2017/11/30

Recuperación de Información y Recomendaciones en la Web

Ivory

Un conjunto de herramientas de Hadoop para la investigación de recuperación de información a escala web

Autor:

César Britos

Docente: Libertad Tansini

ontenido

Resumen

En este informe se desea hacer una introducción a Ivory, un conjunto de herramientas de Hadoop para la investigación de recuperación de información a escala web. Hadoop es una implementación de código abierto de MapReduce.

Este informe se enfocará en tres aspectos: una arquitectura de recuperación construida directamente en el Sistema de Archivos Distribuidos Hadoop (HDFS), un algoritmo escalable Map-Reduce para la indexación invertida y una clasificación de páginas web para mejorar la efectividad de la recuperación.

Introducción

Contexto

Se reconoce comúnmente que las colecciones a escala web han superado las capacidades de las máquinas individuales, lo que requiere el uso de clusters para abordar problemas básicos en la recuperación de información.

Los cálculos distribuidos son inherentemente difíciles de organizar, gestionar y razonar. Con los modelos de programación tradicionales como MPI, el desarrollador debe manejar explícitamente una gama de detalles a nivel de sistema, que van desde la sincronización hasta la distribución de datos a la tolerancia a fallas. Recientemente, Map-Reduce se ha convertido en una alternativa atractiva: su abstracción funcional proporciona un modelo fácil de entender para diseñar algoritmos escalables y distribuidos.

MapReduce se basa en la observación de que muchas tareas de procesamiento de información tienen la misma estructura básica: se aplica un cálculo sobre una gran cantidad de registros (por ejemplo, páginas web) para generar resultados parciales, que luego se agregan de alguna manera. Inspirándose en las funciones de orden superior de la programación funcional, MapReduce proporciona una abstracción para "mapeadores" definidos por el programador (que especifican el cálculo por registro) y "reductores" (que especifican la agregación de resultados). Los pares clave-valor forman las primitivas de procesamiento. El mapeador se aplica a cada par clave-valor de entrada para generar un número arbitrario de pares clave-valor intermedios. El reductor se aplica a todos los valores asociados con la misma clave intermedia para generar un número arbitrario de pares clave-valor finales como salida.

Bajo este marco, un programador solo necesita proporcionar implementaciones del mapeador y el reductor. Además de un sistema de archivos distribuido, el marco de

ejecución maneja de forma transparente todos los demás aspectos de la ejecución en clústeres que van desde unos pocos a unos pocos miles de núcleos. Es responsable, entre otras cosas, de la programación (cambio de código a datos), el manejo de fallas y el gran problema de clasificación distribuida y mezcla (sort y shuffle) entre el la fases de mapeo y de reducción por las cuales los pares clave-valor intermedios deben agruparse por clave.

Hadoop, la implementación de código abierto de MapReduce, ha ganado popularidad como un marco accesible, rentable para el procesamiento de grandes conjuntos de datos. Este documento describe tanto a hadoop en forma concisa y explica un intento de construir un sistema de recuperación distribuidos alrededor del ecosistema Hadoop.

El sistema que se analizara se llama Ivory, integra el motor de recuperación SMRF (Búsqueda mediante Markov Random Fields) de Metzler. Ivory se ha publicado bajo una licencia de código abierto y puede ser descargado libremente desde la web. Se discuten tres aspectos fundamentales sobre la implementación: una arquitectura de recuperación construida directamente en HDFS, un algoritmo escalable de MapReduce para indexación invertida y post-procesamiento de resultados para suprimir contenido adulto, spam y baja calidad páginas (ya que es lo que se propuso en el paper base del informe).

HADOOP

Como ya se mencionó, Ivory está implementado sobre Hadoop, por lo cual creo necesario explicar algunos componentes fundamentales de dicho framework.

EL PODER DE HADOOP

Hadoop provee una solución económica y atractiva para el almacenamiento de Big Data, así como también un modelo de proceso escalable para análisis de procesos. Específicamente tiene:

- **Un modelo económico y escalable de almacenamiento,** al aumentar los volúmenes de datos, también lo hace el costo de almacenarlo online. Hadoop al usar commodity hardware con commodity disks, disminuye el precio del terabyte comparado con cualquier otra tecnología.
- **Capacidad de IO masiva y escalable,** Hadoop al usar un gran número de dispositivos de bajo costo, el agregado de IO y la capacidad de red es mayor al provisto por arreglos de almacenamiento dedicados en los que un pequeño número de grandes discos son provistos por un número aún menor de procesadores. Además, agregar nuevos servidores a Hadoop agrega almacenamiento, IO, CPU y capacidad de red de una sola vez, mientras que el

agregado de discos para almacenamiento pondría simplemente el cuello de botella de la CPU o de la red.

- **Confiabilidad,** los datos en Hadoop son almacenados redundantemente en múltiples servidores y pueden ser distribuidos a través de múltiples racks. La falla de un server no resulta en una pérdida de datos, de hecho, Hadoop continuará funcionando incluso si un servidor falla (el procesado simplemente cambiará a otro servidor).
- **Un modelo de procesamiento escalable,** MapReduce representa un modelo de procesamiento distribuido altamente aplicable y escalable. Aunque MapReduce no es la implementación más eficiente para todos los algoritmos, es capaz de tener un desempeño aceptable aplicando fuerza bruta para la mayoría de ellos.
- **Esquema en lectura,** los datos pueden ser cargados a Hadoop sin tener que tener que convertirlos a un formato de estructura normalizado. Esto facilita a Hadoop la ingesta rápida de gran cantidad de datos de varias formas. La imposición de la estructura puede ser demorada hasta que los datos son accedidos, esto usualmente se refiere a schema on read, contrariamente al schema on write de las bases de datos relacionales.

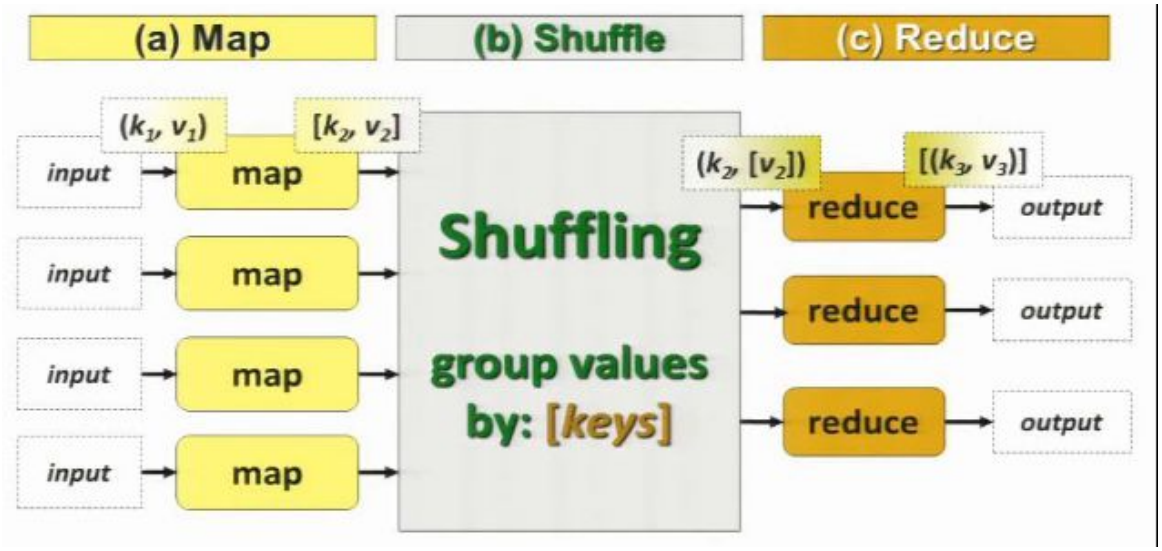
Arquitectura

En su núcleo, Hadoop tiene dos capas principales:

- Informática/Computación capa (MapReduce), y
- Capa de almacenamiento (Hadoop Distributed File System).

