DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico #3

23 de Diciembre de 2013 — Sistemas operativos

Integrante	LU	Correo electrónico
Leandro Matayoshi	79/11	leandro.matayoshi@gmail.com
Martin Santos	413/11	martin.n.santos@gmail.com
Alexander Szyrej	642/11	luzbelito.as@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

MongoDB es una base de datos cross-platform orientada a documentos, clasificada como NoSQL y que implementa el paradigma MapReduce para análisis de datos.

El objetivo de este trabajo consiste en comprender e implementar distintos algoritmos de análisis de datos sobre una pequeña base de datos de *Rededit* y efectuar un análisis de escalabilidad y performance de la arquitectura utilizando MongoDB para el soporte de los datos y una arquitectura basada en maquinas virtuales.

Para esto último se tienen en cuenta distintas alternativas de $HaaS$ (Hardware as a Service) para hacer el despliegue de las máquinas virtuales, evaluando performance considerando las capacidades de MongoDB.

2. Desarrollo

2.1. Una única instancia de MongoDB. Prueba de concepto MapReduce

En esta sección implementaremos distintos algoritmos de análisis de datos para operar sobre una pequeña colección de posts de *Rededit*, con un tamaño aproximado de 17MB. Dado el pequeño volumen de datos con el cual trabajaremos dispondremos de una única instancia de MongoDB (no sharding).

Queremos:

- Analizar la reacción de la comunidad en promedio frente a posts de otros usuarios.
- Calcular un promedio de comentarios por submission.
- Encontrar el usuario con mayor score.
- Determinar en qué horario se producen más comentarios, y en cuál se comenta menos.

2.1.1. Comunidad Upvoter-Neutral-Downvoter

Queremos analizar el comportamiento de la comunidad en base a una reducida colección de posts. Para determinar el vote-trend implementamos un único MapReduce sobre dicha colección.

La función **map** genera *cuatro* claves:

- $posts - upvoted$ obtiene un nuevo valor '1' por cada post donde los comentarios positivos superan en cantidad a los negativos.
- $posts - downvoted$, a la inversa que el anterior, obtiene un nuevo valor '1' por cada post donde los downvotes superan a los upvotes.
- $neutral - posts$ recibe un valor '1' por cada post donde los downvotes igualan en cantidad a los upvotes.
- $vote - trend$ recibe un '1' por cada $post - upvoted$ y un '-1' por cada $post - downvoted$.

La función **reduce** se reduce en sumar todos los valores para cada clave. De esta forma contabilizamos los posts de votos positivos, negativos y neutrales y, al mismo tiempo, calculamos el $vote - trend$. Si este último resultado es positivo la comunidad podría definirse como *upvoter*. Si es negativo clasificaría como *downvoter* o *neutral* si $vote - trend$ es exactamente cero.

Con 21270 $downvoted - posts$, 106092 $upvoted - posts$ y 4941 $neutral - posts$, el $vote - trend$ da como resultado 84821, lo cual indica que la comunidad es upvoter.

2.1.2. Comentarios por submission

Se desea encontrar en promedio cuántos comentarios se realizan por cada submission. Para ello se implementa un único MapReduce sobre la colección de posts en conjunto con una función finalize para calcular el promedio final.

La función **map** genera una única clave *comments* donde por cada post se genera un valor compuesto por el número de comentarios del mismo y un field *count* para contabilizar la cantidad de posts.

La función **reduce** hace la suma total de todos los comentarios y los posts (count).

Luego de que se efectuan los reduce necesarios, se procede con la función **finalize**, la cual calcula el promedio de comentarios teniendo ya el número total de comentarios (sum($comments$)) y el número de posts (sum($count$)).

Como resultado obtuvimos que, en promedio, se efectuan alrededor de 39 comentarios por submission.

2.1.3. Usuario con más puntaje

Queremos hallar el usuario cuya suma de los scores de sus posts supere la de los demás. Para ello encadenamos dos MapReduce: el primero sobre la colección posts y el segundo sobre la colección generada por el primer MapReduce.

