DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Sistemas Operativos

Trabajo práctico 3

Algoritmos en sistemas distribuidos

Resumen

Integrante	LU	Correo electrónico
Acosta, Javier Sebastian	338/11	acostajavier.ajs@gmail.com
Gomez, Fernando Nahuel	695/11	fernando.gmz12@gmail.com
Russo, Christian Sebastián	679/10	christian.russo@gmail.com

Palabras claves:

Map, Reduce

Índice

1. Implementacion Map-Reduce

1.1. Encontrar el subreddit con mayor score promedio

Para realizar este análisis se implemento una función **Map**, la cual utiliza como clave el campo *subreddit* del post y a dicha clave asocia la tupla formada por el campo *score* del post y un uno para contabilizar el post. Luego, la función **Reduce** contabiliza la cantidad de post de un mismo subreddit y calcula la sumatoria de los scores de estos, retornando ambos en una tupla. Finalmente, la función **Finalize** se encarga de calcular el promedio con los dos valores anteriores.

Luego, ejecutamos en Mongo los siguientes comandos para obtener el resultado buscado:

```
use reddit
db.default.find().sort({"value":-1).limit(1)
```

Simplemente ordenamos los resultados en orden descendente en función del campo *value*, el cual contiene el score promedio de cada subreddit, y nos quedamos con el primero, es decir, el máximo.

El resultado obtenido es que el subreddit *GirlGamers* es el que tiene mayor score promedio, con un valor de 2483.

1.2. Encontrar los doce títulos con mayor score de la colección de posts con al menos 2000 votos

La función **Map** implementada utiliza como clave el campo title del post y como valor asociado utiliza una tupla formada por los campos total_votes y score de dicho post. Luego, la función **Reduce** calcula la sumatoria de los votos totales y la sumatoria de los scores y retorna ambos resultados en una tupla. Finalmente, la función **Finalize** verifica si titulo tiene mas de 2000 votos y, en caso afirmativo, retorna la sumatoria de los scores. En caso contrario retorna cero.

Luego, utilizamos los siguientes comandos de Mongo:

```
use reddit
db.default.find().sort({"value":-1).limit(12)
```

Utilizamos casi los mismos comandos que en el punto anterior, solo que aqui ordenamos de forma descendente el campo *value*, el cual contiene el score de cada titulo, y nos quedamos con los primeros doce.

Los doce títulos con mayor score y, al menos, 2000 votos son:

```
The Bus Knight
Haters gonna hate
So my little cousin posted on FB that he was bored and gave everyone his new phone number...
My friend calls him &quot;Mr Ridiculously Photogenic Guy&quot;
This is called humanity.
Genius
Seems legit.
President Obama's new campaign poster
Poster ad for the Canadian Paralympics
Fuck the police
Soon...
I'm sorry pinata bro
```

1.3. Para los diez mejores scores, calcular la cantidad de comentarios en promedio por sumisión

La función **Map** implementada para este ejercicio utiliza como clave el campo *score* del post y le asocia la tupla formada por un uno (para contabilizar la sumisión) y el campo *number_of_comments* de dicho post. Luego, la función **Reduce** retorna una tupla formada por la sumatoria de unos (cantidad de sumisiones) y la sumatoria del numero de comentarios para cada score (cantidad de comentarios). Finalmente, la función **Finalize** se encarga de calcular la cantidad de comentarios en promedio realizando la división de los dos valores retornados por Reduce.

Luego, utilizamos en Mongo los siguientes comandos:

```
use reddit
db.default.find().sort({"_id": -1).limit(10)
```

Lo que hacemos es ordenar los resultados en forma descendente por su score (el cual recordemos que utilizamos como clave) y luego no quedamos con los 10 mejores.

El resultado obtenido es:

Score	Cantidad de comentarios en promedio por sumision
20570	1463
12333	1612
11908	2681
10262	1514
8935	480
8835	1716
8699	934
8241	571
7297	1110
6741	2204

1.4. Entre los usuarios con a lo sumo 5 sumisiones, encontrar el que posee mayor cantidad de upvotes

La función **Map** utiliza el campo *username* como clave y le asocia la tupla formada por un uno (para contabilizar la sumisión) y el campo *upvotes* del post. Luego, la función Reduce se encarga de realizar la sumatoria de sumisiones y upvotes para cada usuario y, finalmente, la función Finalize se encarga de retornar el numero de upvotes para aquellos usuarios que tengan a lo sumo 5 sumisiones. Para aquellos usuarios que tienen mas de 5 sumisiones retorna cero.

En Mongo utilizamos los comandos:

```
use reddit
db.default.find().sort({"value":-1).limit(1)
```

De la misma forma que en el primer ejercicio, obtenemos al usuario con mayor cantidad de upvotes y a lo sumo 5 sumisiones. El usuario encontrado es *lepry* con 90396 upvotes.