MapReduce

MapReduce es un modelo de programación paralela para escribir las aplicaciones distribuidas de Google para garantizar un proceso eficaz de grandes cantidades de datos (juegos de datos de varios terabytes), en grandes grupos (miles de nodos) de hardware básico de manera segura, tolerante a fallos. El programa se ejecuta en MapReduce Hadoop Apache que es un marco de código abierto.



Hadoop Distributed File System

El Hadoop Distributed File System (HDFS) se basa en el Google File System (GFS) y proporciona un sistema de ficheros distribuido que está diseñado para ejecutarse en hardware asequible. Tiene muchas similitudes con los actuales sistemas de archivos distribuidos. Sin embargo, las diferencias con otros sistemas de ficheros distribuidos son importantes. Es muy tolerante a errores y está diseñado para ser instalado en hardware de bajo costo. Proporciona un alto rendimiento en el acceso a datos de aplicaciones y es adecuado para aplicaciones con grandes conjuntos de datos.

Aparte de los mencionados dos componentes básicos, el marco Hadoop incluye también los dos módulos siguientes:

- Hadoop common: Estas son las bibliotecas de Java y las utilidades requeridas por otros módulos Hadoop.
- Hadoop YARN: Este es un marco para la planificación de tareas y administración de recursos de clúster.

MapReduce

MapReduce es un framework que proporciona un sistema de procesamiento de datos paralelo y distribuido basada en java. El algoritmo MapReduce contiene dos tareas importantes, Map y Reduce. Cuando se realiza el Map, se toman un conjunto de datos y se convierten en otro conjunto de datos, en el que los elementos se dividen en tuplas (pares clave/valor). Por otro lado, al Reduce, se toma la salida de un Map como entrada y se combinan las tuplas de datos en un conjunto más pequeño de tuplas. En MapReduce como el nombre lo implica, el Reduce se realiza siempre después que el Map.

La principal ventaja de MapReduce es que es fácil de escalar realizando el procesamiento de datos en múltiples nodos. En el modelo MapReduce, el procesamiento de datos primitivos son llamados mapas y reductores.

La descomposición de una tarea de procesamiento de datos en Map y reductores a veces es no trivial. Pero, una vez que se logra descomponer en el MapReduce adecuadamente, la tarea se ejecuta en cientos, miles o incluso decenas de miles de máquinas en un clúster simplemente realizando un cambio de configuración. Esta escalabilidad sencilla es lo que ha atraído a muchos programadores a usar el modelo MapReduce.

MapReduce tiene una arquitectura Maestro / Esclavo. Cuenta con un servidor maestro o JobTracker y varios servidores esclavos o TaskTrackers, uno por cada nodo del clúster

El JobTracker es el punto de interacción entre los usuarios y el framework MapReduce. Los usuarios envían trabajos MapReduce al JobTracker, que los pone en una cola de trabajos pendientes y los ejecuta en el orden de llegada. El JobTracker gestiona la asignación de tareas y delega las tareas a los TaskTrackers. Los TaskTrackers ejecutan tareas bajo la orden del JobTracker y también manejan el movimiento de datos entre la fase Map y Reduce.

Para ver las diferencias entre JobTracker y TaskTracker vamos a ver las características de cada uno.

JobTracker

- Capacidad para manejar metadatos de trabajos
- Estado de la petición del trabajo
- Estado de las tareas que se ejecutan en TaskTracker
- Decide sobre la programación
- Hay exactamente un JobTracker por cluster.
- Recibe peticiones de tareas enviadas por el cliente.

- Programa y monitoriza los trabajos MapReduce con TaskTrackers.

TaskTracker

- Ejecuta las solicitudes de trabajo de JobTrackers
- Obtiene el código que se ejecutará
- Aplica la configuración específica del trabajo
- Comunicación con el JobTracker:
- Envíos de la salida, finalizar tareas, actualización de tareas, etc.

El algoritmo

El programa MapReduce se ejecuta en tres etapas, a saber: etapa Map, shuffle, y Reduce.

Etapa Map: La función Map recibe como parámetros un par de (clave, valor) y devuelve una lista de pares. Esta función se encarga del mapeo y se aplica a cada elemento de la entrada de datos, por lo que se obtendrá una lista de pares por cada llamada a la función Map. Después se agrupan todos los pares con la misma clave de todas las listas, creando un grupo por cada una de las diferentes claves generadas. No hay requisito de que el tipo de datos para la entrada coincida con la salida y no es necesario que las claves de salida sean únicas.

Map (clave1, valor1) –> lista (clave2, valor2)

La operación de Map se paraleliza, el conjunto de archivos de entrada se divide en varias tareas llamado FileSplit, ver Figura 2, el tamaño típico de bloque es de 128MB. Las tareas se distribuyen a los nodos TaskTrackers, y estos a su vez pueden realizar la misma tarea si hiciera falta.

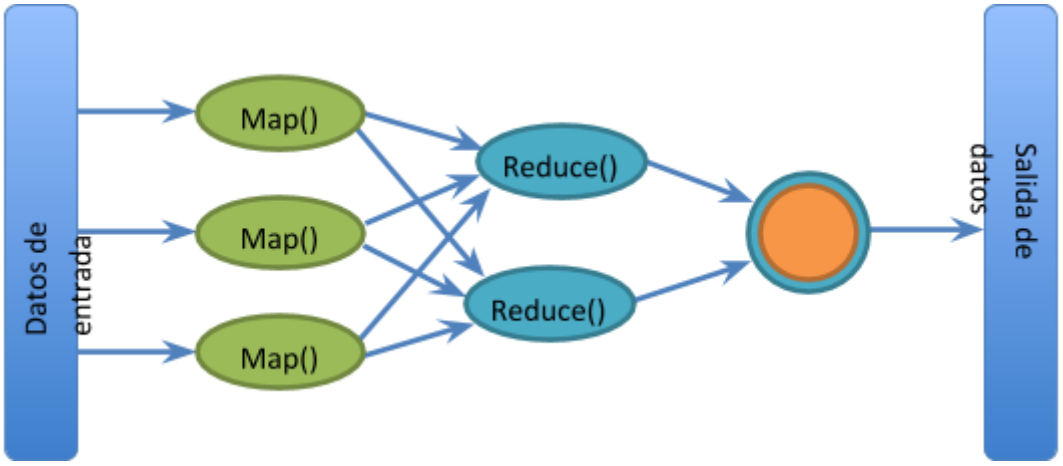
Etapa Reduce: La función Reduce se aplica en paralelo para cada grupo creado por la función Map().La función Reduce se llama una vez para cada clave única de la salida de la función Map. Junto con esta clave, se pasa una lista de todos los valores asociados con la clave para que pueda realizar alguna fusión para producir un conjunto más pequeño de los valores.