El primer MapReduce se resume en generar una clave por cada usuario, cada una emitida conjuntamente con su score. Luego se suman todos los scores para cada usuario y de esa manera se obtiene el score total de cada uno.

El segundo MapReduce consiste en encontrar, dados los scores totales de cada uno de los usuarios, aquel que suma más que el resto. Para ello se emite una única clave 'max', con valores asociados compuestos por una tupla usuario-score por cada usuario de nuestra red. Luego durante el reduce tomamos el usuario con el máximo score dentro de nuestra lista de values.

Encontramos que el usuario con mayor score es 'nombre vacio' con 1311814 pts.

2.1.4. Horario más y menos activo

Queremos encontrar el horario en el cual la comunidad participa más comentando posts, es decir, determinar de alguna manera el mejor horario para postear. Análogamente, hallamos el horario en el cual menos se comenta. Para ello utilizamos nuevamente dos MapReduce encadenados.

El primer MapReduce se resume en generar una clave por hora, es decir un total de 24 claves. Para ello se utiliza la función $substring(int\ idx0,\ int\ idxn)$ para obtener del $raw - time$ sólo la hora en la cual fue subido el post. Cada una de ellas viene acompañada por un valor igual al número de comentarios de dicho post. El reduce sólo calcula la suma de todos los comentarios en una determinada hora.

El segundo MapReduce, semejante al punto anterior, consiste en encontrar la hora cuyo número de comentarios es máximo, y aquella que minimiza dicho valor.

Encontramos que los posts subidos de 00:00 a 01:00 son aquellos que más comentarios suman, con un total de 632402 comentarios. Luego de 01:00 a 02:00 es el horario en el cual menos comentarios hay, con tan solo 91634, lo cual parece ser razonable si pensamos que el momento pico de actividad es antes de ir a dormir.

2.1.5. ¿Dónde montar reddit?

Debemos realizar un manejo eficiente de la base de datos de *Rededit*. El siguiente esquema representa las distintas alternativas que podemos tomar:

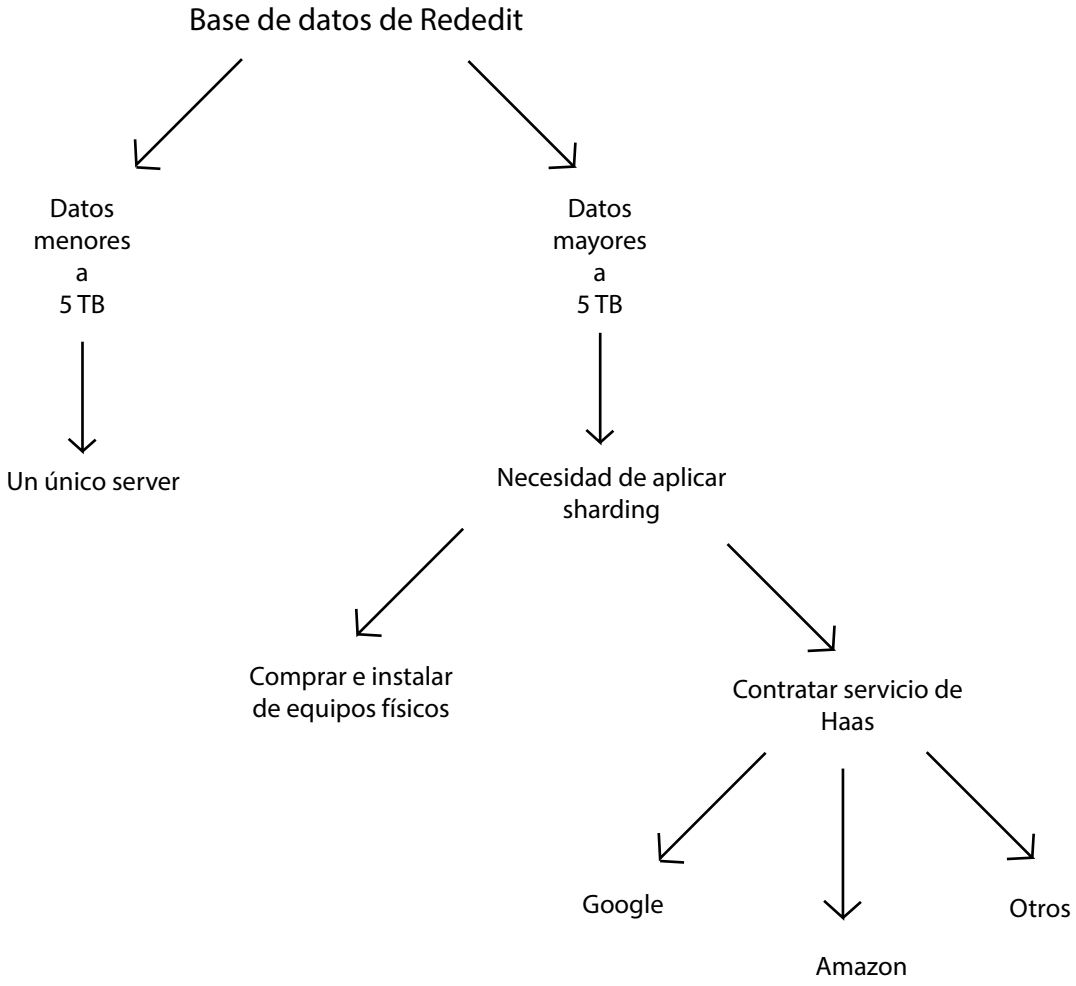

Figura 1: Distintas alternativas a la hora de montar Reddit

Si la base de datos es pequeña, como por ejemplo menor a 5TB y no se espera que crezca demasiado nos conviene tener un único servidor funcionando. Sin embargo, a medida que el volumen de datos crece empieza a resultar necesario aplicar sharding sobre nuestros datos para mantener la performance del sistema. En ese momento nos planteamos 2 alternativas. ¿Montamos la aplicación sobre hardware propio o contratamos web-services de alguno de los proveedores como Amazon o Google? En las siguientes secciones vamos a analizar cada uno de los casos.

3. Arquitectura elegida para MongoDB

3.1. Sharding, conceptos generales

Sharding es el proceso de almacenar datos a lo largo de múltiples máquinas y es la herramienta a través de la cual MongoDB combate el crecimiento de la demanda de datos. A medida que el tamaño de los mismos aumenta, una única máquina puede no ser suficiente para almacenar y proveer rates de lectura y escritura aceptables. Sharding resuelve el problema de escalamiento horizontal. Se agregan más máquinas para soportar el crecimiento de los datos y la demanda de usuarios de la base de datos.

Por el contrario, escalamiento vertical agrega más CPU y memoria para incrementar la capacidad. Escalar a través del aumento de los recursos de una única máquina tiene sus limitaciones: Los sistemas de alta

performance con gran cantidad de CPU's y memoria RAM son desproporcionadamente más costosos que un conjunto de sistemas pequeños. En suma, los proveedores basados en la nube únicamente proveen instancias pequeñas a los usuarios. Como resultado, hay un tope máximo en la capacidad de escalar a nivel vertical.

Por el contrario, sharding o escalamiento horizontal, divide el set de datos y lo distribuye a lo largo de múltiples servidores. Cada shard es una base de datos independiente, y colectivamente, los shards conforman una única lógica base de datos.

MongoDB sharding-guide.

3.2. Sharding en MongoDB

MongoDB soporta sharding a través de estructuras llamadas *sharded clusters*, que cuentan con los siguientes componentes:

- Shards o *mongods*
- Query routers o *mongos*
- Config servers

Shards o *mongods* son los nodos de almacenamiento de datos. Cada nodo está compuesto por un conjunto replicado (*replica set*). Los Query routers o *mongos* actúan de intermediarios entre los pedidos de los clientes y los shards correspondientes que contienen esos datos específicos. Un cluster puede contener más de un query router (y es conveniente que así) sea, para disminuír cantidad de pedidos que debe manejar un único nodo. Finalmente, los Config servers almacenan la metadata del cluster: principalmente, el mapeo de los datos en cada uno de los shards.