1.5. Para todos los subreddit que poseen un score entre 280 y 300, indicar la cantidad de palabras presentes en sus títulos

La función **Map** utiliza como clave el campo *subreddit* del post y le asocia la tupla formada por el campo *score* del post y un entero indicando la cantidad de palabras en el titulo del post *(calculado como this.title.split().length)*. Luego, la función **Reduce** lleva a cabo la sumatoria de la cantidad de palabras en el titulo y la sumatoria de los score para cada subreddit. Finalmente,

la función **Finalize** determina si el subreddit tiene un score entre 280 y 300. En caso afirmativo retorna la sumatoria de la cantidad de palabras en sus títulos. En caso contrario retorna -1.

Luego, utilizamos los siguientes comandos en Mongo:

```
use reddit
db.default.find({"value": {$ne: -1}})
```

Con este comando filtramos los resultados para solo quedarnos con aquellos que nos interesan, es decir, para los cuales **Finalize** no retorno -1 y que, por lo tanto, tienen un score entre 280 y 300.

El resultado obtenido es que los siguiente subreddit son los que tienen un score entre 280 y 300 (entre paréntesis colocamos la cantidad de palabras presentes en sus títulos):

Subreddit	Cantidad de palabras presentes en sus titulos
Feminism	6
Firearms	24
HeroesofNewerth	6
Sexy	13
TheRealZachAnner	4
ragecomics	4
xkcd	4

2. Investigación

2.1. Explique cuál es el entorno y la situación donde se plantea el problema (Motivación)

MapReduce y su implementación open-source *Hadoop* fueron originalmente optimizadas para grandes lotes de trabajo (batch jobs), como la creación de un índice web. Sin embargo, otro caso de uso surgió recientemente: compartir un cluster MapReduce entre varios usuarios, los cuales ejecutan una mezcla entre grandes lotes de trabajo y pequeñas queries interactivas sobre un conjunto de datos en común. Compartir permite utilizar multiplexación estadística, la cual reduce los costos y, además, conduce a la consolidación de los datos. Esto evita la costosa replicación de los datos a través de clusters privados y permite a una organización ejecutar queries no anticipadas a través de conjuntos de datos disjuntos eficientemente.

La motivación original surge a partir de la carga de trabajo MapReduce en Facebook. Los registros de eventos del sitio web de Facebook son importados a un cluster Hadoop cada hora, donde son utilizados para una variedad de aplicaciones, incluyendo análisis en los patrones de uso para mejorar el diseño del sitio web, detectar spam, data mining, etc.

La central de datos de Facebook corre en 600 máquinas y almacena 500 TB de información comprimida, la cual crece a un promedio de 2 TB por día. Además de los trabajos de *producción* que se deben ejecutar periódicamente, también existen varios trabajos experimentales que van desde cálculos de machine-learning de varias horas hasta queries ad-hoc enviadas cada 1 o 2 minutos a través de una interfaz SQL. Este sistema ejecuta 3200 trabajos MapReduce por día y ha sido utilizado por más de 50 ingenieros de Facebook.

2.2. Identifique situaciones donde el problema se dispara y la consecuencias que esto genera

Cuando varios grupos de trabajo comienzan a utilizar Hadoop, el tiempo de respuesta de los trabajos comienzan a sufrir debido al scheduler FIFO que este utiliza. Esto es inaceptable para los trabajos de producción y hace las queries interactivas imposibles, reduciendo altamente la utilidad del sistema. Algunos grupos consideraron construir un cluster privado para su carga de trabajo, pero esto era demasiado caro e injustificado para muchas aplicaciones.

Por otra parte, debido a la interdependencia que existe entra las tareas Map y Reduce, puede suceder que un trabajo que toma mucho tiempo en ejecutarse reserve muchos slots para llevar a cabo su etapa de Reduce, los cuales no liberará hasta haber acabado con su fase Map, produciendo así inanición en otros trabajos y desperdiciando recursos.

2.3. Identifique del problema

Los algoritmos tradicionales de scheduling pueden desempeñarse muy pobremente en MapReduce degradando el throughput y el tiempo de respuesta debido a dos aspectos de este: la necesidad de la localidad de los datos (procesar los datos en donde estos se encuentran) y la dependencia entre las tareas Map y Reduce. Estos dos aspectos diferencian el scheduling en clusters MapReduce del scheduling en clusters tradicionales.