Reduce (clave2, lista(valor2)) –> lista(valor2)

Cuando se inicia la tarea reduce, la entrada se encuentra dispersa en varios archivos a través de los nodos en las tareas de Map. Los datos obtenidos de la fase Map se ordenan para que los pares clave-valor sean contiguos (fase de ordenación, sort fase), esto hace que la operación Reduce se simplifique ya que el archivo se lee secuencialmente. Si se ejecuta el modo distribuido estos necesitan ser primero copiados al filesystem local en la fase de copia. Una vez que todos los datos están disponibles a

nivel local se adjuntan a una fase de adición, el archivo se fusiona (merge) de forma ordenado. Al final, la salida consistirá en un archivo de salida por tarea reduce ejecutada.

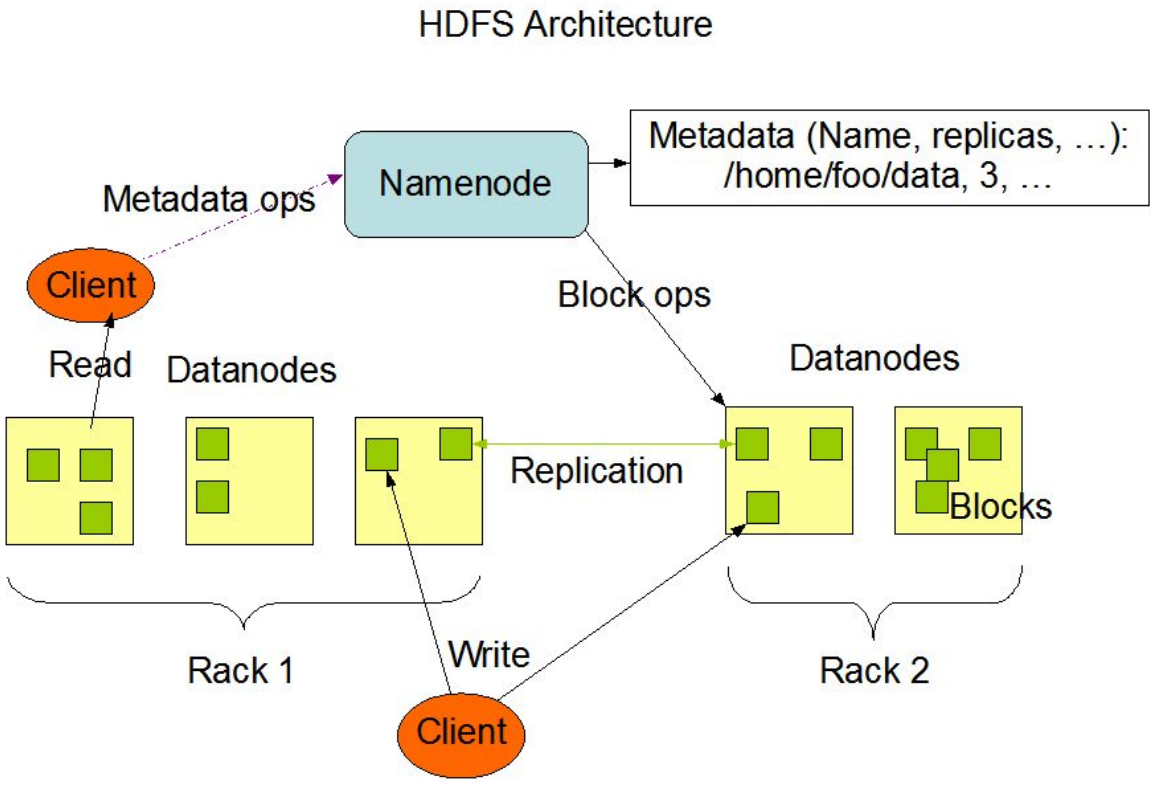
Por lo tanto, N archivos de entrada generará M mapas de tareas para ser ejecutados y cada mapa de tareas generará tantos archivos de salida como tareas Reduce hayan configuradas en el sistema.



Sistema de archivos

El sistema de arquivos de Hadoop se ha desarrollado utilizando diseño de sistema de archivos distribuidos. A diferencia de otros sistemas distribuidos, HDFS es muy tolerante a fallos y esta diseñado para ser utilizado en hardware de bajo coste.

HDFS es un sistema de archivos distribuidos, escalable y portátil escrito en Java y creado especialmente para trabajar con archivos de gran tamaño. Una de las principales características es un tamaño de bloque muy superior al habitual (128 MB) para no perder tiempo en los accesos de lectura.



HDFS tiene una arquitectura Maestro/Esclavo y se compone de los siguientes elementos:

Namenode

El Namenode es la pieza central del sistema de archivos HDFS. Este mantiene el árbol de direcciones de todos los archivos del sistema de archivos, y el seguimiento a través de todo el cluster de la ubicación de los datos. No almacena los datos de esos archivos.

Una aplicación cliente habla con el Namenode cuando desea saber la ubicación de un archivo o cuando quiere agregar, copiar, mover o borrar un archivo. El Namenode responde satisfactoriamente la petición retornando la lista de servidores Datanodes donde se encuentran los datos.

El Namenode es un único punto de fallo para los cluster HDFS. HDFS no es actualmente un sistema de alta disponibilidad. Cuando un Namenode se cae, el sistema de archivos se vuelve offline. Hay un segundo Namenode opcional que puede ser hosteado en una máquina separada. Solo crea checkpoints de espacios de nombres al combinar el archivo de ediciones en el archivo fsimage y no proporciona ninguna redundancia real.

Datanode

Un DataNode almacena datos en el HDFS. Un sistema de archivos funcional tiene más de un DataNode, con datos replicados a través de ellos.

Al inicio, un DataNode se conecta al NameNode. A continuación, responde a las solicitudes del Namenode para las operaciones del sistema de archivos. Permiten realizar operaciones tales como creación, supresión, con lo que la replicación de acuerdo con las instrucciones del Namenode.

Las aplicaciones cliente pueden hablar directamente con un DataNode, una vez que Namenode ha proporcionado la ubicación de los datos. De forma similar, las operaciones de MapReduce implementadas en instancias de TaskTracker cerca de un DataNode, hablan directamente con el DataNode para acceder a los archivos. Las instancias de TaskTracker deberían implementarse en los mismos servidores que alojan las instancias de DataNode, de modo que las operaciones de MapReduce se realicen cerca de los datos.

