PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

PROCESAMIENTO DE GRANDES VOLÚMENES DE DATOS UTILIZANDO MAPREDUCE

GUSTAVO ADOLFO RIVEROS GONZÁLEZ

INFORME FINAL DEL PROYECTO
PARA OPTAR AL TÍTULO PROFESIONAL DE
INGENIERO DE EJECUCIÓN EN INFORMÁTICA

ABRIL, 2013

Pontificia Universidad Católica de Valparaíso
Facultad de Ingeniería
Escuela de Ingeniería Informática

PROCESAMIENTO DE GRANDES VOLÚMENES DE DATOS UTILIZANDO MAPREDUCE

GUSTAVO ADOLFO RIVEROS GONZÁLEZ

Profesor Guía: **Dr. Wenceslao Palma Muñoz**
Profesor Co-referente: **Dr. Rodolfo Villaroel Acevedo**

Carrera: **Ingeniería de Ejecución en Informática**

ABRIL, 2013

Agradezco a mis padres, que sin su apoyo incondicional a lo largo de toda mi vida no sería posible la presentación de este tabajo. Te agradezco Astrid, por darme la fuerza de seguir este camino cuando la confusión atormentaba mis pensamientos y el futuro era incierto, sé que estarás siempre a mi lado. Finalmente, agradezco a Dios, compañero fiel que ha estado conmigo cuando más lo he necesitado.

Resumen

MapReduce es un modelo de programación para el procesamiento de grandes volúmenes de datos, el cual fue introducido por Google a mediados del año 2004 en base a la necesidad de realizar consultas sobre una enorme cantidad de datos que Google ya manejaba en ese entonces. Uno de los factores que carateriza al modelo MapReduce es la tolerancia a fallas y el poder realizar cálculos paralelos en un cluster de computadoras. Su aceptación y utilización se ha incrementado a lo largo del tiempo, llegando a ser utilizado por grandes Empresas como Facebook y Amazon, entre otras. Todo el éxito que ha alcanzado el modelo ha sido en gran parte gracias a cualidades como la abstracción sobre la forma en que maneja las fallas, y el procesamiento en paralelo. De éste modo resulta de fácil utilización por parte de programadores, debido a su baja complejidad y rápida puesta en marcha.

Palabras Clave: Map,Reduce, Paralelismo, Datos.

Abstract

MapReduce is a programming model for processing large volumes of data, which was introduced by Google in mid-2004 based on the need to conduct consultations on an enormous amount of data that Google already handled at the time. One factor that characterizes the MapReduce model is fault tolerance and the power of parallel calculation hundreds or thousands of machines. His acceptance and use has increased over time, becoming used by big companies like Facebook and Amazon, among others. All the success he has achieved the model has been largely thanks to the qualities of abstraction over the way it handles failures, and parallel processing. This makes it easy to use by developers due to its low complexity and fast set-up.

Keywords: Map, Reduce, Parallelism, Data.

Índice

Lista de Figuras

Lista de Tablas

1. Introducción

Mapreduce es un modelo de programación, concebido por Google, para el procesamiento de grandes volúmenes de datos en clusters de computadores. Inicialmente utilizado para facilitar la tarea de construcción del índice invertido de Google.com, MapReduce ha sido utilizado para tareas tales como el procesamiento de grafos de gran tamaño, procesamiento de texto, bioinformática y análisis de logs entre otros. Además, compañías tales como Facebook, Yahoo! y Twitter utilizan MapReduce en sus tareas de procesamiento de datos mostrando que es posible procesar petabytes en un par de horas. Las fortalezas de MapReduce se encuentran en aspectos ocultos a los desarrolladores de aplicaciones tales como el soporte a la tolerancia a fallas y el balance de carga.

La investigación y análisis de MapReduce presente en este informe se basa en el Modelo MapReduce propuesto en Hadoop, que corresponde a una plataforma opensource de procesamientos de datos, desarrollada en su mayoría por Apache y Yahoo. El modelo introducido por Apache se basa en el modelo de Google y, según sus desarrolladores, posee mínimas diferencias técnicas. Cifras o datos técnicos sobre el modelo MapReduce original son relativamente difíciles de obtener, debido a la restrictiva licencia que posee el mismo.