2.4. Comente el background histórico del problema

El scheduler integrado de Hadoop ejecuta trabajos en orden FIFO, con cinco niveles de prioridad. La desventaja de un scheduler FIFO es el pobre tiempo de respuesta para trabajos cortos en presencia de trabajos que consumen mucho tiempo. La primer solución para este problema en

Hadoop fue Hadoop On Demand (HOD), el cual provee clusters MapReduce privados sobre un gran cluster físico utilizando Torque. HOD permite a los usuarios compartir un sistema de archivos en común (corriendo en todos los nodos) mientras poseen un cluster MapReduce privado en su nodo asignado. Sin embargo, HOD tiene dos problemas:

- **Pobre localidad**: Cada cluster privado corre en un conjunto definido de nodos, pero los archivos HDFS (sistema de archivos que utiliza Hadoop) están distribuidos a través de todos los nodos. Como resultado, algunos Maps deben leer datos a través de la red, degradando el rendimiento y el tiempo de respuesto.
- **Pobre utilización**: Como cada cluster privado tiene un tamaño fijo, algunos nodos en el cluster físico pueden estar inactivos. Esto es subóptimo ya que los trabajos en Hadoop son *elásticos*, en el sentido de que estos pueden cambiar en el tiempo en cuántos nodos se están ejecutando. Los nodos inactivos podrían así incrementar la velocidad a la que se ejecutan los trabajos.

2.5. Explique otros problemas asociados a la búsqueda de una mejor solución

Localidad de datos

El primer aspecto de Map Reduce que plantea retos de scheduling es la necesidad de colocar cálculos cerca de los datos. Esto aumenta el rendimiento porque el ancho de banda de la red de bisección es mucho más pequeño en un clúster grande que el ancho de banda total de los discos del clúster. Ejecutar sobre un nodo que contiene los datos (localidad de nodos) es más eficiente, pero cuando esto no es posible, ejecutar sobre un mismo rack (localidad de racks) provee mejor performance que utilizar un rack no local. Por ejemplo, los nodos de Facebook poseen cuatro discos con un ancho de banda de 50 a 60 MB/s cada uno o 2 GB/s en total, mientras que las conexiones al rack son de 1 GB/s

El primer problema de localidad se vio con trabajos pequeños. Es causado por un comportamiento denominado *head-of-line scheduling*. En cualquier momento, hay un único trabajo que debe ser el proximo en ser ejecutado, de acuerdo con *Fair sharing*: el trabajo con el que menos justo se fue. Cualquier esclavo que solicite una tarea se le otorga una a través de ese trabajo. Sin embargo, si el trabajo head-of-line es pequeño, la probabilidad de que los bloques de entrada de un esclavo solicitando una tarea estén disponibles es baja.

Un problema relacionado, *sticky slots* (slots pegajosos), sucede incluso con trabajos largos si se usa ecuanimidad. Por ejemplo si hay 10 trabajos en uno cluster de 100 nodos con un slot por nodo y cada uno corre 10 tareas. Supongamos que el trabajo 1 finaliza una tarea en el nodo X. El Nodo X envía una señal solicitando una nueva tarea. En este punto, el trabajo 1 tiene 9 tareas corriendo mientras todos los otros trabajos tienen 10. Por lo tanto, el trabajo 1 es asignado al nodo X de nuevo. En consecuencia, en estado estable, los jobs nunca dejan sus slots originales. Esto conduce a una mala localidad como en HOD, porque los archivos de entrada son distribuidos a través del cluster. La localidad puede ser incluso menor con pocos trabajos. Incluso con 5 trabajos, la localidad está por debajo del 50 %.

Interdependencia entre Reduce/Map

En adición a la localidad, otro aspecto de MapReduce que crea retos en el sheduling es la dependencia entre las fases de Map y Reduce. Las fases Map generan resultados intermedios que son almacenados en disco. Cada Reduce copia su porción de resultados de cada Map, y solamente puede aplicar la función de reducción del usuario una vez que se cuenta con los resultados de todos

los Maps. Esto puede llevar a un problema de *acaparamiento de los slot* donde los trabajos largos mantienen los slots de Reduce durante mucho tiempo, haciendo que los trabajos pequeños sufran de inanición y se subutilizen recursos.

El primer problema de scheduling de Reduce ocurre en Fair sharing cuando hay trabajos grandes que se ejecutan en el cluster. Hadoop normalmente lanza tareas de Reduce de un trabajo tan pronto como terminan sus primeros Maps, por lo que los Reduce pueden empezar a copiar salidas de Map mientras que los Map restantes se están ejecutando. Sin embargo, en un trabajo grande, de decenas de miles de tareas de Map, la fase de Map puede tardar mucho en completarse. El trabajo retendrá cualquier slot de Reduce que reciba hasta que finalicen todos sus Map. Esto significa que si hay periodos de baja actividad de otros usuarios, el trabajo grande se hará cargo de la mayor parte o de la totalidad de los slots de Reduce en el cluster, y si cualquier trabajo se presenta más tarde, este trabajo sufrirá de inanición hasta que termine el trabajo grande.