Las instancias de DataNode pueden comunicarse entre sí, que es lo que hacen cuando replican datos.

Generalmente no hay necesidad de usar el almacenamiento RAID para datos DataNode, ya que los datos están diseñados para replicarse en varios servidores, en lugar de múltiples discos en el mismo servidor.

Una configuración ideal es que un servidor tenga un DataNode, un TrackTracker y, luego, discos físicos, una ranura de TrackTracker por CPU. Esto permitirá que cada Rastreador de tareas sea 100% de una CPU, y discos separados para leer y escribir datos.

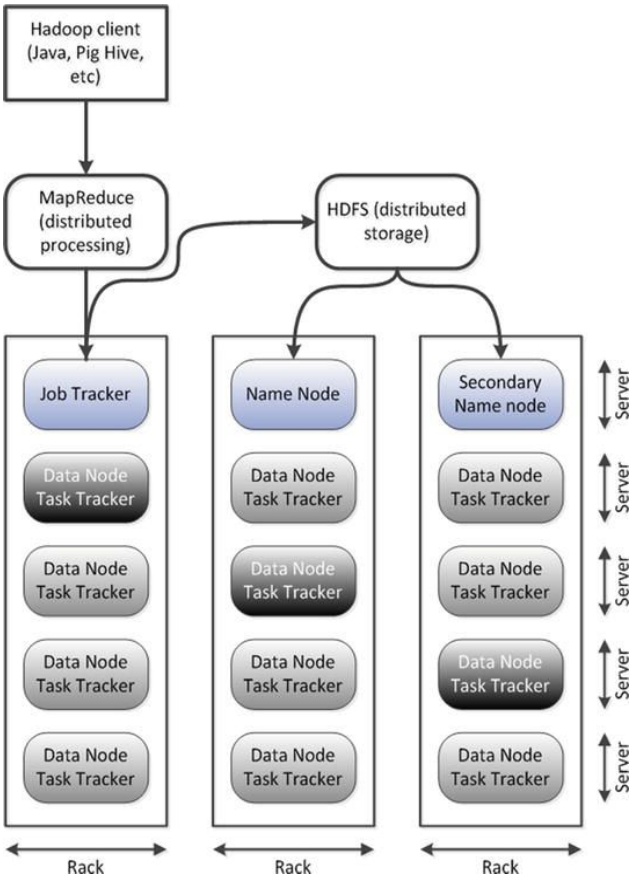
Blocks (Bloque)

En general los datos de usuario se almacenan en los archivos de HDFS. El archivo en un sistema de archivos se divide en uno o más segmentos y/o almacenados en los nodos de datos. Estos segmentos se denominan como bloques. En otras palabras, la cantidad mínima de datos que HDFS puede leer o escribir se llama un bloque. El tamaño de bloque por defecto es de 128 MB, pero puede ser aumentado por la necesidad de cambiar de configuración HDFS. Cada bloque se replica un número especificado de veces. Las réplicas de los bloques se almacenan en diferentes DataNodes elegidos para reflejar la carga en un DataNode, así como para proporcionar velocidad de transferencia y resiliencia en caso de falla de un rack.

Características de los HDFS

- **Detección de fallos y recuperación:** HDFS incluye un gran número de componentes de hardware, los fallos de componentes son frecuente. Por lo tanto, HDFS debe contar con mecanismos para una rápida y automática detección de fallos y recuperación.
- **Conjuntos muy grandes:** HDFS debe tener cientos de nodos por clúster para administrar las aplicaciones de grandes conjuntos de datos.
- **Hardware a una velocidad de transferencia de datos:** una tarea solicitada se puede hacer de una manera eficiente, cuando el cálculo se lleva a cabo cerca de los datos. Especialmente en los casos en que se tratan grandes conjuntos de

datos, reduce el tráfico de red y aumenta el rendimiento.

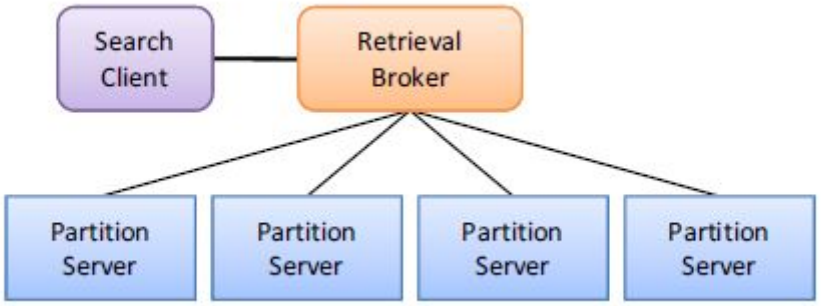


Objetivos del Informe

El objetivo es mostrar una posible alternativa desafiando la sabiduría convencional al explorar si es realmente factible "ejecutar" algoritmos de evaluación de consultas directamente en índices almacenados en HDFS. Además, se demuestra que es posible usar el mismo clúster de Hadoop para el procesamiento orientado por lotes (por ejemplo, indexación) y para servicios en tiempo real (por ejemplo, recuperación).

Estado del Arte

La arquitectura de recuperación particionada de documentos estándar es como indica la siguiente figura:



Dada una consulta del usuario, la recuperación implica obtener listas de publicaciones correspondientes a los términos de la consulta y calcular los puntajes de los documentos de consulta de acuerdo con el modelo de recuperación especificado.

Más allá de colecciones de cierto tamaño, no es práctico almacenar todo el índice en una sola máquina. La solución estándar distribuida es una arquitectura de recuperación mediada por documentos mediada por brokers, ilustrada en la Figura anterior.

Toda la colección de documentos se divide en varias particiones (a veces llamadas "fragmentos"), y los índices se crean para cada partición por separado; un servidor es responsable para buscar cada índice, independientemente de los demás. Las interacciones entre un cliente de búsqueda y los servidores de partición están mediadas por el broker. En el ciclo estándar de consulta y respuesta, el cliente emite una consulta al broker, que luego distribuye la consulta a todos los servidores de partición en paralelo. Cada servidor calcula una lista clasificada en su partición de documento asignada de forma independiente, y los resultados se transmiten al broker. El broker fusiona los resultados y devuelve la lista clasificada final al cliente.

Sabiduría Convencional vs Hadoop

Existe un desajuste fundamental entre la arquitectura de recuperación particionada de documentos estándar y las características del entorno MapReduce.

Primero, se debe considerar el problema de la evaluación de consultas en cada partición individual. MapReduce, que se diseñó para el procesamiento por lotes, no es apropiado para esta tarea. En Hadoop, los mapeadores pueden tardar decenas de segundos incluso en iniciarse, ya que las tareas deben ponerse en cola en el jobTracker antes de que puedan asignarse a job individuales. Además, el diseño actual de Hadoop limita la

velocidad a la que se pueden generar nuevas Maps Tasks. Para la latencia de consulta sub-segundo que esperan los buscadores de hoy, no existe una forma obvia de implementar algoritmos de recuperación viables en MapReduce.