3.3. Decisiones de arquitectura

Para establecer la base de datos de *Rededit* tomamos las siguientes decisiones de arquitectura:

3.3.1. *Shards y Replica sets*

Cada *shard* está constituida por un *replica set*: un conjunto de nodos que mejoran la disponibilidad de los datos del servidor en base a la redundancia. Existen tres tipos de miembros:

Primario Recibe las operaciones de escritura.

Secundario Replican las mismas operaciones efectuadas por el primario para mantener el set de datos consistente.

Árbitro Opcional. No almacena datos. Simplemente participa en la elección de un nuevo nodo primario en los casos en donde el actual se encuentra inhabilitado.

Decidimos adoptar una de las configuraciones típicas propuestas en *MongoDB-replication-guide*: Una estructura conformada por tres nodos, todas ellas almacenando datos:

Uno primario

Dos secundarios Ambos son candidatos a volverse primarios luego de una elección, en el caso de una falla en el nodo primario.

Ningún árbitro

Mediante esta elección disponemos de 2 copias completas del set de datos además del primario. Provee tolerancia a fallos en el nodo primario, y buena velocidad de respuesta debido a la redundancia.

3.3.2. Config servers

El cluster va a contar con 3 servidores de configuración (Cantidad sugerida por *MongoDB-sharding-guide*).

3.4. *Mongos y shards*

El cluster cuenta inicialmente con 2 *mongos* o query routers y 2 shards (cluster minimal de MongoDB). La idea es ir incrementando la cantidad de nodos a medida que aumente la cantidad de usuarios de la aplicación, y por lo tanto, las necesidades de nuestro servidor para satisfacer exitosamente todos los pedidos.

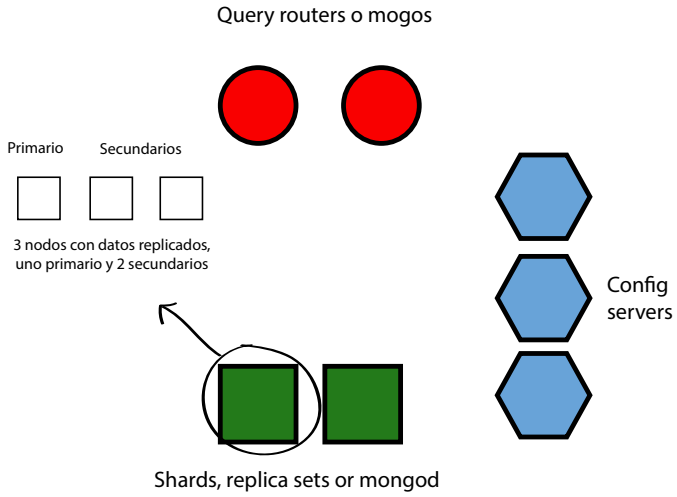

Figura 2: Arquitectura inicial con la que empieza la base de datos de mongoDB

3.5. Incremento de Mongos y shards

Finalmente hemos tomado la siguiente decisión para determinar el momento de aumentar o disminuír la cantidad de nodos de nuestro cluster.

Cuando los datos almacenados alcanzan $\frac{3}{4}$ de la capacidad total de almacenamiento de nuestro servidor (suma de las capacidades de los shards) duplicamos la cantidad de shards del cluster.

Si por el contrario, los datos almacenados disminuyen a menos de $\frac{1}{4}$ de la capacidad total (por ejemplo, debido a la baja del sistema de muchos de los usuarios), entonces reducimos a la mitad la cantidad de shards.

Algo similar sucederá con la cantidad de *mongos* o query routers del cluster. En cuanto el sistema detecte que el tiempo de respuesta ante los pedidos de los usuarios es lento duplicamos la cantidad de routers, ya que la cntidad actual no logran satisfacer las demandas de los clientes. En el caso de que haya gran cantidad de query routers que no procesan ningún pedido (los pedidos se distribuyen de manera uniforme entre los *mongos*, cuando la base está bien balanceada), es momento de reducir a la mitad la cantidad de query routers.

4. Eligiendo un web service

4.0.1. Breve descripción de cloud computing y web services

”Procesamiento en la nube” o *cloud computing* es una expresión que hace referencia a la posibilidad de realizar gran variedad de cómputos utilizando un conjunto de muchas máquinas conectadas entre sí (cluster). El término ”nube” intenta describir la abstracción del cluster en sí mismo como componente físico.