Este problema también afecta al rendimiento. En muchos trabajos grandes, las salidas de los Map son pequeñas, debido a que los Map están filtrando un conjunto de datos. Por lo tanto, las tareas Reduce que ocupan slots estarán, mayormente, ociosas a la espera de que terminen los Maps. Sin embargo, un slot Reduce completo es asignado a estas tareas, ya que su fase de computo será costosa una vez que termine su fase de copiado. El trabajo efectivamente *reserva* slots para una futura operación intensiva en el uso del CPU o de la memoria, pero durante el tiempo que los slots están reservados, otros trabajos podrían haber aprovechado el CPU o la memoria para realizar sus propias operaciones.

Puede parecer que estos problemas pueden resolverse comenzando tareas de reduce más tarde o haciéndolas pausables. El problema es que dos operaciones con distintas necesidades de recursos – una copia intensiva en IO y un Reduce intensivo en CPU – se agrupan en una sola tarea. En cualquier momento, un slot de reduce esta o bien utilizando la red para copiar salidas de Map o esta usando el CPU para aplicar la función de Reduce, pero no ambos. Esto puede degradar el rendimiento en un solo trabajo con múltiples oleadas de tareas de Reduce pues hay una tendencia a reducir Reduces a copiar y calcular de forma sincronizada. Cuando los Maps de los trabajos terminan, la primera ola de tareas Reduce comienzan a computar casi simultáneamente, y luego terminan en más o menos el mismo tiempo y la próxima ola empieza a copiar, y así sucesivamente. En cualquier momento, todos los slots están copiando o todas están computando. El trabajo correría mas rápido superponiendo la utilización de IO y CPU (continuar copiando mientras que la primera oleada está computando).

2.6. Explique las soluciones propuestas

Para solucionar los problemas se desarrolló el **scheduler FAIR**. Este scheduler tiene dos metas principales:

- **Aislamiento**: Darle a cada usuario (trabajo) la ilusión de poseer (correr) un cluster privado.
- **Multiplexación estadística**: Redistribuir la capacidad no utilizada por algunos usuarios (trabajos) a otros usuarios (trabajos).

FAIR utiliza una jerarquía de dos niveles. En el nivel más alto, FAIR asigna slots de tareas a través de *conjuntos*, y en el segundo nivel, cada conjunto asigna sus slots entre los múltiples trabajos que contiene. Este modelo puede ser fácilmente extendido a más niveles.

FAIR utiliza una versión de max-min fairness con garantías mínimas para asignar slots a través de los conjuntos. A cada conjunto i se le otorga una cuota mínima m_i de slots, la cual puede ser cero. El scheduler se asegura de que cada conjunto reciba su cuota mínima de slots, siempre y cuando tenga suficiente demanda, y la sumatoria sobre todas las cuotas mínimas de slots de cada conjunto no exceda la capacidad del sistema. Cuando un conjunto no utiliza completamente su cuota de slots, otros conjuntos tienen permitido utilizar los que este no utilice.

Delay scheduling

Para solucionar el problema de la localidad de los datos se utilizó una técnica simple denominada *Delay scheduling*. Cuando un nodo solicita una tarea, si el trabajo no puede lanzar una tarea local, se lo saltea y se pasa a ver los trabajos subsecuentes. Sin embargo, si un trabajo a sido salteado durante un tiempo determinado, se le permite lanzar una tarea no local, evitando así la inanición.

Copy-compute splitting

Para solucionar los problemas generados por la dependencia entre las fases Map y Reduce, se propuso dividir las tareas Reduce en dos tareas lógicas: tareas de copia y tareas de cómputo, cada cual con una forma diferente de control de admisión, es decir, cada nodo posee slots de copia y slots de cómputo para la fase de Reduce. Las tareas de copia buscan y combinan la salida de las tareas Map, una operación que normalmente está ligada a realizar IO en la red. Las tareas de cómputo aplican la función Reduce del usuario a la salida de las tareas Map.

La última complicación generada por la interdependencia entre las fases Map y Reduce es el espacio en disco utilizado por los resultados intermedios (salida de la fase Map). Hadoop ya cuenta con mecanismos de seguridad para asegurarse de que los discos de los nodos no se llenen. Sin embargo, es posible que se produzca un Deadlock si dos trabajos se ejecutan concurrentemente y el disco se llena antes de que alguno de los dos termine. Para evitar este problema, los resultados de uno de los trabajos deben ser eliminados hasta que el otro trabajo termine. Cualquier heurística para elegir que trabajo desalojar sirve siempre y cuando permita a algún trabajo en el cluster avanzar en su progreso. Algunas opciones son:

- Desalojar al último trabajo que llegó.
- Desalojar al trabajo con el menor porcentaje de progreso.

Sin embargo, hay una heurística que no funciona: Desalojar al trabajo que esta utilizando mas espacio del disco. Este método puede llevar a una situación donde los nuevos trabajos que llegan desalojan a los trabajos viejos, y ningún trabajo termina nunca.