Además, el ecosistema del software MapReduce presenta desafíos adicionales para los algoritmos de recuperación en tiempo real. Un componente integral de MapReduce es el sistema de archivos distribuido subyacente (DFS), que se diseñó en torno a una serie de suposiciones sobre la carga de trabajo. Dado que se supone que los trabajos de MapReduce realizan un procesamiento por lotes de grandes conjuntos de datos, el sistema de archivos distribuidos se optimizó para un alto rendimiento sostenido y no para un acceso aleatorio de baja latencia.

El DFS emplea una arquitectura simple maestro-esclavo y almacena archivos en bloques de tamaño fijo. El maestro (llamado namenode en HDFS) almacena metadatos y asignaciones de espacios de nombres a bloques de datos, que a su vez están almacenados en los discos locales de los esclavos (llamados datanode en HDFS). El maestro solo comunica metadatos; la transferencia de datos ocurre directamente entre el cliente de la aplicación y el datanode relevante. En la medida de lo posible, el programador MapReduce inicia las tareas de asignación en las máquinas que contienen el bloque de datos para su procesamiento, garantizando así un alto rendimiento sostenido ya que la tarea se lee desde el disco local. Este diseño dificulta el acceso aleatorio de baja latencia a los datos DFS desde un cliente de aplicación arbitrario (por ejemplo, un servidor de partición). Para acceder a una posición aleatoria en un archivo (por ejemplo, buscar una lista de publicaciones), el cliente primero debe contactar al namenode para localizar el bloque de datos relevante. Luego, el cliente debe contactar al datanode apropiado para obtener los datos solicitados. Además de una búsqueda de disco en el datanode, todo el proceso implica comunicaciones de ida y vuelta con varias máquinas y transferencia de datos a través de la red. Este problema no se puede resolver simplemente ejecutando el cliente de la aplicación en el datanode que tiene el bloque almacenado localmente. El sistema de archivos distribuidos, por diseño, propaga los bloques de datos a través de los nodos en el clúster (para garantizar la fiabilidad, para proporcionar una mejor localidad, etc.) y, por lo tanto, incluso para archivos moderadamente grandes, ningún nodo de datos sostendrá la totalidad de los contenidos de un archivo.

El diseño del sistema de archivos distribuidos está directamente en desacuerdo con los requisitos para la evaluación de consultas, ya que es necesario el acceso aleatorio de baja latencia a las publicaciones. A pesar de que MapReduce proporciona un buen marco para construir índices invertidos, la discusión anterior sugiere que el DFS

constituye un sustrato de almacenamiento deficiente para la recuperación. Esta es de hecho la sabiduría convencional.

La solución típica a este problema es emplear una arquitectura separada para la recuperación. Una vez que los índices se han creado usando Hadoop y se han escrito en HDFS, luego se copian en otro clúster (en sistemas de archivos POSIX estándar) para admitir la recuperación. Normalmente, esto implica copiar índices de partición individuales en el disco local del servidor de partición correspondiente. Esta solución, aunque ciertamente factible, adolece de dos inconvenientes importantes, que se analizan a continuación.

Primero, esta solución requiere el mantenimiento de dos arquitecturas separadas: una para el procesamiento por lotes y otra para la consulta en tiempo real. Esto también requiere la división de los recursos de hardware, lo que hace que sea difícil llevar toda la capacidad disponible a un gran problema. Aunque es posible que las mismas máquinas físicas sirvan "doble función", una configuración de este tipo puede tener efectos de rendimiento impredecibles ya que varios procesos compiten por los mismos núcleos, memoria, disco y red. Además, mantener arquitecturas independientes inevitablemente requerirá mantener múltiples copias de los datos.

Por ejemplo, la colección debe residir en HDFS para admitir la indexación, pero puede ser necesaria una copia por separado en el clúster de recuperación para que los usuarios puedan examinar los resultados.

En segundo lugar, la solución de dos arquitecturas genera un flujo de trabajo complejo que necesita copiar grandes índices a través de la red, lo que complica la gestión de datos. Tal configuración requiere un buen mecanismo para control de versiones y metadatos, ya que los datos duplicados pueden residir en sistemas independientes en cualquier momento dado. La gestión del flujo de trabajo es notoriamente difícil en un entorno de investigación en rápida evolución.

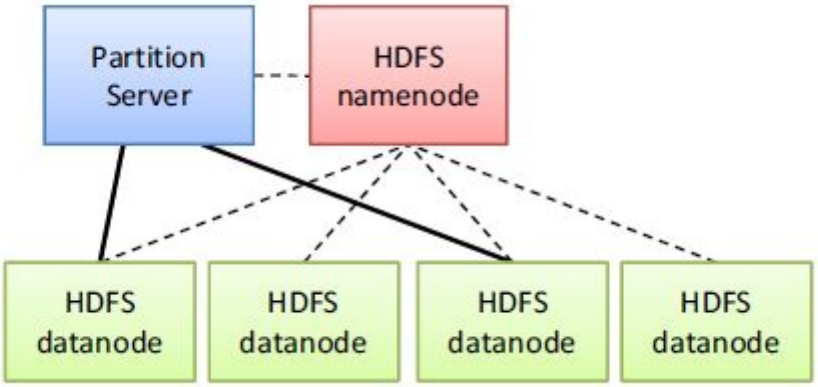
Además, las latencias no triviales involucradas en la copia de índices a través de la red en discos locales no son conducentes a los rápidos tiempos de respuesta necesarios para los experimentos de IR.

Al desarrollarse el sistema Ivory, se decidio cuestionar la sabiduría convencional y explorar si era realmente factible "ejecutar" algoritmos de evaluación de consultas directamente en índices almacenados en HDFS. Además, se preguntaron si era posible usar el mismo clúster de Hadoop para el procesamiento orientado por lotes (por ejemplo, indexación) y para servicios en tiempo real (por ejemplo, recuperación).

Se hicieron dos observaciones adicionales que llevaron a creer que esta arquitectura al menos valía la pena intentarlo. La primera parte de la evidencia proviene de BigTable, que es un mapa ordenado, disperso, distribuido y persistente multidimensional construido sobre el sistema de archivos de Google. BigTable se utiliza para una serie de servicios de producción con requisitos de baja latencia (por ejemplo, Google Earth).

Aunque es muy diferente de la arquitectura de recuperación distribuida que exploramos aquí, BigTable demuestra que no existe una razón de principio para que las aplicaciones de nivel superior no puedan ocultar las latencias DFS. La segunda evidencia proviene de la arquitectura de clúster físico: el ancho de banda entre una máquina y los discos de cualquier otra máquina rack local es sorprendentemente competitivo con respecto al ancho de banda de los discos locales (ya que, en su mayoría, los switches de nivel de rack no son sobresuscrito internamente). Operativamente, esto significa que leer datos del disco de otra máquina en el mismo bastidor no es mucho más lento que leer datos del disco local.