El usuario (donde usuario puede ser un individuo, una empresa o cualquier otro tipo de organización) es consciente de que los cómputos que él necesita realizar se efectúan en algún sitio con los componentes de hardware necesarios pero no requiere saber donde. El procesamiento es realizado por alguna de las computadoras, de alguno de los clusters de la nube.

Este sistema intenta maximizar la efectividad de los cómputos de todos los usuarios y disminuír el precio al compartir los recursos. Los mismos no solo son compartidos por distintos usuarios sino que también la cantidad de recursos asignados para un mismo usuario varía según distintos factores (como por ejemplo la demanda de procesamiento en un determinado período de tiempo), por lo que el sistema se vuelve muy

flexible y permite que las aplicaciones que utilizan la nube escalen a medida que crezcan en tamaño (volumen de datos o de cómputo, cantidad de usuarios, etc).

Paralelamente al desarrollo de "cloud computing", empresas como Amazon, Google, dotCloud, Red Hat, etc. comenzaron a ofrecer lo que conocemos como "web servicies": ofrecer al usuario la posibilidad de utilizar el poder de cómputo de sus computadoras y clusters a cambio de una determinada suma de dinero. Si bien el servicio es pago, incluye todas las ventajas de la computación en la nube. El servicio es muy flexible y logra que los clientes paguen únicamente por los recursos que utilizan (y no más que eso). Para cumplir con estos objetivos utilizan el montaje de máquinas virtuales sobre computadoras físicas. De esta manera son las máquinas virtuales las que son ofrecidas a los clientes. Estos últimos pueden pagar por más máquinas o menos dependiendo de la demanda de sus aplicaciones. Al mismo tiempo es posible pagar por más o menos tiempo para cada una.

Ya hemos definido la arquitectura del cluster de MongoDB que necesitábamos para nuestra base de datos. Ahora es momento de analizar distintas alternativas de web services y elegir la que mejor se adapte a nuestras necesidades.

4.0.2. Amazon

Ec2 ("elastic computing cloud") es una rama muy importante dentro de "Amazon web servicies" . El mismo está basado en AMI's (Amazon Machine Images). Cada machine image puede generar una o miles de instancias, en donde cada instancia es equivalente a una de las máquinas virtuales mencionadas anteriormente.

Según la documentación oficial de MongoDB (http://docs.mongodb.org/ecosystem/platforms/amazon-ec2/), mongo es compatible con Amazon Ec2. En suma, existen 3 AMI's oficiales:

MongoDB 2.4 with 1000 IOPS - data: 200 GB @ 1000 IOPS, journal: 25 GB @ 250 IOPS, log: 10 GB @ 100 IOPS

MongoDB 2.4 with 2000 IOPS - data: 200 GB @ 2000 IOPS, journal: 25 GB @ 250 IOPS, log: 15 GB @ 150 IOPS

MongoDB 2.4 with 4000 IOPS - data: 400 GB @ 4000 IOPS, journal: 25 GB @ 250 IOPS, log: 20 GB @ 200 IOPS

Todas ellas tienen una arquitectura basada en 64-bits, Linux/Unix como sistema operativo, un espacio reservado para almacenar la información de journal y log de la base de datos. Además cuentan con EBS ("elastic block store"), que posibilitan el almacenamiento permanente de los datos más allá de la vida de la instancia. Las máquinas son muy eficientes en el almacenamiento de datos con una gran cantidad operaciones de entrada y salida por segundo y permiten aprovechar y explotar todas las funcionalidades de MongoDB, como búsqueda completa por índices (incluídos compuestos y geoespaciales)

Las máquinas que nos ofrecen son del tipo: M3 (propósito general) o High-Memory (optimizadas en el uso de memoria RAM).