2.7. Evalué soluciones propuestas

Para evaluar la técnicas de scheduling sea realizaron macrobenchmarks y microbenchmarks en tres ambientes diferentes: *Amazon Elastic Compute Cloud* (EC2), que es un entorno virtualizado de hosting comercial, un cluster de 100 nodos (SPC) y un cluster de 450 nodos (LPC).

Environment	Nodes	Hardware and Configuration
EC2	100	4 2GHz cores, 4 disks and 15 GB RAM per node; appear to have 1 Gbps links. 4 map and 2 reduce slots per node.
Small Private Cluster	100	8 cores and 4 disks per node; 1 Gbps Ethernet; 4 racks. 6 map and 4 reduce slots per node.
Large Private Cluster	450	8 cores and 4 disks per node; 1 Gbps Ethernet; 34 racks. 6 map and 4 reduce slots per node.

Figura 1: Entornos utilizados

Para las cargas de trabajo se utilizó *Loadgen*. Loadgen es un trabajo configurable donde las tareas Map devuelven un keepMap % de los datos datos de entrada, luego las tareas Reduce devuelven un keepReduce % de los datos intermedios.

2.7.1. Macrobenchmark

Se ejecutó un benchmark multiusuario con tamaños de trabajo y tiempos de llegada basados en la carga de trabajo de Facebook. El benchmark utiliza trabajos loadgen con valores al azar para keepMap y keepReduce, y tiempos de uso de CPU al azar (los Maps toman entre 9 y 60 segundos). Este benchmark, referido como BM a partir de ahora, consiste de 50 trabajos de 9 tamaños (números de Maps). La siguiente tabla muestra la distribución de los tamaños en Facebook, la cual fue agrupada en 9 grupos.

Bin	#Maps	% at Facebook	Size in BM	Jobs in BM
0	1-25	58%	16	29
1	25-50	9.6%	40	5
2	50-100	8.6%	80	4
3	100-200	8.4%	160	4
4	200-400	5.6%	320	3
5	400-800	4.3%	600	2
6	800-1600	2.5%	1200	1
7	1600-3200	1.3%	2400	1
8	> 3200	1.7%	6400	1

Figura 2: Grupos de trabajos

Se eligió un tamaño representativo para cada grupo y se envió algún numero de trabajos de dicho tamaño para hacer más fácil el promediar el rendimiento de estos trabajos. Para cada trabajo, se estableció el número de Reduces entre un 5 % y un 25 % del número de Maps. Se envió los trabajos cada 30 segundos aproximadamente, siguiendo un proceso de Poisson, correspondiente al ritmo de envíos en Facebook.

Se generaron tres órdenes de envío de los trabajos de acuerdo a este modelo y se compararon cinco algoritmos de scheduling: FIFO y Fair sharing con y sin copy-compute splitting, así como Fair sharing con ambos copy-compute splitting y delay scheduling.

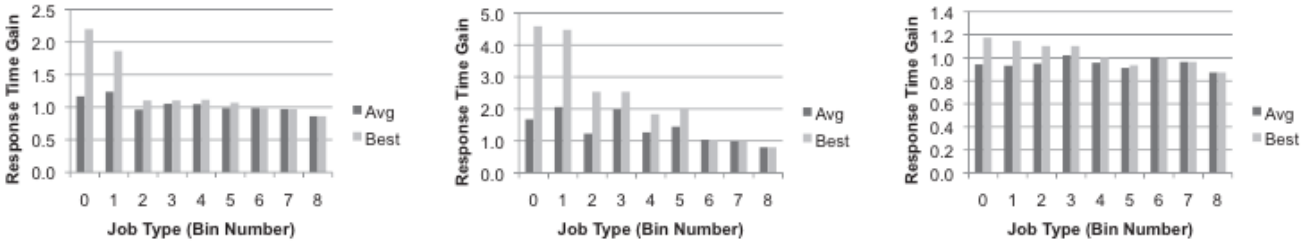

Figura 3: Ganancia promedio y mejor ganancia en tiempo de respuesta obtenidas por los trabajos de cada grupo en los diferentes algoritmos

La figura muestra el tiempo de respuesta promedio ganado por cada tipo (bin) de trabajos por

sobre su tiempo de respuesta en FIFO (por ejemplo, un valor 2 indica que el trabajo se ejecutó 2x más rápido que en FIFO), así como la máxima ganancia para cada grupo (la ganancia del trabajo que mejoró más). Se puede ver qué Fair sharing (figura (a)) puede acortar los tiempos de respuesta a la mitad en comparación con FIFO en los trabajos pequeños, a expensa de que los trabajos más largos tomen un poco mas. Sin embargo, algunos trabajos aún son demorados debido al acaparamiento de slots Reduce. Fair sharing con copy-compute splitting (figura (b)) provee ganancias de hasta 4.6x para trabajos pequeños, con el promedio ganado para los trabajos del tipo 0 y 1 siendo de 1.8-2x. Finalmente, FIFO con copy-compute splitting (figura (c)) produce algunas ganancias pero menos que Fair sharing.

