongoDB

Sistema	MySQL				MongoDB	
Término	Compuesta					
Comando	BD		Parche		-	
Tabla	Mysqlslap	time	Mysqlslap	time	Explain()	time
Tiempos (ms)	618.621	645.065	30.415	47.320	14.611	66.793
	611.162	632.407	28.783	41.517	14.126	65.679
	619.951	633.445	30.048	41.283	15.081	75.453
T. Medio (ms)	616.578	636.972	29.749	43.373	14.606	69.308

Tabla 15. Tiempos. Compuesta Índices BD VS Parche VS MongoDB

Sistema	MongoDB				MySQL			
Time >	Borrar		Insertar		Borrar		Insertar	
Tabla	No Índice	Índice	No Índice	Índice	No Índice	Índice	No Índice	Índice
Tiempos (ms)	90.625	27.001	3.753	2.724	31.078	40.527	51.001	56.357
	90.440	26.486	2.518	3.383	31.227	43.640	51.829	48.950
	91.311	26.857	2.211	3.063	33.886	38.095	49.182	48.721
T. Medio (ms)	90.792	26.781	2.827	3.057	32.064	40.754	50.671	51.343

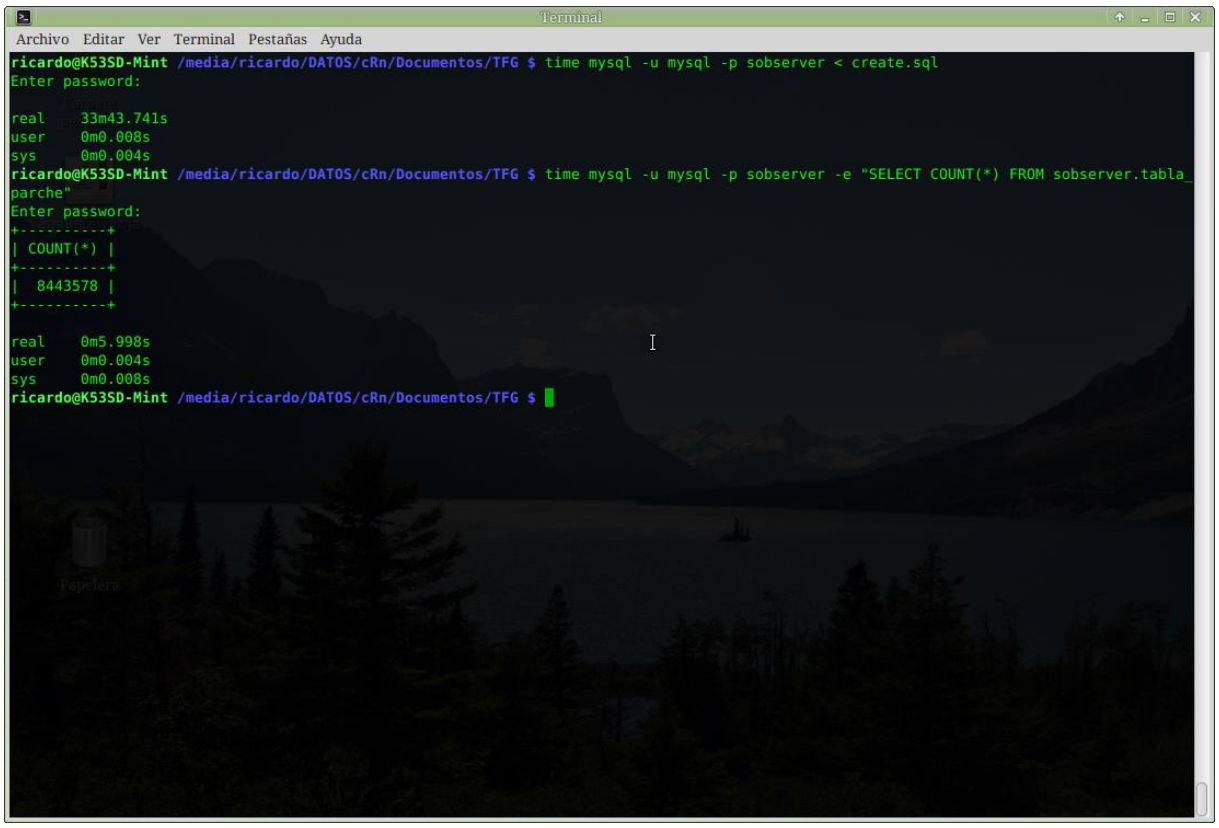
Tabla 16. Tiempos. Administración No Índices VS Índices