En el presente trabajo se desarrolla una completa investigación e implementación del algoritmo de Join en MapReduce. Existen variadas formas de implementar este algoritmo utilizando MapReduce, de las cuales son dos las seleccionadas y las que se llevan a cabo. Estas implementaciones se realizan sobre tres plataformas o medios diferentes, el primero de ellos corresponde a Hadoop, los siguientes se desarrollan sobre Pig y MongoDB, cabe destacar que cada uno de estos entornos es explicado en profundidad mas adelante.

Finalmente luego de realizar cada una de estas implementaciones es posible establecer comparaciones de desempeño sobre cada plataforma o entorno, y sobre las dos ténicas que se utilizaron en cada implementación del algoritmo Join. Todos los resultados obtenidos son analizados en profundidad en la sección final del presente informe.

2. Objetivos

2.1. Objetivo General

Analizar el funcionamiento del modelo MapReduce a través de implementaciones de estrategias de Join.

2.2. Objetivos Específicos

- Comprender la problemática del procesamiento de grandes volúmenes de datos.
- Comprender el modelo MapReduce.
- Implementar estrategias de Join en plataformas MapReduce.
- Comparar y Analizar resultados obtenidos de las diferentes implementaciones.

3. El Modelo MapReduce

MapReduce es un modelo de programación para el procesamiento de grandes volúmenes de datos, el cual puede encontrarse implementado dentro de un programa usuario, el cual a su vez debe cumplir una tarea específica dentro de un sistema. Esta metodología resulta de fácil manejo por parte del usuario, debido a que la complejidad de realizar cálculos en paralelo por sobre los datos, mantener la comunicación de una máquina con otra dentro del sistema, y el poder sobrellevar las fallas que se pueden producir en cada operación, entre otras, son tareas que se encuentran insertas en el modelo MapReduce y por lo tanto no deben ser manejadas por el usuario, por lo menos no en mayor profundidad. Estos aspectos facilitan enormemente su utilización y aceptación.

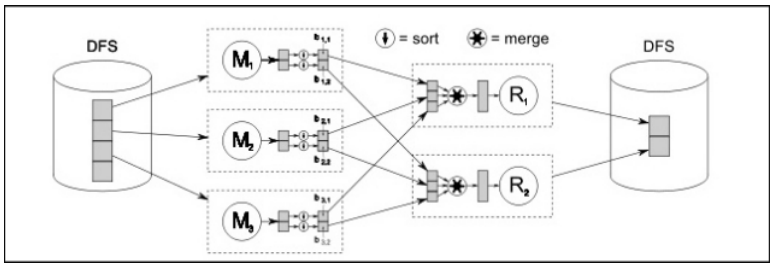

Figura 1: Esquema Modelo MapReduce.[1]

[1]Spyros Blanas et al. A Comparison of Join Algorithms for Log Processing in MapReduce . volume 1, página 2. ACM Sigmod, June 2010.

Es así como grandes compañías alrededor del mundo han visto en MapReduce una posibilidad para afrontar la complejidad de procesar enormes volúmenes de datos, entre estas empresas destacan Facebook, IBM y Google entre otras, en donde las dos primeras utilizan Hadoop, que corresponde a una plataforma de procesamiento de datos libre que incorpora el modelo MapReduce dentro de sus análisis. Una de las potencialidades por las cuales las grandes compañías utilizan este modelo, es el procesamiento de Logs [2], que resulta altamente importante al momento de generar tendencias sobre ciertas aplicaciones o funcionalidades dentro de un sistema, un ejemplo es el número de veces que se realiza un click sobre cierta funcionalidad dentro de Facebook o que eventos estan disminuyendo en el tiempo, podría ser información

[2]Un Log corresponde a un registro de una actividad dentro de un sistema, que generalmente se guarda en un fichero de texto, al que se le van añadiendo líneas a medida que se realizan esas acciones sobre el sistema

valiosísima al momento de decidir si mantener o no dicha funcionalidad. De ésta forma MapReduce entrega información que incide directamente en deciciones comerciales dentro de la grandes compañias.

La forma en que el modelo MapReduce trabaja internamente es a través de dos fases o etapas centrales, que como su nombre lo dice son Map y Reduce, a continuación en la figura 1 se ilustra el trabajo de las dos etapas.