Añadiendo delay scheduling no cambia el tiempo de respuesta de Fair sharing con copy-compute splitting de forma perceptible. Sin embargo, incrementa la localidad de 15-95 % a 99-100 % como se muestra en la siguiente figura (los valores para FIFO, Fair sharing, etc fueron similares, de modo que sólo se muestra un gráfico). Esto incrementa el rendimiento en un entorno con un ancho de banda más limitado que el de EC2 (el cual posee un ancho de banda de 1 GB).

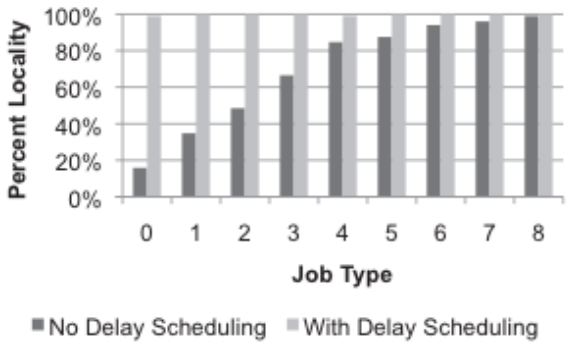

Figura 4: Localidad obtenida por cada grupo de trabajos

2.7.2. Microbenchmark

Delay scheduling con trabajos pequeños

Para testear el efecto de delay scheduling sobre la localidad y el rendimiento cuando se utilizan trabajos con poca carga de trabajo, se generó un gran conjunto de datos al azar y se corrió los trabajos sobre el en el entorno SPC. Cada trabajo es solamente un *scan*, es decir, un Map que lee un archivo y retorna un 0.5 % de los datos en el, simulando los comunes trabajos de filtrado cuando se analizan logs. Se ejecutaron trabajos con 3, 10 y 100 tareas Map. El benchmark ejecuta varios números de trabajos basado en el tamaño de los trabajos. Se compararon Fair sharing y FIFO con y sin delay scheduling. FIFO se desempeñó igual que Fair sharing, de modo que solo se mostraran dos barras.

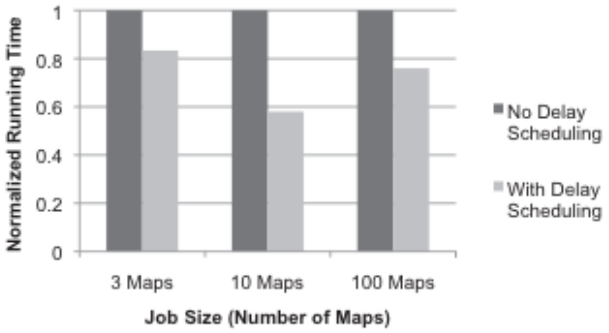

Figure 12: Finish times in small-jobs experiment.

Figura 5: Tiempos de ejecución para el experimento con trabajos pequeños

Job Size	Node/Rack Locality with Standard Sched.	Node/Rack Locality with Delay Sched.
3 maps	2% / 50%	75% / 96%
10 maps	37% / 98%	99% / 100%
100 maps	84% / 99%	94% / 99%

Figura 6: Localidad conseguida en el experimento con trabajos pequeños

Los gráficos muestran los tiempos de ejecución normalizados de los trabajos, mientras que la tabla 3 la localidad conseguida por cada scheduler. Delay scheduling incrementó el rendimiento **1.2x** en trabajos con 3 Maps, **1.7x** en trabajos con 10 Maps y **1.3x** en trabajos con 100 Maps, e incrementó la localidad de los datos al menos a un **75 %** y la localidad en los rack a al menos un **94 %**. El rendimiento ganado es mayor para los trabajos con 10 Maps que para los trabajos con 100 Maps debido a que la localidad en los trabajos con 100 Maps es bastante buena, incluso sin delay scheduling. Sin embargo, la ganancia para los trabajos más pequeños (3 Maps) es menor que para los trabajos con 10 Maps, debido a que los trabajos de pequeño tamaño al inicializarse producen un cuello de botella en Hadoop.

Delay scheduling con sticky slots

Sticky slots no ocurre en Hadoop debido a un bug en este, de modo que se arregló este bug para poder medir los efectos de sticky slots.

Este test se corrió en el entorno EC2. Se generó un gran conjunto de datos de 180 GB (2 GB por nodo), se envío entre 5 y 50 trabajos *scan* concurrentes y se midió el tiempo que tardaron en terminar todos los trabajos y la localidad conseguida. Los siguientes gráficos muestran los resultados obtenidos con y sin delay scheduling.