Recordando la configuración inicial de nuestra arquitectura, requeríamos de 2 query routers, 3 config servers y 2 shards. Cada shard estaba compuesto a su vez por 3 nodos: uno primario y 2 secundarios. Por lo tanto, con esta alternativa deberíamos comenzar con 6 AMI's MongoDB 2.4 with 4000 IOPS de tipo de instancia High-Memory (para cada uno de los nodos que van a contener los datos). En suma deberíamos agregar 5 ami's de propósito general para los query routers y config servers: Las instancias del tipo M1 y M3 proveen un balance entre cómputo, memoria y recursos de red.

Otra posibilidad en vez de utilizar las máquinas proupestas por el manual para los nodos de los shrads consiste en utilizar instancias optimizadas para almacenamiento: I2 y HI1. (http://aws.amazon.com/ec2/instance-types/). Las mismas están optimizadas para obtener una gran densidad de almacenamiento con bajo costo, y una gran performance en operaciones de I/0. Los discos son SSD. Estas instancias funcionan muy bien con bases de datos noSQL, como mongoDB.

Se incluyen a continuación algunos detalles y precios de instancias I2, m2(memory optimized) y m3 (general purplse) disponibles:

Instance family	Type	Arch	vCPU	ECU	Memory(GB)	Storage (GB)	EBS-opt	Network perf
general purpose	m3.xlarge	64-bits	4	13	15	2x40 SSD	yes	Moderate
memory optimized	m2.xlarge	64-bits	2	6.5	17.1	1x420	no	Moderate
storage optimized	i2.xlarge	64-bits	4	14	30.5	1x800 SSD	yes	Moderate

La elección de Amazon como proveedor de web-services nos posibilita la la incorporación de S3, en el caso de que queramos combinar mongoDB con otro tipo de tecnologías : Simple storage service. Una interfaz utilizada para almacenar y envíar datos a cualquier lugar de la web. Los procedimientos se realizan de forma segura, confiable, rápida y con precios accesibles de forma tal que nuestra aplicación pueda escalar a medida que el volumen de transferencias sea cada vez mayor. (http://aws.amazon.com/s3/)

Finalmente podemos utilizar Ec2 Auto-Scaling con las condiciones que habíamos definido anteriormente (duplicar la cantidad de AMI's en los casos en donde los datos superen los $\frac{3}{4}$ o disminuyan a menos de $\frac{1}{4}$, o cuando sea necesario aumentar o disminuir la cantidad de query routers). De esta manera podemos aprovechar la flexibilidad del servicio pagando únicamente por los recursos que efectivamente estamos consumiendo: http://aws.amazon.com/autoscaling/.

4.0.3. Google compute engine

De forma similar a Amazon, y compartiendo las mismas ideas y beneficios, Google es otra de las alternativas para montar nuestro cluster de mongoDB a través de "Google compute engine". Esta empresa nos ofrece la misma infraestructura que utiliza para realizar billones de búsquedas en milisegundos, administrar millones de cuentas de gmail, etc.

A su vez permite combinar sus servicios de cómputo, almacenamiento y funcionalidades (a través de Google API's).

Las opciones que nos ofrece Google compute engine son similares a las Ec2 de Amazon: Máquinas de propósito general, optimizadas para procesamiento y optimizadas para el manejo de la memoria. Estas últimas son las que nos interesan como plataforma en donde almacenar los datos de nuestra aplicación.

Dentro de las características especiales que ofrece el servicio, una de las más importantes es la velocidad y robustez de las transferencias utilizando la red global de google. Además brinda un servicio de balanceo para distribuír los datos en diversas instancias disminuyendo el tráfico de la red y aumentando la performance de nuestra aplicación.

Finalmente, así como Amazon ofrecía el servicio de S3, google provee *Cloud storage o Cloud Datastore*, en el caso de que queramos combinar mongoDB con otras tecnologías.

Si bien "Google compute engine" no está incluída como opción en el conjunto de plataformas que soporta mongoDB mencionadas en el manual (http://docs.mongodb.org/ecosystem/platforms/), actualmente existe dicho soporte. Las capacidades de la infraestructura de GCE hacen que la base de datos pueda funcionar muy satisfactoriamente, principalmente por la velocidad de la red que hemos mencionado más arriba: http://blog.mongolab.com/2013/05/mongolab-now-supports-google-cloud-platform/

4.0.4. dotCloud

MongoDB puede correr en las plataformas que ofrece "dotCloud". Las mismas soportan *replica sets* y sharding. La principal ventaja de dotCloud es que permite correr las aplicaciones y mantener la base de datos en un mismo lugar para optimizar la latencia y confiabilidad.