Anexo 6: Desnormalización de la BD en MySQL

Procedemos a la creación de una tabla 'parche' en MySQL con los datos contenidos en la BD y observables en la Vista aggregated_data, para ello utilizamos un comando CREATE TABLE... SELECT en MySQL. Debido a que hay algunas fechas de tweets en la BD introducidas a mano, hay que utilizar el comando SET para poder importar las fecha inválidas. Éste es el Script utilizado:

```
SET SQL_MODE='ALLOW_INVALID_DATES';
CREATE TABLE tabla_parche SELECT
`aggregated_data`.`licenseid`,
    `aggregated_data`.`entityid`,
    `aggregated_data`.`entity`,
    `aggregated_data`.`termid`,
    `aggregated_data`.`term`,
    `aggregated_data`.`tweetid`,
    `aggregated_data`.`retweetid`,
    `aggregated_data`.`tweet`,
    `aggregated_data`.`taxonomyid`,
    `aggregated_data`.`taxonomy`,
    `aggregated_data`.`categoryid`,
    `aggregated_data`.`category`,
    `aggregated_data`.`score`,
    `aggregated_data`.`scoremanual`,
    `aggregated_data`.`tweetdate`,
    `aggregated_data`.`placeid`,
    `aggregated_data`.`place`,
    `aggregated_data`.`userid`,
    `aggregated_data`.`user`,
    `aggregated_data`.`userlocation`,
    `aggregated_data`.`country`,
    `aggregated_data`.`language`
FROM `sobserver`.`aggregated_data`
;
```

La ejecución del script se realizará por consola de Unix. Ésta es la salida:

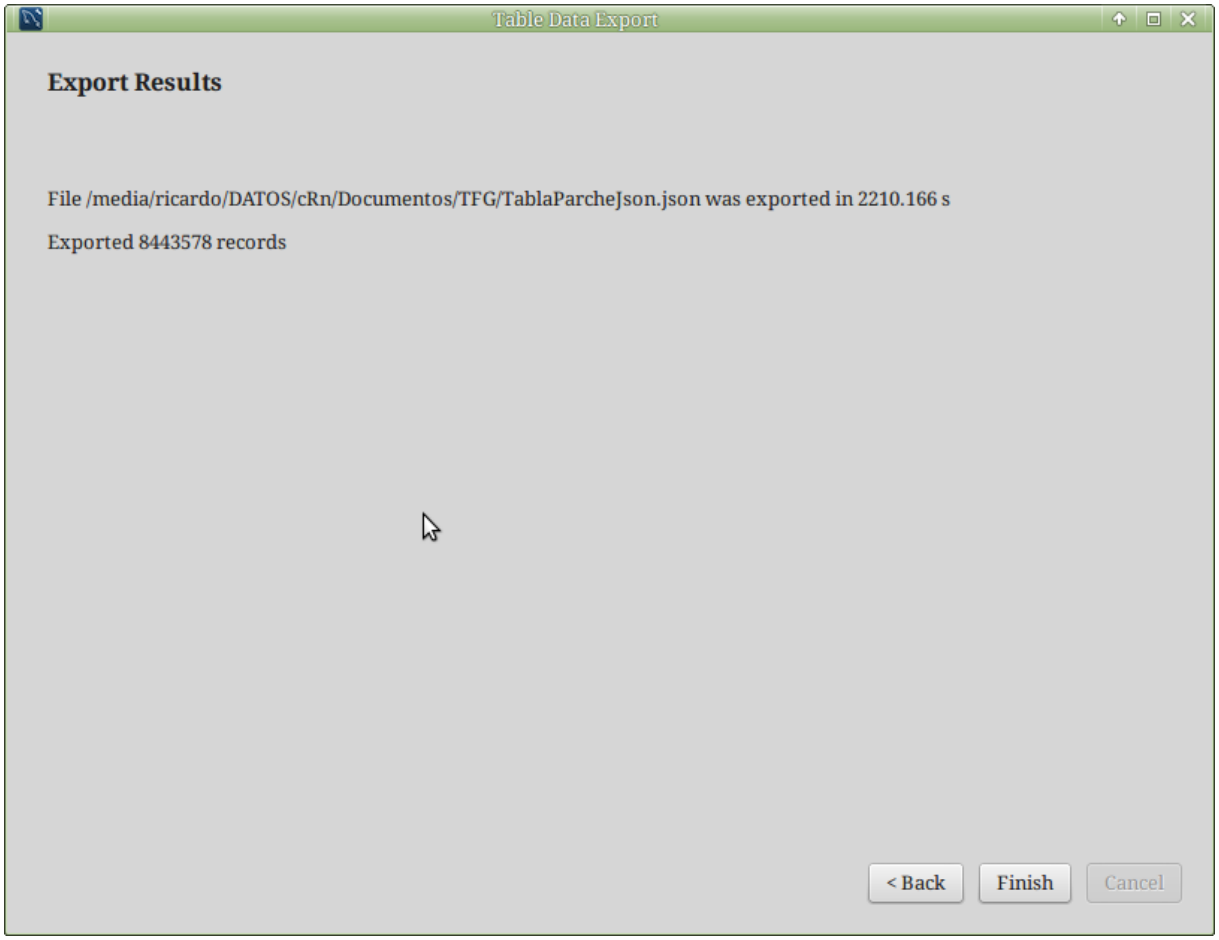


Captura 3. Desnormalización de la BD. Creación de la tabla

Ha tardado unos 33 min y 43 seg aprox. en crearse la tabla parche. Comprobamos que, efectivamente, tiene 8443578 filas

Anexo 7: Importación a MongoDB

Para la exportación de la tabla parche se ha utilizado MySQL WorkBench, la utilidad Table Data Export Wizard



Captura 4. Importación a MongoDB. Tiempo Table Data Export a .json

Se puede observar que ha exportado 8443578 registros correctamente en unos 37 min. Sin embargo el archivo ocupa 4,5 GB, que es demasiado para mongoimport, por lo que hay que dividirlo con el comando *split*.

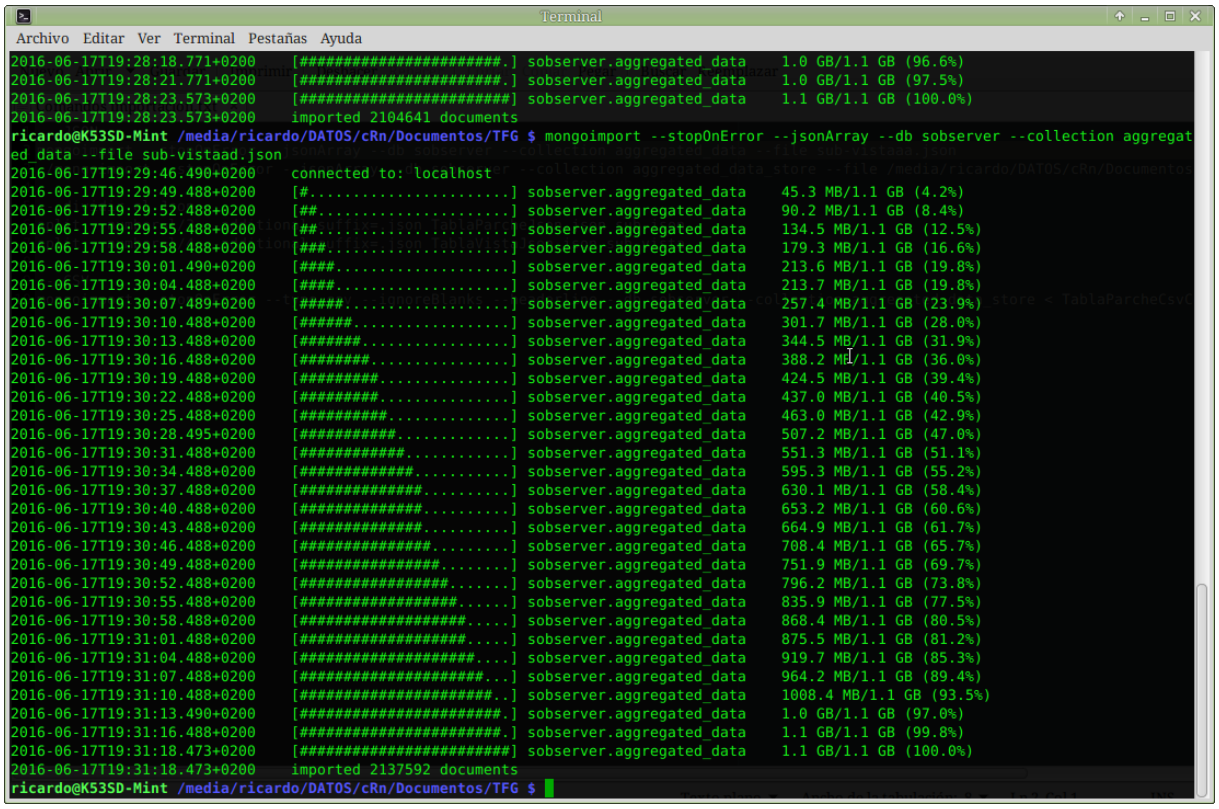
El comando utilizado para dividir el archivo .json ha sido:

```
# split --number=l/4 --additional-suffix=.json
TablaParhceJson.json sub-parche
```