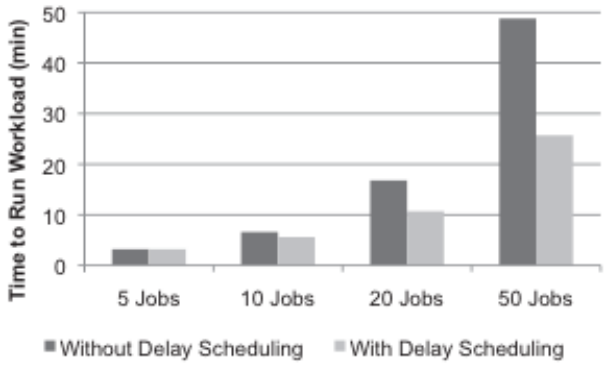

Figura 7: Tiempos de ejecución en el experimento de sticky slots

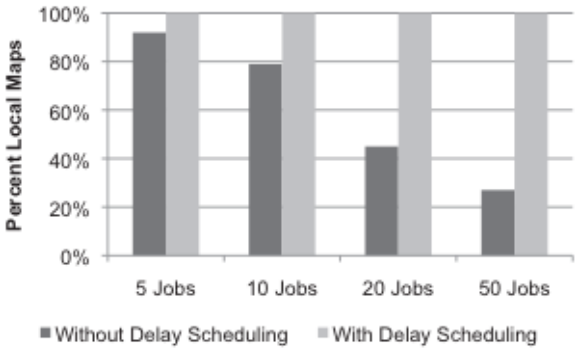

Figura 8: Localidad de nodos conseguida en el experimento de sticky slots

Delay scheduling incrementó el rendimiento en **1.1x** para 10 trabajos concurrentes, **1.6x** para 20 trabajos concurrentes y **2x** para 50 trabajos concurrentes. También incrementó la localidad de **27-90 %** hasta **99-100 %**.

Impacto de delay scheduling en el tiempo de respuesta

Se midió el tiempo de respuesta de un pequeño trabajo con 16 Maps y 1 Reduce (escaneando 2 GB y enviando **0.1 %** de los datos al Reduce) con y sin delay scheduling en el entorno EC2 para cuantificar el impacto en el tiempo de respuesta. Se ejecutó al trabajo ocho veces sobre cada condición. Ejecutándose sólo, el trabajo tomo 32 a 37 segundos, con una media de 35.4 segundos. Con delay scheduling, el trabajo tomo 32 a 38 segundos, con una media de 35.a segundos. La diferencia en los tiempos de respuesta no fue estadísticamente significativa, pero la localidad mejoró de un **25 %** a un **100 %**.

Copy-compute splitting con un solo trabajo

Desafortunadamente, las tareas de copia en Hadoop son bastante intensivas en el uso del CPU ya que estas realizan grandes cantidades de copias en memoria cuando combinan las salidas de los

Maps, así que hay poca ganancia por parte de copy-compute splitting a menos que el ancho de banda de la red sea muy limitado (las tareas de copia compiten con las tareas de cómputo por el CPU). En el entorno LPC se ejecutó un trabajo con una función identidad en Map y una función Reduce intensiva en CPU. El trabajo tomo 1.8 TB de datos de entrada. 14624 tareas Map y 5000 tareas Reduce. Los Reduce corrieron en tres oleadas, ya que el cluster está configurado con 1800 slots de Reduce (cuatro por nodo). El trabajo tomó, en promedio, 12.9 minutos con el scheduler de Reduce estándar y 11.8 minutos con copy-compute splitting, una ganancia del 9 %. Se espera que la ganancia sea mayor en clusters más grandes o con una implementación más eficiente de las tareas de copia.

Tiempo de respuesta Batch con Fair sharing vs FIFO

Se ejecutaron cinco trabajos en el entorno EC2. Cada trabajo lee 100 GB de datos. Los Maps del trabajo producen 200 GB de salida, en total. Estos datos son pasados a 190 tareas Reduce. Las tareas Reduce devuelven solo un pequeño conjunto pero realizan intensivos cálculos matemáticos por cada documento. Se repitió el experimento tres veces. Fair sharing tomo 25.9 a 26.5 minutos en completar el trabajo, mientras que FIFO tomo 21.9 a 22.3 minutos. Esto corresponde a un rendimiento **19 %** mayor con FIFO.

2.8. Discusión

Se identificaron dos aspectos que plantean problemas de los cluster que llevan a cabo cómputo intensivo en datos: la localidad de los datos y la interdependencia de las tareas. Cualquier sistema que coloque las tareas en los nodos que contienen los datos necesarios corre el riesgo de sufrir los problemas descritos: *head-of-line scheduling* y *sticky slots*. Similarmente, son comunes las tareas que deben recolectar datos de múltiples otras tareas antes de poder realizar su cómputo. Las técnicas de *delay scheduling* y *copy-compute splitting* pueden mejorar el rendimiento y el tiempo de respuesta en cualquier sistema basado en flujo de datos.

