MapReduce en el procesamiento distribuido de grandes volúmenes de información

by Yixander Yero Tarancon - Wednesday, December 04, 2013

https://vinculando.org/beta/revision-modelo-programacion-mapreduce.html

Resumen:

El aprovechamiento efectivo de toda la capacidad de computación disponible hoy en día en los diferentes centros de cómputo, ha provocado un crecimiento del número de lenguajes y modelos de programación para cada arquitectura específica. Las características de estos lenguajes los vuelven un trabajo serio para los programadores pues ofrecen en su mayoría implementaciones de bajo nivel que alejan al programador del diseño del programa y más centrado en el código. MapReduce es un modelo de programación para el procesamiento y generación de grandes volúmenes de datos. Los programas escritos en este estilo funcional son automáticamente paralelizados y ejecutados en un clúster de máquinas. El ambiente se encarga de los detalles de particionamiento de los datos de entrada, el control de las instancias del programa en las máquinas, manipular los fallos y la comunicación; permitiéndole a un programador sin experiencia en sistemas distribuidos y paralelos utilizar los recursos del ambiente distribuido fácilmente. Este trabajo hace una revisión de las características del modelo de programación tratado así como los principales sistemas donde se aplica, resaltando las ventajas que lo vuelven atractivo y extensible al análisis de grandes volúmenes de datos.

Palabras clave: map; reduce; procesamiento paralelo; sistema distribuido; data.

Review of the MapReduce Programming Model as a alternative for the large amount of data distributed processing.

Abstract:

The effective use of all the computing capabilities available today in the different data centers across the world, has led to a growing number of languages ??and programming models for each specific architecture. The characteristics of these languages ??make them serious work to programmers as they offer mostly low-level implementations that keep the developer away of the program design and mostly centered to data code. MapReduce is a programming model to processing and generating large volume data. Programs written in this functional style are automatically parallelized and executed on a cluster of machines. The environment takes care of the details of partitioning the input data, control of the instances of the program, handling failures and communication; allowing an inexperienced programmer in distributed and parallel systems to use the distributed resources more easily. This paper reviews the characteristics of this programming model, pointing the advantages that make it attractive and extensible to analyze large volumes of data.

Keywords: *map; reduce; parallel processing; distributed system; data.*

Introducción

Dada la necesidad de realizar implementaciones para trabajar con grandes volúmenes de información en entornos de trabajo distribuidos y procesar los resultados relevantes, teniendo en cuenta los problemas de cómo paralelizar los cálculos, distribuir la información y manejar los fallos, de cierto modo, complejizan la labor del desarrollador.

El modelo de programación MapReduce delega los cómputos intensivos en datos a un clúster de máquinas remotas que, mediante un sistema de ficheros distribuido, repartirán la carga de trabajo, optimizando tiempo y recursos. Facilita un patrón de desarrollo paralelo para simplificar la implementación de aplicaciones en entornos distribuidos. Este modelo puede dividir un espacio grande de problema en espacios pequeños y paralelizar la ejecución de tareas más pequeñas en estos sub-espacios. Está diseñado para la escalabilidad y tolerancia a fallos en grandes sistemas, está basado en la combinación de operaciones Map y Reduce, fue diseñado originalmente por Google y es usado en múltiples operaciones que conllevan a un manejo de varios petabytes diarios.

Este modelo tiene una gran aplicabilidad en entornos de *Cloud Computing* [1]. El Cloud Computing es un paradigma de computación que consiste en ofrecer servicios alojados en máquinas remotas bajo demanda y a través de Internet. Este paradigma ofrece ventajas como la centralización de los datos en servidores remotos, eliminando las dependencias con los soportes físicos; o la contratación de servicios en función de las necesidades de las empresas, sin tener que añadir equipos, software o personal.

El modelo de programación MapReduce

MapReduce [2] es un modelo de programación desarrollado y utilizado por Google para procesar grandes conjuntos de datos distribuidos a lo largo de un clúster de servidores. Este puede aplicarse tanto sobre datos almacenados en sistemas de ficheros, como en bases de datos. El modelo de programación permite al desarrollador expresar sus algoritmos utilizando únicamente dos funciones, *map* y *reduce*.