El componente de recuperación en Ivory obtiene las publicaciones directamente desde HDFS en lugar del disco local. Esto se muestra en la siguiente Figura, que se centra en un servidor de partición individual.



Como es estándar en la mayoría de los motores de recuperación, el vocabulario se guarda en la memoria. Con la codificación frontal, esto es relativamente fácil de lograr, incluso para colecciones grandes. El vocabulario mantiene desplazamientos de bytes en archivos de índice almacenados en HDFS que corresponden a ubicaciones de listas de publicaciones. La obtención de una lista de contabilizaciones implica primero ponerse en contacto con el namenode para la ubicación del bloque y, luego, contactar con el datanode para obtener los datos reales, lo cual no es diferente de cualquier otra lectura de HDFS.

Dentro de un entorno de clúster de Hadoop, aún se debe abordar el problema de cómo se inicializan los servidores de partición y el broker, dado que el único punto de contacto entre un cliente y el clúster de Hadoop es el jobTracker. La solución que diseñaron involucra embeber servidores en trabajos MapReduce (aunque degenerados en la mayoría de los casos).

Los servidores de partición pueden generarse como un trabajo de MapReduce que ejecuta mapeadores pero no reductores. Incrustado dentro de cada mapper se encuentra un servidor que maneja las consultas a través de una conexión TCP y accede a las publicaciones directamente en HDFS (como se describe arriba). Para iniciar varios servidores de partición, creamos un trabajo Map-Reduce que se mapea con un archivo de configuración que especifica las ubicaciones de los índices de partición. Al configurar apropiadamente el trabajo, se genera un número de mapeadores igual al número de particiones de documentos. Cada mapeador lee en la ubicación del índice de partición, inicializa un motor de consulta, y luego se inicia en un bucle de servicio infinito a la espera de conexiones TCP entrantes. El runtime de Hadoop es, en esencia, cooptado para servir como un simple scheduler.

Sin embargo, así se tiene poco control sobre los nodos de clúster en los que se inician los mapeadores. Afortunadamente, esta situación se corrige fácilmente: cuando se inicia cada mapeador, primero escribe su información de host en una ubicación DFS conocida. Después de que se hayan inicializado todos los servidores de partición, el broker se puede iniciar como un trabajo MapReduce de 1-mapper / 0-reducer, leyendo la información de host de todos los servidores de partición y completando la arquitectura del broker distribuido.

Esta solución aborda muchos de los problemas con la solución de dos arquitecturas discutida con anterioridad en está misma sección. En lugar de mantener un clúster de Hadoop para la indexación y otro clúster para la recuperación, podemos lograr ambos dentro de un ambiente homogéneo. Esto nos permite utilizar mejor los recursos de hardware disponibles y simplifica la administración de datos y flujo de trabajo. La desventaja potencial es, por supuesto, el rendimiento de consulta degradada debido a la lectura de las publicaciones de forma remota.

Índice Invertido

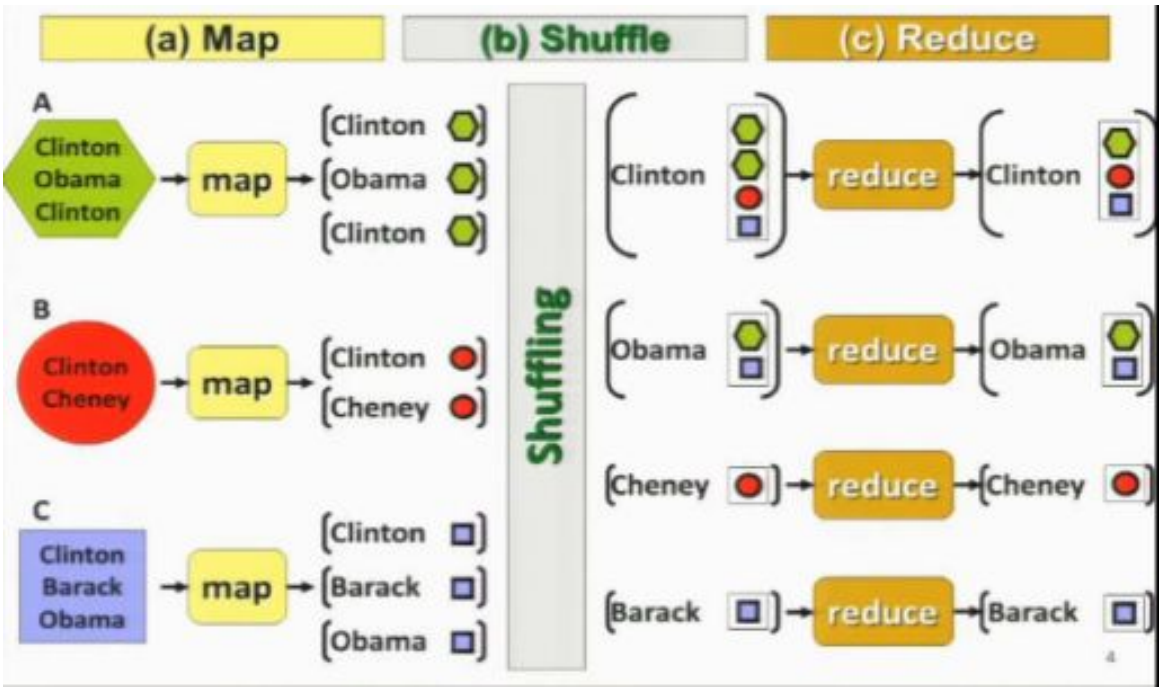
El algoritmo básico MapReduce para la indexación invertida se muestra en la Figura siguiente.

```
1: class Mapper
2:     method Map(docno n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)
1: class Reducer
2:     method Reduce(term t, postings [⟨n1, f1⟩ . . .])
3:         P ← new List
4:         for all posting ⟨n, f⟩ ∈ postings [⟨n1, f1⟩ . . .] do
5:             P.Append(⟨n, f⟩)
6:         P.Sort()
7:         Emit(term t, postings P)
```

Como entrada a los mapeadores tenemos los pares de, números de documento (claves) junto con el contenido del documento (valores). Dentro del mapeador, cada documento se tokeniza, deriva y filtran stopwords. Los términos se procesan secuencialmente para construir un histograma de frecuencias de términos (implementado como una matriz asociativa). El algoritmo luego itera sobre todos los términos: para cada uno, un posting que consiste en el número de documento y la frecuencia de término es creado (indicada por n, H {t} en el pseudocódigo). El mapeador luego emite un par clave-valor intermedio con el término como la clave y el posting como el valor. En este caso sencillo, la carga útil de cada publicación incluye solamente la TF, pero esto puede ser fácilmente ampliada con información de posición del término para construir índices posicionales.