Este genera 4 archivos ‘sub-parcheaX’, siendo X ‘a’,’b’, ‘c’ o ‘d’.

Hay que formatear los archivos para que estén en un formato correcto para JSON, eso significa añadirle '[' o ']' al principio y/o al final, según convenga y quitar las ',' finales de los primeros archivos. Esto se hace a mano con un editor de textos y, a pesar de ser archivos más pequeños (unos 1,1GB aprox.) sigue necesitando mucha RAM para la tarea.

Una vez formateado el documento usaré mongoimport para la inserción de los archivos. Un ejemplo de salida del mongoimport:



Captura 5. Importación a MongoDB. Tiempo mongoimport de un archivo

Una vez importada comprobamos con un count que tiene, efectivamente, 8443578 documentos importados,

Anexo 8: Consultas en MongoDB

Consulta a MongoDB mediante el script *script-SQL-parche.js*:

```
var output = db.aggregated_data.find()
          .toArray();
```

Consulta a MongoDB con explain("executionPlan"), script llamado *script-stats.js*:

```
var output = db.aggregated_data.find()
          .explain("executionStats");
printjson("executionTimeMillis: " +
output.executionStats.executionTimeMillis);
```

Consulta a MongoDB con filtros simples, scripts *script-SQL-parche-where.js* y *script-SQL-parche-where-alicante.js*:

```
var output = db.aggregated_data.find(
          { term : 'Javier Echenique'})
          // { term : #Alicante'})
          .toArray();
```

Consulta a MongoDB con explain("executionPlan") con filtros simples, scripts *script-stats-where* y *script-stats-where-alicante.js*:

```
var output = db.aggregated_data.find(
          { term : 'Javier Echenique'})
          // { term : '#Alicante'})
          .explain("executionStats");
printjson("executionTimeMillis: " +
output.executionStats.executionTimeMillis);
```

Consulta a MongoDB con filtro compuesto, script *script-SQL-parche-where-compuesto.js*:

```
var output = db.aggregated_data.find(
          { term :
               { $in : [
               /pp/i,
               /ppopular/i,
               /psoe/i,
```

```
                /ciuda/i,
                /compromis/i,
                /UPyD/i
                ]}
          })
          .explain("executionStats");
printjson("executionTimeMillis: " +
output.executionStats.executionTimeMillis);
```

Consulta a MongoDB con explain("executionPlan"), script llamado *script-stats.js*:

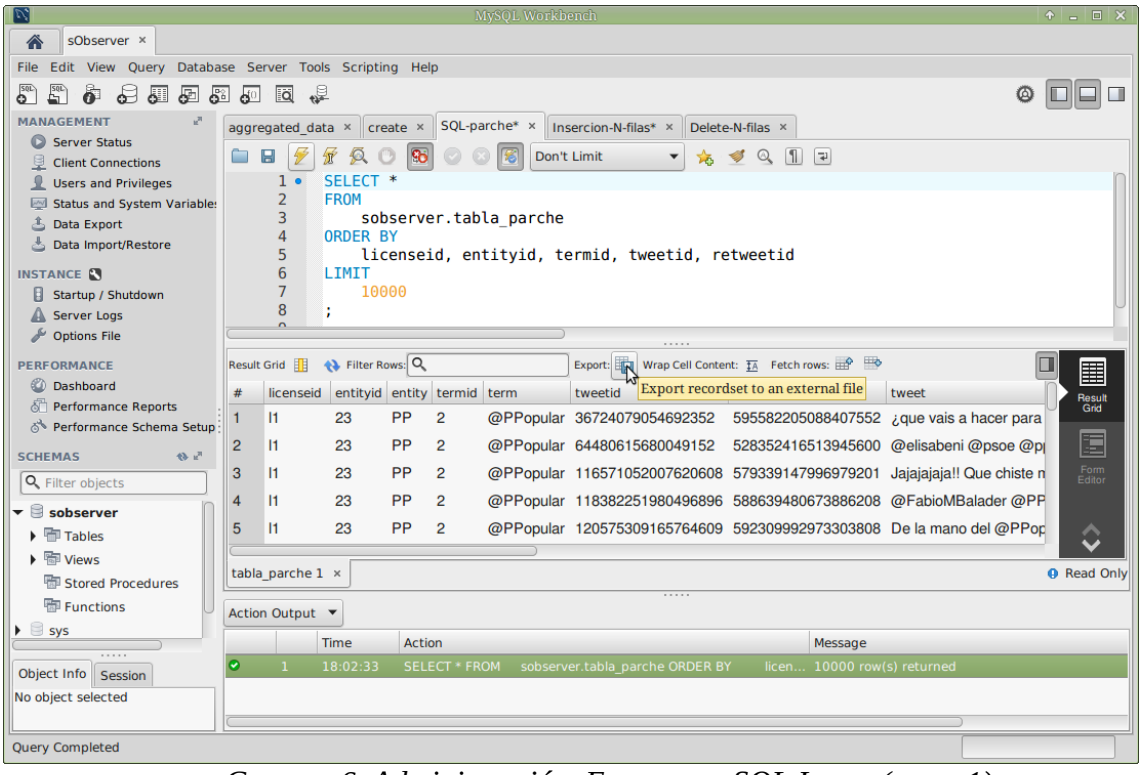
```
var output = db.aggregated_data.find()
          .explain("executionStats");
printjson("executionTimeMillis: " +
output.executionStats.executionTimeMillis);
```

Anexo 9: Consultas de administración

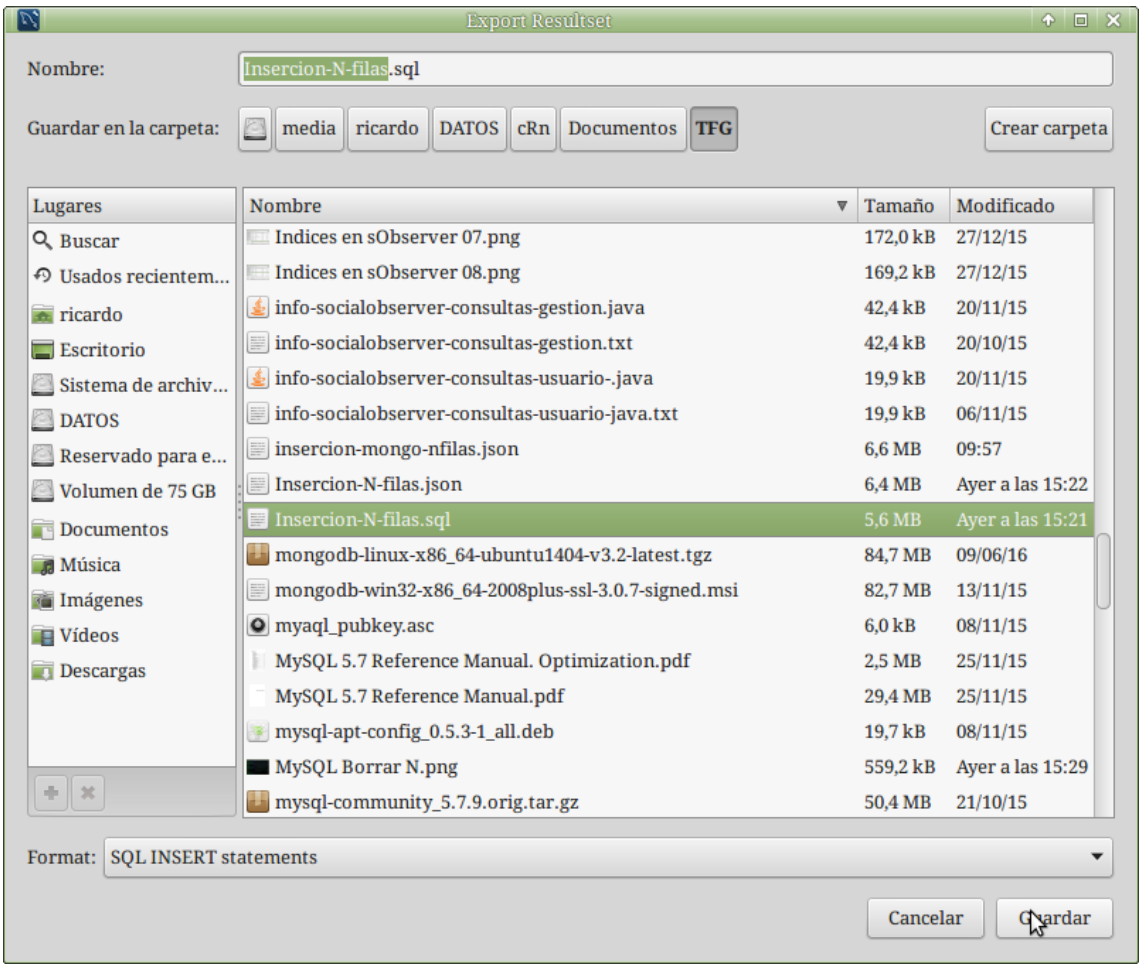
Para probar la inserción de N filas de datos voy a realizar primero una consulta con la que guardaré las N filas en un archivo para luego poder volver a introducirlas. Deberé usar un ORDER BY o .sort() para asegurar que siempre inserto los mismos datos, lo que ralentiza las consultas