Por estos motivos, dotCloud aparece como segunda alternativa en la documentación de plataformas de mongo. (http://docs.mongodb.org/ecosystem/platforms/dotcloud/).

Para inicializar nuestra aplicación podemos olvidar la arquitectura diseñada previamente y dejar que dotCloud realice las configuraciones de escalabilidad de manera automática.

Por lo tanto simplemente debemos elegir los servicios que va a requerir la aplicación, incluyendo lógicamente entre ellas mongoDB para la elección de la base de datos.

(https://www.dotcloud.com/howitworks.html)

4.0.5. Escalabilidad lineal

La utilización de *Haas* como medio para albergar nuestra base de datos es una muy buena opción. Además de los motivos anteriormente mencionados, uno muy importante es que la inversión inicial es muy baja. No es necesario pagar por adelantado. Simplemente se paga por los recursos utilizados. En cuanto haya recursos pedidos que no están siendo utilizados, fácilmente podemos dejar de contratarlos y (por lo tanto) dejar de pagarlos.

Suponiendo que el primer año tengamos un gasto de x por haber pedido recursos (como por ejemplo, las distintas AMI's de Ec2 de Amazon), se espera que el siguiente tengamos un gasto de cx, donde c es una constante. Lo mismo para los años siguientes. Por lo tanto, la empresa probablemente pueda afrontar los gastos año tras año, considerando que la inversión durante el primer año no es tan alta, y que los ingresos que genera la aplicación son suficientes (no necesariamente demasiado elevados).

Por lo tanto la solución propuesta mediante *Haas* escala a nivel económico.

4.0.6. Migración a Hardware propio

Si en un futuro la empresa decidiera cambiar la implementación *Haas* por una de hardware propio podrían surgir grandes riesgos especialmente a nivel económico:

En primer lugar la inversión inicial que habría que hacer sería muy grande, especialmente si la aplicación cuenta para esa altura con muchos clientes y una base de datos considerablemente grande.

En segundo lugar debería destinarse una porción de los recursos económicos a la creación de un departamento de mantenimiento del hardware enfocado a bases de datos de gran tamaño. (Dentro del mismo, deberían tratarse temas como por ejemplo backup de datos y seguridad, como los que aprendimos en la segunda mitad de la materia. Entre ellos, distintos niveles de RAID para proteger los datos).

En suma, la empresa no contaría con la suficiente flexibilidad como para afrontar las subas y bajas de la demanda: Se pierde el concepto de escalar hacia abajo porque deja de ser viable "tirar o vender" el hardware sobrante. Al mismo tiempo, escalar hacia arriba pasa a ser algo riesgoso, ya que cada vez que se quiera ampliar la capacidad de la base no va a saberse con certeza si la compra está justificada o no.

Una solución parcial sería brindar un web-service propio con el hardware propio, aunque eso no es algo para nada fácil. Por un lado debería haber empleados que se dediquen a brindar ese servicio, y por el otro la empresa debería competir con otras como Amazon o Google.

En resumen, la migración de *Haas* hacia hardware propio no parece algo demasiado seguro .

4.0.7. Hecho real: Reddit sobre Amazon Ec2

Reddit es una página de internet en donde los usuarios postean, comentan y votan ideas, hechos y acontecimientos de interés general. Actualmente el sitio corre enteramente sobre AWS Ec2, y escala exitosamente mes tras mes, lo que le posibilita poder manejar hasta 4 billones de visitas a su página por mes. Según Keith Mitchell (programador del sitio), en un video que podemos ver en: http://aws.amazon.com/solutions/case-studies/reddit/ (link desde la página oficial de aws), explica que la empresa migró de tener sus propios servidores físicos a contratar los servicios de Amazon obteniendo resultados asombrosos. Anteriormente disponían de 5 equipos completos de mantenimiento de hardware (hecho que resultaba costoso). La empresa cuenta actualmente con menos de 20 empleados, y este hecho no les impide manejar el volumen de datos que maneja. Debido al encapsulamiento que provee el uso de web-services, pueden enfocarse en cuestiones como el código de la aplicación en sí mismo, o aspectos publicitarios.