En la fase de ordenar y mezclar (sort and shuffle), el runtime de MapReduce realiza un "group by" grande y distribuido de las publicaciones por término. Sin ningún esfuerzo adicional por parte del programador, el marco de ejecución reúne todas las publicaciones asociadas con el mismo término. Esto simplifica enormemente la tarea del reducer, que recoge las publicaciones y las escribe en el disco. El reducer comienza inicializando una lista vacía y luego agrega todas las publicaciones asociadas con el mismo término (clave) a la lista. Las publicaciones se ordenan (dependiendo del tipo de índice, por número de documento o frecuencia de término) y se escriben en el disco (comprimidas adecuadamente).

El modelo de programación de MapReduce proporciona una expresión muy concisa del algoritmo de indexación invertida, y puede implementarse en un par de docenas de líneas de código en Hadoop. Un ejemplo más visual del funcionamiento se adjunta a continuación.



En un indexador tradicional (es decir, no implementado en MapReduce), se debe dedicar gran atención a la tarea de agrupar contabilizaciones por término, dadas las restricciones impuestas por la memoria y el disco (que la capacidad de memoria es limitada, las búsquedas de disco son lentas, las operaciones secuenciales son preferidas , etc.) En MapReduce, el programador no tiene que preocuparse por ninguno de estos problemas: el tratamiento pesado de la agrupación de publicaciones es manejado por el runtime.

Algoritmo de indexación escalable de MapReduce

Sin embargo, hay un cuello de botella significativo en el algoritmo básico de MapReduce para la indexación invertida: supone que hay suficiente memoria para contener todas las publicaciones asociadas con el mismo término. Dado que el marco de ejecución de MapReduce no garantiza el orden de los valores asociados con la misma clave, el reducer primero debe almacenar todas las publicaciones y luego realizar una ordenación en memoria antes de que las publicaciones puedan escribirse en el disco.

Dado que Ivory crea índices ordenados por documentos, restringimos nuestra atención al problema de ordenar las publicaciones por el número de documento ascendente. Dado que el runtime garantiza que las claves lleguen a cada reducer en orden, una forma de superar el cuello de botella de escalabilidad es permitir que el runtime Map-Reduce haga la clasificación. En lugar de emitir pares de valores clave del siguiente tipo:

(término t, posting n, f)

Emitimos pares intermedios clave-valor del tipo:

(tupla t, n, tf f)

En otras palabras, la clave es una tupla que contiene el término y el número de documento, y el valor es el término frecuencia. Es necesario redefinir el orden de clasificación de manera que las claves se clasifican primero por término t, y luego por DOCNO n. Además, necesitamos un particionador personalizado para garantizar que todas las tuplas con el mismo término se mezclen con el mismo reducer. Con estos dos cambios, el marco de ejecución de MapReduce garantiza que las publicaciones lleguen en el orden correcto. Esto, combinado con reducers que preservan el estado a través de múltiples claves, permite que las publicaciones comprimidas se escriban con un uso de memoria mínimo.

```
1: class Mapper
2:     method Map(docno n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(tuple ⟨t, n⟩, tf H{t})
```

```
1: class Reducer
2:     method Initialize
3:         t_prev ← ∅
4:         P ← new PostingsList
5:     method Reduce(tuple ⟨t, n⟩, tf [f])
6:         if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:             Emit(term t, postings P)
8:             P.Reset()
9:         P.Add(⟨n, f⟩)
10:        t_prev ← t
11:    method Close
12:        Emit(term t, postings P)
```

El algoritmo de indexación invertido MapReduce revisado se muestra en la figura anterior. El mapeador permanece sin cambios en su mayor parte, a excepción de las diferencias en los pares de valores clave intermedios. El reductor contiene dos métodos adicionales: Initialize, que se llama antes sean procesadas las claves, y Close, que se llama después de que se procesa la clave final. El método Reduce se llama para cada clave (es decir, t, n), y por diseño, no sólo será un valor asociado con cada clave. Para cada par clave-valor, una publicación se puede agregar directamente a la lista de publicaciones. Dado que se garantiza que las publicaciones llegarán en el orden correcto, pueden codificarse de forma incremental en forma comprimida, lo que garantiza una pequeña footprint de memoria.

Finalmente, cuando las publicaciones asociados con el mismo término se han procesado, la lista de t se escribe en HDFS. La lista de publicaciones final debe ser escrito en el método Close.

En este algoritmo, el espacio de claves se divide por términos; es decir, todas las claves con el mismo término se envían al mismo reducer. Como en Hadoop cada reducer escribe su salida en un archivo separado en HDFS, nuestro índice final se divide en

archivos r, donde r es el número de reducers. En otro paso de Map-Reduce sobre estos archivos, construimos un índice de publicaciones posteriores para almacenar la posición de desplazamiento de bytes de cada lista de publicaciones.

Esto se usa durante la recuperación para buscar publicaciones que corresponden a los términos de la consulta. No hay necesidad de consolidar los archivos r, ya que el índice hacia delante de publicaciones puede realizar un seguimiento de en qué archivo se encuentra la lista de publicaciones de un término.

Tres detalles más completan la descripción del algoritmo de indexación MapReduce de Ivory en el paper estudiado: información de posición, datos de longitud del documento y configuración de parámetros para la compresión de Golomb. En primer lugar, los índices de posición pueden ser construidos mediante la simple sustitución del valor intermedio f (frecuencia termino) con un array de posiciones de termino; de lo contrario, no se requieren modificaciones adicionales al algoritmo.

En segundo lugar, dado que casi todos los modelos de recuperación tienen en cuenta la longitud del documento, esta información debe ser calculada. Si bien es sencillo expresar este cálculo como otro trabajo de MapReduce, esta tarea se puede plegar en el proceso de indexación invertida. Al procesar los términos en cada documento, la longitud del documento es conocida y puede escribirse como "datos secundarios" directamente en HDFS. Aprovechamos la capacidad de un mapeador para mantener el estado en el procesamiento de varios documentos de la siguiente manera: se crea una matriz asociativa en memoria para almacenar las longitudes de los documentos, que se completa a medida que se procesa cada documento.

Cuando el mapeador termina de procesar los registros de entrada, las longitudes de documentos se escriben en HDFS (es decir, en el método Close). Por lo tanto, los datos de longitud del documento termina en m diferentes archivos, donde m es el número de mapeadores; estos archivos se consolidan en una representación más compacta.

Finalmente, los parámetros deben configurarse adecuadamente para la compresión de las listas de publicaciones. La mejor práctica prescrita es utilizar la compresión Golomb en las diferencias del número de documento de primer orden (es decir, -gaps d). La dificultad, sin embargo, es que la compresión Golomb requiere dos parámetros: el tamaño de la colección de documentos y el número de publicaciones para una lista publicaciones particular (es decir, df). El primero es fácil de obtener y se puede pasar al reducer como una constante. El df de un término, sin embargo, no se conoce hasta que todos los mensajes han sido procesados-y, por desgracia, el parámetro debe ser

conocida antes que se codifican publicaciones. Una solución de dos pasos que involucra primero el almacenamiento en búfer de las publicaciones (en la memoria) sufriría el cuello de botella de memoria que hemos estado tratando de evitar en primer lugar.