MySQL

Para generar un archivo que me guarde las filas que voy a borrar utilizo el MySQLWorkbench, que me permite guardar las filas consultadas en un archivo .sql con los comandos INSERT para cada una



Captura 6. Administración. Exportar a SQL Insert (parte 1)



Captura 6. Administración. Exportar a SQL Insert (parte 1)

Una vez guardado procedo a borrar las filas consultadas con la siguiente consulta. Es necesario el order by para asegurar que sean las primeras filas guardadas. El script se llamará *borrar.sql*:

```
SET SQL_SAFE_UPDATES = 0;
DELETE
FROM
    sobserver.tabla_parche
ORDER BY
     licenseid, entityid, termid, tweetid, retweetid
LIMIT
     10000
;
```

Luego volveré a insertar las filas con el script creado por el MySQL Workbench. Para ello utilizo la consola de Linux para ejecutar dicho archivo .sql.

MongoDB

En Mongo tengo que utilizar un script en JavaScript para guardar las filas al que he llamado *script-salida.js*. Igual que en MySQL es importante realizar un *.sort()* para asegurarse que son las 10.000 primeras filas

```
var output = db.aggregated_data.find()
          .limit(10000)
          .sort({
               'licenseid' : 1,
               'entityid' : 1,
               'termid' : 1,
               'tweetid' : 1,
               'retweetid' : 1
               })
          .toArray();
printjson(output);
```

Ahora para borrar N filas usaré otro script llamado *script-borrar.js*. Al igual que antes, he de seguir el orden

```
var arrayEliminar = db.aggregated_data.find()
          .limit(10)
          .sort({
               'licenseid' : 1,
               'entityid' : 1,
               'termid' : 1,
               'tweetid' : 1,
               'retweetid' : 1
               })
          .toArray()
          .map(function(doc){
               return doc._id;
               });
var output = db.aggregated_data.remove({
               _id : {$in : arrayEliminar}
               });
print(output);
```

La inserción se puede llevar a cabo bien con el comando insert() y un array de json, por lo que me tendría que crear un script *insercion-mongo-nfilas.js* para utilizar el array de json que creé anteriormente desde la consola de Unix:

```
var output = db.aggregated_data.insert(
[
     {
          "_id" : ObjectId("576c00e8c726895bf58ce94c"),
          "licenseid" : "l1",
.
.
.
]);
print(output);
```

*Nota: Se han omitido los 10.000 registros para mostrar únicamente el comando *.insert()*

También puedo usar el comando mongoimport utilizando el mismo archivo sin formatear.