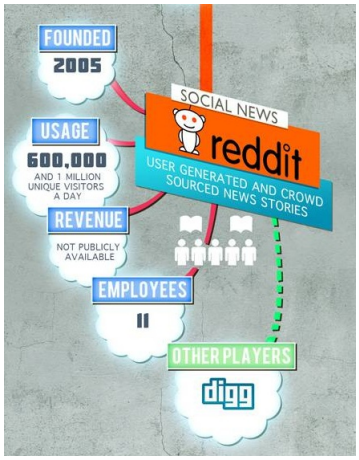

Figura 3: Esquema gráfico: datos reales de Reddit

5. Conclusiones

Durante el transcurso de este trabajo práctico comprendimos los usos y ventajas del paradigma MapReduce para análisis de datos en sistemas distribuidos, así como tambien las ventajas de montar un proyecto sobre HaaS, evaluando distintas alternativas de dicho servicio.

Entendemos que, para la primera parte, MapReduce ofrece una gran herramienta al momento de efectuar ciertos análisis sobre bases de datos muy extensas, dado que se puede paralelizar de manera distribuida sin perder precisión en los resultados. A su vez, encadenando secuencialmente Map Reduces sobre las sucesivas colecciones que se van generando se pueden realizar profundos análisis muy fácilmente y con peso computacional distribuido.

Aun así, errores en el código pueden llevar a resultados inesperadamente engañosos, por lo que es importante que las funciones *map* y *reduce* cumplan con ciertos requisitos (http://docs.mongodb.org/manual/reference/comm map-cmd), fundamentalmente idempotencia en la función *reduce*, que el orden de los valores no altere el resultado final, y que el valor de retorno de la función *reduce* matchee con el valor emitido por *map* (puesto que puede que se llame a *reduce* mas de una vez para la misma key).

El modelo de procesamiento map reduced permite escalar la base de datos horizontalmente.

Al utilizar este paradigma, mongoDB puede explotar todos los recursos que brinda un cluster de procesamiento.

Para montar inicialmente la base de datos de *Reddit* proponemos un cluster minimal que soporta sharding (2 query routers, 3 config servers y 2 shards). La idea es ir agregando o quitando nodos a medida que la demanada aumenta o disminuye.

Alcanzada esta etapa abrimos el siguiente interrogante: ¿Conviene montar el cluster con hardware propio o utilizar web-services provistos por empresas como Amazon o google? El primer camino es riesgoso a nivel económico, ya que requiere una fuerte inversión inicial, poca flexibilidad para escalar junto con las fluctuaciones de la demanda y la necesidad de invertir recursos en el mantenimiento de hardware. La segunda, por el contrario, es mucho más viable. La empresa puede obtener poder de procesamiento y almacenamiento a través de máquinas virtuales por un bajo costo. Al tratarse justamente de componentes no físicos, el usuario puede pedir o liberar los recursos de forma rápida y flexible. De esta manera termina pagando exactamente por los recursos que está utilizando, y no más que eso. En suma, se libera de la responsabilidad de realizar cualquier tipo de tarea de mantenimiento. Los precios bajos y el buen rendimiento de este tipo de servicios permite que las aplicaciones (junto con sus bases de datos) escalen tanto en performance como económicamente a medida que el volumen de los datos y de los clientes aumenta. Finalmente optamos por los servicios Ec2 de AWS debido a que ofrecen un soporte oficial para mongoDB.

Muchas aplicaciones y páginas webs importantes en la actualidad se han beneficiado con el surgimiento y desarrollo de "cloud computing" y del fenómeno "everything as a service" (donde *Haas* está incluído). Reddit es un ejemplo real. Una página web que soporta 4 billones de visitas diarias es capaz de funcionar con menos de 20 empleados debido a que delega gran parte de las responsabilidades necesarias para el funcionamiento de su aplicación en los servicios de Amazon.