Las funciones *map* y *reduce* de MapReduce se definen sobre datos estructurados en pares clave-valor. La función *map*, escrita por el usuario, recibe un par clave-valor y devuelve un conjunto de pares clave-valor intermedio:

map (k1, v1) –> list (k2, v2)

Esta función se aplica en paralelo a cada par del conjunto de datos de entrada produciendo una lista de pares **(k2, v2)** por cada llamada. MapReduce agrupa todos los valores intermedios asociados con la misma clave *k* y se los pasa a la función *reduce*.

La función *reduce* recibe esa clave y su conjunto de valores asociados y los fusiona para formar un conjunto de valores posiblemente más pequeño:

reduce (k2, list (v2)) -> list (v3)

Cada llamada *reduce* produce típicamente bien un valor v3 o un valor vacío, aunque una misma llamada puede devolver más de un valor. Los resultados de las llamadas se recopilan en la lista de resultados buscada. Para ilustrar en líneas generales el funcionamiento de este modelo de programación, tómese de ejemplo el problema de contar el número de ocurrencias de cada palabra en un conjunto de documentos.

```
map(String key, String value):
// key: nombre del documento
// value: contenido del documento
for each word w in value:
EmitIntermediate(w, “1?);
reduce(String key, Iterator values):
// key: una palabra
// values: una lista de ocurrencias
int result = 0;
for each v in values:
```

```
result += ParseInt(v);
Emit(AsString(result));
```

Implementación de MapReduce

Son posibles muchas implementaciones distintas del modelo MapReduce. La elección correcta depende del entorno del trabajo. De este modo, una implementación será adecuada para una máquina de memoria compartida y otra para un conjunto de máquinas conectadas a una red.

Como la librería MapReduce está diseñada para ayudar en el procesamiento de un gran número de datos usando una gran cantidad de máquinas, esta debe estar capacitada para tolerar problemas como la paralelización, el control de concurrencia, las comunicaciones por red y la tolerancia a fallos. De hecho, lleva a cabo varias optimizaciones para disminuir el sobrecoste asociado a la planificación, comunicación de red y agrupamiento de resultados intermedios.

Las invocaciones *map* se distribuyen a lo largo de múltiples máquinas particionando automáticamente la información de entrada en un conjunto de *M* subconjuntos que pueden ser procesados en paralelo por distintas máquinas. Las invocaciones *reduce* se distribuyen particionando el espacio de claves intermedio en *R* trozos utilizando una función de particionado, así como un número de particiones *R*, que pueden ser relativas a cada usuario. El sistema de ejecución de MapReduce reparte las tareas *map* y *reduce* entre los recursos distribuidos. Ambos, los pares de entrada *map* y los pares de salida *reduce*, son almacenados en un sistema de ficheros distribuidos.

Cuando un usuario invoca una implementación de MapReduce, se ejecutarán por orden las siguientes acciones :

1. La biblioteca de funciones MapReduce que utilice el usuario en el programa primero divide los ficheros de entrada en *M* bloques de típicamente entre 16 y 64 Mb cada uno. A continuación inicia varias copias del programa en el clúster de máquinas.
2. Una de dichas copias es diferente del resto –la copia maestra. El resto son trabajadores que ejecutan las tareas asignadas por el maestro. Existen *M* tareas *map* y *R* tareas *reduce* para asignar. El maestro elegirá nodos "ociosos" y le asignará a cada uno una tarea *map* o una *reduce.*
3. Cada trabajador al que se le asigna una tarea *map* lee los contenidos del subconjunto correspondiente. Obtiene los pares clave-valor a partir de la información suministrada y se los pasa a la función *map* definida por el usuario. Estos pares clave-valor intermedios generados por la función *map* se almacenan en memoria caché.
4. Periódicamente, estos pares en memoria caché se escriben a disco, particionados en *R* regiones por la función de partición. Las localizaciones de estos pares en disco se transmiten de vuelta al maestro, que es el responsable de redirigirlas a los nodos trabajadores *reduce*.
5. Cuando a un nodo *reduce* le notifica el nodo maestro estas localizaciones, utiliza llamadas remotas para leer el contenido almacenado localmente en los discos de los trabajadores *map*. Cuando ha finalizado de leer toda la información, la ordena por las claves intermedias, de tal forma que todas las ocurrencias asociadas a la misma clave vayan al mismo grupo. Este ordenamiento es necesario porque típicamente muchas claves diferentes se asignan a la misma tarea *reduce*. Si la cantidad de información intermedia es demasiado grande para almacenarse en memoria, se utiliza un ordenamiento externo.
6. El nodo *reduce* itera sobre la información intermedia ordenada y por cada clave intermedia distinta encontrada, pasa dicha clave y el correspondiente conjunto de valores intermedios a la función *reduce* (también definida por el usuario). La salida de dicha función se añade a un fichero local de esta partición *reduce*.
7. Cuando todas las tareas *map* y *reduce* se hayan completado, el nodo maestro devuelve el control de nuevo