El comando mongoimport tarda algo menos que insert() en volver a insertar los documentos, aunque en la documentación no recomiendan utilizar este método para la inserción de documentos

Anexo 10: Índices

En MySQL creo los índices utilizados en las consultas a la tabla Vista, pero existe un problema, ningún índice puede ser unique en la tabla_parche porque hay valores repetidos; además también tenemos el problema de las fechas de algunos tweets. El script *indices.sql* de creación del Índices queda así:

```
SET SQL_MODE='ALLOW_INVALID_DATES';
CREATE INDEX -- no puede ser unique
     ix_licenseid
ON
     sobserver.tabla_parche
     (licenseid)
USING
     BTREE
;
CREATE INDEX
     ix_entityid
ON
     sobserver.tabla_parche
     (entityid)
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
     ix_entity
ON
     sobserver.tabla_parche
     (entity)
USING
     BTREE
;
CREATE INDEX
     ix_termid
ON
     sobserver.tabla_parche
     (termid)
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
     ix_term
ON
     sobserver.tabla_parche
     (term)
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
```

```
     ix_tweetid
ON
     sobserver.tabla_parche
     (tweetid)
USING
     BTREE
;
CREATE INDEX
     ix_retweetid
ON
     sobserver.tabla_parche
     (retweetid)
USING
     BTREE
;
CREATE INDEX
     ix_taxonomyid
ON
     sobserver.tabla_parche
     (taxonomyid)
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
     ix_taxonomy
ON
     sobserver.tabla_parche
     (taxonomy)
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
     ix_categoryid
ON
     sobserver.tabla_parche
     (categoryid)
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
     ix_category
ON
     sobserver.tabla_parche
     (category)
USING
     BTREE
;
CREATE INDEX
     ix_score
ON
     sobserver.tabla_parche
     (score)
```

```
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
     ix_userid
ON
     sobserver.tabla_parche
     (userid)
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
     ix_user
ON
     sobserver.tabla_parche
     (user)
USING
     BTREE
;
CREATE INDEX -- no puede ser unique
     ix_place
ON
     sobserver.tabla_parche
     (place)
USING
     BTREE
;
```

En Mongo utilizo el script *script-indices.js* para la creación de los mismos:

```
var output = db.aggregated_data.createIndex({ 'licenseid' : 1
}, { name : "ix_licenseid" });
printjson(output);
output = db.aggregated_data.createIndex({ 'entityid' : 1 }, {
name : "ix_entityid" });
printjson(output);
output = db.aggregated_data.createIndex({ 'entity' : 1 }, {
name : "ix_entity" });
printjson(output);
output = db.aggregated_data.createIndex({ 'termid' : 1 }, {
name : "ix_termid" });
printjson(output);
output = db.aggregated_data.createIndex({ 'term' : 1 }, {
name : "ix_term" });
printjson(output);
output = db.aggregated_data.createIndex({ 'tweetid' : 1 }, {
name : "ix_tweetid" });
printjson(output);
output = db.aggregated_data.createIndex({ 'retweetid' : 1 },
```

```
{ name : "ix_retweetid" });
printjson(output);
output = db.aggregated_data.createIndex({ 'taxonomyid' : 1 },
{ name : "ix_taxonomyid" });
printjson(output);
output = db.aggregated_data.createIndex({ 'taxonomy' : 1 }, {
name : "ix_taxonomy" });
printjson(output);
output = db.aggregated_data.createIndex({ 'categoryid' : 1 },
{ name : "ix_categoryid" });
printjson(output);
output = db.aggregated_data.createIndex({ 'category' : 1 }, {
name : "ix_category" });
printjson(output);
output = db.aggregated_data.createIndex({ 'score' : 1 }, {
name : "ix_score" });
printjson(output);
output = db.aggregated_data.createIndex({ 'userid' : 1 }, {
name : "ix_userid" });
printjson(output);
output = db.aggregated_data.createIndex({ 'user' : 1 }, {
name : "ix_user" });
printjson(output);
output = db.aggregated_data.createIndex({ 'place' : 1 }, {
name : "ix_place" });
printjson(output);
```

Bibliografía y enlaces

[0] Blog Internetría. NoSQL. Artículo introductorio a esta tecnología

http://www.internetria.com/blog/2013/05/08/nosql/

[1] Types of NoSQL Databases. Tipos de NoSQL y descripciones

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

[2] KDB Database intro. Esquema relacional y por columnas

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

[3] Univ. De la Plata. Bases de Datos NoSQL: Escalabilidad y alta disponibilidad a través de patrones de diseño. Críticas y Ventajas de las tecnologías NoSQL

http://sedici.unlp.edu.ar/bitstream/handle/10915/36338/Documento_completo.pdf?sequence=5

[4] Ranking Tecnologías BBDD

http://db-engines.com/en/ranking

[5] Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. Demostración del teorema CAP