Lecciones para scheduling en clusters multiusuario

Siempre que los trabajos estén compuestos de pequeñas tareas independientes, es posible aislar a los usuarios mientras que se utiliza el cluster eficientemente. Se identificaron cuatro principios que ayudan a alcanzar esta meta:

- Hacer las tareas pequeñas en duración y en consumo de recursos. Tener tareas cortas permite a los nuevos trabajos comenzar más rápido. Limitar los recursos de cada tarea (por ejemplo, haciendo que las tareas utilicen un único thread y teniendo múltiples slots por nodo) incrementa las oportunidades de scheduling.
- Dividir las tareas en partes con requerimientos ortogonales de recursos para tener diferentes controles de admisión para cada tarea.
- Cuando las tareas deben ser largas, también deben ser desalojables.
- Estar preparado para sacrificar algo de aislamiento a cambio de rendimiento, como lo ilustra delay scheduling.

Schedulers para MapReduce:

Si comparamos los schedulers: FIFO, Fair sharing, SJF y colas multi-nivel (colas separadas para trabajos largos y cortos, con Fair sharing ponderado entre las colas); y utilizamos las métricas:

tiempo de respuesta promedio, tiempo de respuesta batch (tiempo de respuesta para un grupo de trabajos que llegan a la vez), espacio intermedio y aislamiento de usuarios (definido como la habilidad de proveer al usuario un rendimiento similar al de un pequeño cluster privado en el peor caso, sin importar su carga de trabajo); se obtienen tres resultados interesantes:

- El tiempo de respuesta Batch sufre en todos los schedulers menos en FIFO, ya que los Maps pueden terminar más tarde y, por lo tanto, el pipelining entre las tareas Map y Reduce sufre.
- FIFO utiliza el menor espacio intermedio (no más que el necesario por el trabajo más grande), mientras que los que permiten a varios trabajos coexistir necesitan más espacio. Con SJF la cantidad de espacio utilizado puede ser arbitrariamente grande ya que muchos trabajos pueden estar en el sistema por un periodo arbitrariamente largo.
- Sólo Fair sharing provee aislamiento de usuarios (incluso en colas multi-nivel, el trabajo de un usuario puede ser retrasado por otros usuarios)

Aunque Fair sharing es subóptimo en tiempo de respuesta y utilización de espacio, hemos encontrado que el aislamiento de usuarios supera a estas preocupaciones mediante la creación de un ambiente donde los usuarios pueden iniciar trabajos en cualquier momento sin temor a interferencias. Un beneficio de que los usuarios puedan lanzar trabajos inmediatamente es que permite que la detección de bugs sea mas rápida: sin Fair sharing, un usuario que envía un trabajo con un Map o Reduce buggeados no lo detectara hasta que este alcance la primer posición de la cola y sea lanzado.

Otros problemas de scheduling

- **Localidad de código**: Ejecutar múltiples tareas de un trabajo en un nodo es beneficioso ya que amortiza el costo de tener copiar el código del trabajo. El problema de maximizar la localidad de código es similar a scheduling por afinidad en los sistemas operativos multiprocesador, donde se intenta que un thread se ejecute en un mismo procesador por todo el tiempo posible.
- **Gestión de la carga de los esclavos**: Cuando trabajos con diferentes requisitos de recursos (especialmente de memoria) coexisten, podría ser beneficioso utilizar un mecanismo más elaborado que slots de tareas para manejar la carga en los nodos esclavos. Sin embargo, otro problema que puede darse es inanición para los procesos intensivos en el uso de la memoria si trabajos con menores requisitos de memoria ocupan los slots.

2.9. Conclusiones

Mientras que MapReduce ha probado ser un modelo popular para ejecutar grandes lotes de trabajos, recientemente, muchas organizaciones han comenzado a compartir su cluster MapReduce (Hadoop) entre múltiples usuarios, los cuales ejecutan una mezcla de lotes de trabajos y pequeños trabajos interactivos. Para habilitar este modelo, se ha propuesta a FAIR, un scheduler que provee aislamiento, garantiza una cuota mínima para cada usuario (trabajo) y logra multiplexación estadística. En una primera instancia, se han identificado dos aspectos de MapReduce (localidad de los datos e interdependencia entre tareas Map y Reduce) que afectaban considerablemente el rendimiento de FAIR. Para solucionar este problema se desarrollaron dos simples y, a su vez, robustas soluciones: delay scheduling y copy-compute splitting. Utilizando un amplio conjunto de experimentos se demostró que FAIR consigue aislamiento, bajo tiempo de respuesta y alto rendimiento.