al código del usuario. Tras una ejecución exitosa, la salida del trabajo MapReduce está disponible en los *R* ficheros de salida (uno por tarea *reduce*, con nombres personalizables por el usuario). Por norma general, no hará falta combinar estos *R* ficheros en un único archivo pues se pasan a menudo como entrada a otra llamada MapReduce, o se utilizan desde otra aplicación distribuida que sea capaz de manejar información fraccionada en múltiples ficheros. [3]

Fiabilidad y tolerancia a fallos

MapReduce consigue fiabilidad en sus ejecuciones gracias a su estrategia de reparto de operaciones sobre el conjunto de datos asignados a cada nodo. Las operaciones individuales utilizan operadores atómicos para nombrar ficheros de salida como comprobación para asegurarse que no existen hilos conflictivos ejecutándose paralelamente. Cuando los ficheros se renombran, es posible también copiarlos con otro nombre adicional al nombre de la tarea (para evitar posibles problemas de efectos laterales). Asimismo, el nodo maestro intenta planificar operaciones *reduce* en el mismo nodo, o en el mismo rack en el que se encuentra el nodo que retiene la información sobre la que se está trabajando. Esta propiedad es deseable ya que conserva el ancho de banda de la red troncal del centro de datos.

Por su parte, la tolerancia a fallos es uno de los aspectos más importantes en un modelo de programación distribuido, ya que se está trabajando con volúmenes muy grandes de datos almacenados en decenas o centenas de máquinas. Se pueden distinguir dos tipos de fallos principalmente:

Fallo de un nodo trabajador

El nodo maestro mantiene varias estructuras de datos. Para cada tarea *map* o *reduce*, almacena el estado (*ocioso*, *en progreso* o *completado*) y la identidad del nodo trabajador. El nodo maestro es el túnel a través del que la localización de las regiones de los ficheros intermedios se propaga de los nodos *map* a los nodos *reduce*. Por tanto, para cada tarea *map* completada, el maestro almacena la localización y tamaños de las R regiones de ficheros intermedios generadas por las tareas *map*. Cuando estas tareas se completen, se actualizarán estas localizaciones y se recibirá información acerca del tamaño de datos procesados. Además, esta información se suministrará incrementalmente a los nodos trabajadores que ejecuten tareas *reduce* en estado *en progreso*. En caso de que el nodo maestro no reciba una respuesta de un trabajador en una cantidad de tiempo determinada, lo marcará como caído y redirigirá el trabajo asignado a ese nodo a otros. Cualquier tarea *map* completada por un nodo trabajador caído volverá a su estado inicial (*ocioso*), y por tanto se volverá seleccionable para planificarse en otro nodo trabajador. De igual forma, cualquier tarea *map* o *reduce* que se ejecute sobre un nodo trabajador caído se resetea a *ocioso* y se vuelve seleccionable.

Las tareas *map* completadas se re-ejecutan en caso de fallo debido a que sus resultados se almacenan en los discos locales de la máquina caída y se vuelven, por tanto, inaccesibles. Las tareas *reduce* completadas no necesitan ser re-ejecutadas ya que sus salidas se almacenan en un sistema de ficheros global. Cuando una tarea *map* se ejecuta primero en el nodo trabajador *A* y luego en el *B* (debido a que *A* ha fallado), todos los trabajadores que estén ejecutando tareas *reduce* son notificados de dicha reejecución. Cualquier tarea *reduce* que no haya leído todavía en ese momento la información del trabajador *A*, lo hará del *B*. De las anteriores afirmaciones, se puede concluir que MapReduce es tolerante a fallos a gran escala de nodos trabajadores.