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf

[6] Genbeta. NoSQL: Clasificación de las BBDD según el teorema de CAP

http://www.genbetadev.com/bases-de-datos/nosql-clasificacion-de-las-bases-de-datos-segun-el-teorema-cap

[7] Cassandra structured Storage over a P2P Network. Comparativa de rendimiento entre MySQL y Cassandra en 50 GB de datos

http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf

[8] No to SQL? Anti-database movement gains steam. Artículo en ComputerWorld sobre ventajas de las NoSQL

http://www.computerworld.com/article/2526317/database-administration/no-to-sql--anti-database-movement-gains-steam.html?page=2

[9] Getting Real about NoSQL and the SQL-Isn't-Scalable Lie. Artículo de Dennis Forbes sobre NoSQL

https://dennisforbes.ca/index.php/2010/03/02/getting-real-about-nosql-and-the-sql-isnt-scalable-lie/

[10] Artículo. MySQL and MemCached: End of a Era?. Todd Hoff sobre MySQL y MemCache

http://highscalability.com/blog/2010/2/26/mysql-and-memcached-end-of-an-era.html

[11] Artículo. Cassandra @ Twitter: An interview with Ryan King. Entrevista sobre el uso de Cassandra en Twitter

http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king

[12] Debunking the NoSQL Hype. Oracle White Paper. Critica a las NoSQL

https://billycripe.files.wordpress.com/2011/10/debunking-nosql-twp-399992.pdf

[13] Comparación y recomendaciones diferentes NoSQL

http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis

[14] MySQL Doc. Optimization

http://dev.mysql.com/doc/refman/5.7/en/optimization.html

[15] MySQL Doc. Mysqlslap. Load Emulation Client

http://dev.mysql.com/doc/refman/5.7/en/mysqlslap.html

[16] Wikipedia. Comando time de Unix

https://es.wikipedia.org/wiki/Time_(Unix)

[17] MongoDB Doc. Mongoimport

https://docs.mongodb.com/manual/reference/program/mongoimport/

[18] MongoDB Doc. Evaluate the Performance of a Query

https://docs.mongodb.com/manual/tutorial/analyze-query-plan/#evaluate-the-performance-of-a-query

[19] MongoDB Doc. Explain Results. executionTimeMillis

https://docs.mongodb.com/manual/reference/explain-results/#explain.executionStats.executionTimeMillis

[20] MySQL Doc. APT Repo Setup

http://dev.mysql.com/doc/mysql-apt-repo-quick-guide/en/#apt-repo-setup

[21] MySQL Doc. Checking GPG Signature

http://dev.mysql.com/doc/refman/5.7/en/checking-gpg-signature.html

[22] Procesamiento del Lenguaje Natural, Revista nº 55, pp. 195-198. Septiembre de 2015. F. Agulló, A. Guillén, Y. Gutiérrez, P. Martínez-Barco.

[23] Stackexchange. When to use Views in MySQL

http://dba.stackexchange.com/questions/16372/when-to-use-views-in-mysq

[24] Percona Performance Blog. MySQL VIEW as performance troublemaker. Peter Zaitsev. Percona Database, 2007

https://www.percona.com/blog/2007/08/12/mysql-view-as-performance-troublemaker/

Otros enlaces consultados

Artículo. Perfilar consultas en MySQL. Medir rendimiento de las consultas

http://www.xaprb.com/blog/2006/10/12/how-to-profile-a-query-in-mysql/

Uso correcto del comando mysqlslap

https://www.digitalocean.com/community/tutorials/how-to-measure-mysql-query-performance-with-mysqlslap

Paper. NoSQL Databases. Como el de la Universidad de la Plata, pero en inglés y con las fuentes

http://www.christof-strauch.de/nosqldbs.pdf

Artículo en Computer Magazine. CAP twelve years later: How the Rules have changed

http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

BDs no relacionales YCSB, Hadoop, etc. Univ. Navarra

http://academica-e.unavarra.es/bitstream/handle/2454/10203/629103.pdf?sequence=1

Introducción a NoSQL y grupos. La Pastilla Roja

http://lapastillaroja.net/2012/02/nosql-for-non-programmers/

Análisis de motores NoSQL 2015. Univ. Cat. Colombia

http://repository.ucatolica.edu.co:8080/jspui/bitstream/10983/2470/1/TRABAJO%20DE%20GRADO.pdf

When to use views in MySQL?

http://dba.stackexchange.com/questions/16372/when-to-use-views-in-mysql