Para solucionar este problema, es necesario informar de alguna manera el reductor de df de un termino antes de cualquiera de sus envíos llegan. La solución es tener los mapeadores emitan claves especiales de la forma de t, * para comunicar frecuencias parciales de documentos. Esto se logra de una manera similar al cálculo de las longitudes de los documentos. El mapeador contiene una matriz asociativa en memoria que realiza un seguimiento de cuántos documentos se ha observado un término (es decir, la frecuencia del documento local del término para el subconjunto de documentos procesados por el mapeador). Una vez que el mapeador ha procesado todos los registros de entrada, claves especiales de la forma t, * se emiten con el df parcial como valor.

Para garantizar que estas teclas especiales llegan primero, se define el orden de clasificación de la tupla para que el símbolo especial * precede a todos los documentos. Por lo tanto, para cada término, el reductor primero encuentra una serie de claves t, * lo que representa dfs parciales provenientes de cada mapeador. Resumiendo todas estas contribuciones parciales rendirá df del término, que luego se puede utilizar para establecer el parámetro de compresión de Golomb. Esto permite que las publicaciones se codifican en una sola pasada.

Fusionar resultados en todas las particiones

El broker en una arquitectura distribuida de documentos particionada es responsable de fusionar los resultados de cada uno de los servidores de partición. Se exploraron dos algoritmos separados para lograr esto.

El primer enfoque, que llamaron estrategia de fusión independiente, es para ver los resultados de la fusión como un problema de búsqueda federada, tratando a cada partición como una colección independiente.

Este enfoque simplifica la construcción del índice, pero hace que la calificación de los documentos en las particiones sea difícil de comparar directamente. Para corregir esto, los puntajes brutos se normalizan, por partición, usando la transformación z-score de la siguiente manera:

$$S^* = (S - \mu o) / \sigma$$

donde S es la puntuación bruta, μo es la media muestral de las puntuaciones crudas, σ 2 es la varianza de la muestra, y S * es la puntuación normalizada. Los puntajes

normalizados ahora se consideran muestras de una distribución normal estándar. El intermediario devuelve una lista ordenada combinando todos los documentos devueltos de todas las particiones en función de sus puntajes normalizados.

La otra estrategia para la fusión de resultados se llama estadísticas globales, que incluye la distribución de las estadísticas globales de recogida a cada uno de los índices de partición. Primero, cada uno de los índices de partición se construye de forma independiente. A continuación, un trabajo MapReduce Maps a través de todos los índices de partición para calcular las estadísticas globales (el df global y cf para cada término y el tamaño de toda la colección). Finalmente, las estadísticas globales se propagan nuevamente a cada índice de partición. Esto también se logra con MapReduce: correlacionamos sobre cada lista de publicaciones, y dentro de cada mapeador las estadísticas globales se cargan en la memoria. Se escribe una nueva versión del índice con las estadísticas actualizadas (no se requieren reductores). Este proceso simple se repite para cada partición. Dado que MapReduce puede aprovechar el rendimiento agregado del disco de varias máquinas, estos trabajos de MapReduce son sorprendentemente rápidos.

La ventaja del enfoque estadísticas global es que las puntuaciones de documentos generados en cada partición son exactamente las mismas que las puntuaciones de documentos en un solo índice global que abarca todas las particiones, al menos por los modelos de recuperación utilizado en sus experimentos (BM25 y consulta de verosimilitud). Por lo tanto, no es necesario manipular el puntaje adicional, y el broker simplemente recurre a los resultados de los servidores de partición y devuelve la lista final rerankeada al cliente.

Prototipo

Para el prototipo se trabajó sobre el sistema operativo Ubuntu 16.04.

Se realizó la instalación de Hadoop e Ivory, ambas de código abierto.

http://ivory.readthedocs.io/en/latest/install/

Para poder ejecutar Ivory en un clúster Hadoop real se uso las imágenes de máquinas virtuales de Cloudera, que vienen con el clúster de un solo nodo preconfigurado. Las imágenes se pueden descargar aquí:

https://www.cloudera.com/downloads/quickstart_vms/5-12.html

Se utilizara la imagen de VirtualBox, ya que VirtualBox está disponible gratuitamente en todas las plataformas principales.

https://www.virtualbox.org/wiki/Downloads

La colección documentos que se utilizó finalmente fue la siguiente:

https://github.com/lintool/Ivory/blob/master/data/cacm/cacm-collection.xml.gz

Dicha colección viene en formato estándar TREC:

<DOC>

<DOCNO>CACM-0001</DOCNO>

...

</DOC>

<DOCNO>CACM-0002</DOCNO>

...

</DOC>

...

Las etiquetas <DOC> y </ DOC> indican documentos. Las etiquetas <DOCNO> y </DOCNO> rodean el identificador de documento único.

Esta es la colección del CACM, que contiene resúmenes de las Comunicaciones de la ACM de finales de los años 1950 a mediados de los años sesenta. El prototipo, en esencia, es lograr instalar y crear un ambiente de recuperación de información basado en Hadoop, por medio de la herramienta Ivory.

Considerando que la colección utilizada fue demasiado pequeña, y que la misma puede ser que no sea útil para los requerimientos del curso se podría optar por otras colecciones. Las mismas se podrían obtener desde

https://github.com/lintool/Ivory/tree/master/data

También se investigó el uso de los dataSets ClueWeb09 fue creado para apoyar la investigación sobre la recuperación de la información y las tecnologías relacionadas con el lenguaje humano. Se compone de alrededor de 1.000 millones de páginas web en diez idiomas que se recogieron en enero y febrero de 2009

https://lemurproject.org/clueweb09/. pero no tuve éxito para acceder a los mismos.

Conclusiones

Como conclusiones me pareció interesante poder instalar un sistema de recuperación de información de documentos, aplicando los conceptos de Hadoop, y sus respectivos HDFS y MapReduce, el cual era un paradigma nuevo para mis conocimientos previos a cursar está materia. Pude ver la implementación de un índice invertido y poder crear un sistema de recuperación con hardware de uso doméstico y usando herramientos de código abierto. Como trabajo a futuro se debería seguir investigando el uso de este nuevo paradigma para la implementación de herramientas útiles en la recuperación de información.

Referencias bibliográficas

https://es.wikipedia.org/wiki/Big_data
https://es.wikipedia.org/wiki/Apache_Hadoop
https://www.youtube.com/watch?v=LtCbCfr3W3Y
https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
https://www.tutorialspoint.com/es/hadoop/index.htm
http://blogs.solidq.com/es/big-data/que-es-mapreduce
http://www.ticout.com/blog/2013/04/02/introduccion-a-hadoop-y-su-ecosistema/
https://eva.fing.edu.uy/course/view.php?id=947
Libro Next Generation Databases - Guy Harrison - [ISBN 978-1-4842-1329-2]
Libro Hadoop: The Definitive Guide - Tom White - [ISBN: 978-1-491-90163-2]
https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm
https://cs.uwaterloo.ca/~jimmylin/publications/Lin_etal_TREC2009.pdf