Fallo del nodo maestro

Es sencillo hacer que el nodo maestro escriba puntos de control periódicos de las estructuras de datos descritas en el anterior epígrafe. Así, si la tarea maestro muere, se puede comenzar una nueva copia desde el último estado guardado. Sin embargo, dado que existe un único nodo maestro, se adopta de forma generalizada la decisión de abortar el trabajo MapReduce si ocurre esta incidencia y reiniciarlo cuando el usuario lo desee. [2]

Implementaciones

Existen multitud de implementaciones del modelo MapReduce de las que podemos destacar:

- Framework MapReduce de Google, implementado en *C++*, con interfaces en Python y Java.
- Hadoop, implementación open-source de MapReduce programada en Java que forma parte del proyecto Apache.
- Greenplum, implementación comercial de MapReduce, con soporte para Python, Perl, SQL y otros lenguajes.
- Phoenix, implementación de memoria compartida de Map*Reduce* escrita en *C*.
- Disco, implementación open-source de MapReduce desarrollada por Nokia. Su núcleo está escrito en *Erlang* y los trabajos se suelen escribir en *Python*.
- MARS, implementación de MapReduce para GPUs (*Graphical Processor Units*) de Nvidia empleando CUDA (*Compute Unified Device Architecture*), una arquitectura de computación paralela desarrollada por la compañía.

Indexado de gran escala

Uno de los usos más significativos de MapReduce ha sido una reescritura completa del sistema de indexado que produce las estructuras de datos usadas por el servicio web de búsquedas de Google. El Sistema de indexado toma como entrada un gran grupo de documentos almacenados como un grupo de archivos GFS. El contenido bruto de estos documentos es mayor de los 20TB de datos. El proceso de indexado corre como una secuencia de 5 a 10 operaciones MapReduce. El uso de MapReduce ha generado muchos beneficios entre los que destacan:

- El código de indexado es más simple, pequeño y fácil de comprender porque la parte que trata la tolerancia a fallos, distribución y paralelización está oculta en la librería MapReduce.
- El desempeño de la librería MapReduce es lo suficientemente bueno como para mantener cálculos no relacionados en separado, evitando mezclarlos para evitar barridos extra en los datos, mejorando así el tiempo de indexado exponencialmente.
- El proceso de indexado se ha vuelto más fácil de operar, debido a la automatización de la respuesta a los problemas por parte de la librería, y ha facilitado mejorar el desempeño del proceso de indexado añadiendo nuevas máquinas al clúster.

Conclusiones

El modelo de programación MapReduce ha sido usado para diferentes objetivos de forma exitosa. Este logro es atribuido en gran escala a la sencillez del modelo incluso para programadores sin experiencia en sistemas distribuidos y paralelos, debido a que oculta los detalles de la paralelización, la tolerancia a fallos, optimización y el balance de carga. Además de que una gran variedad de problemas son fácilmente expresables como procesos MapReduce. Este modelo es usado para la generación de datos, búsquedas, ordenamiento, minería de datos, máquinas de aprendizaje y otros muchos sistemas. La implementación hace uso eficiente de los recursos de la máquina y es óptima para una gran variedad de problemas computacionales.

Referencias

- [1] M. Miller. "*Cloud Computing. Web-Based Applications That Change the Way You Work and Collaborate Online*", Que, 2008.
- [2] J. Dean y S. Ghemawat. "*Mapreduce: Simplified data processing on large*
- *clusters*". Communications of the ACM, 52(1), págs. 107–113, 2008.

- [3] Alberto Luengo Cabanillas. "*Desarrollo de un entorno basado enMapReduce para la ejecución distribuida de algoritmos genéticos paralelos"*
- Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. *"High-performance sorting on networks of workstations."* In Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, May 1997.
- Douglas Thain, Todd Tannenbaum, and Miron Livny. *"Distributed computing in practice: The Condor experience. Concurrency and Computation: Practice and Experience"*, 2004.

Ing. Yixander Yero Tarancón: Ingeniero en Cinecias Informáticas. Departamento de Programación y Sistemas Digitales. Facultad 6. Universidad de las Ciencias Informáticas, Carretera a San Antonio de los Baños, km 2 ½, Torrens, Boyeros, La Habana, Cuba. CP.: 19370 Dirección postal. yyero@uci.cu