Como la figura 1 muestra, la última fase del modelo MapReduce es la etapa Reduce, la cual recibe sus datos de entrada de la función Map. Las etapas intermedias Sort y Merge corresponden a funciones insertas en la plataforma Hadoop que serán explicadas mas adelante. Los datos que se analizarán se almacenan en archivos de sistema (DFS) que en el caso de Hadoop corresponden a HDFS, en la figura 2 es posible visualizar los datos de entrada que posiblemente podrían ser analizados por el modelo MapReduce.

IP	NAVEGADOR	DEPARTAMENTO
192.168.1.1	IE	d_1
192.168.1.2	IE	d_2
192.168.1.3	Chrome	d_3
192.168.1.4	Chrome	d_3
192.168.1.5	IE	d_1
192.168.1.6	IE	d_2
192.168.1.7	Firefox	d_3
192.168.1.8	Chrome	d_4

Figura 2: Tabla de Registro de Navegadores utilizados dentro de una Empresa

La tabla anterior correspondería a un extracto de un registro que almacena los distintos navegadores utilizados por los empleados dentro de una Empresa, muchas compañías establecen navegadores por defecto dentro de sus equipos luego de llegar a acuerdos económicos con estos, por lo tanto es importante mantener un control sobre el uso que se le da a estos equipos y que cada estación utilice las configuraciones que la Empresa estableció inicialmente.

A continuación se desarrollará el trabajo que realiza el modelo MapReduce sobre los datos contenidos en la figura 2 con el fin de establecer cuales son los departamentos que incumplen el utilizar el navegador Internet Ex-

plorer (provisto por la empresa), con el fin de tomar medidas y corregir esta situación.

3.1. Fase Map

La entrada a la Fase Map corresponde a una tupla (clave, valor) que es analizada por la función Map definida, es importante destacar que el entorno Hadoop particiona automáticamente todos los datos de entrada en diferentes archivos, en donde cada uno de estos segmentos es analizado por una función Map paralela, como puede apreciarse en la figura 1. Luego de este análisis la salida de la fase Map corresponde a una tupla (clave, valor), que al igual que en un principio es un valor asociado a una clave, la diferencia radica en los valores tanto de la clave como del valor asociado a ella, por lo general la clave de salida es la misma que la clave de entrada ya que corresponde al identificador del análisis [5].

```
Map(clave:null,valor:registro_completo)
{

clave = registro_completo.getDepartamento();
Navegador = registro_completo.getNavegador();

If (Navegador.getString()!="IE"){

emit(clave,1);
}
emit(clave,0);
}}
```

Figura 3: Función Map que analiza el navegador utilizado.

En la figura 3 es posible visualizar el análisis que se realizará sobre los datos contenidos en la figura 2, los datos de entrada corresponden a una clave inicial que se encuentra vacía y a toda una fila de los datos de entrada. En el interior de la función Map se extrae la clave y el valor asociado a ella, que en este caso en particular corresponden al código del departamento y al navegador que se utiliza, el análisis es simple, solo se necesita saber si el departamento cumple con utilizar Internet Explorer, a través de una sentencia condicional se emite el resultado final de la fase Map, cuya clave sigue siendo el identificador del departamento y un valor dado por la misma sentencia que corresponde a un 1 en el caso que no se utilice IE o un 0 en el caso que se utilice.

La función Map continúa iterando sobre cada una de las líneas del archivo de entrada, es decir, sobre cada ip y sus valores asociados.

3.2. Sort y Merge

Como se mencionó anteriormente, estas funciones son provistas por el entorno Hadoop, ambas resultan de vital importancia en el trabajo completo del modelo MapReduce, según la figura 1 la etapa Sort realiza un ordenamiento de las salidas de cada Fase Map según la clave de la tupla (clave, valor), según el ejemplo planteado se ordenan las tuplas de acuerdo a su código de departamento.

Luego en la etapa Merge, cada uno de estos grupos se une hacia los otros grupos pertenicientes a otras salidas de otras funciones Map que poseen los mismos códigos de departamentos, este ordenamiento y agrupación puede verse reflejado en la sección central de la figura 1. Una vez que todos los grupos de tuplas (clave, valor) han sido creados, comienza la etapa Reduce.

3.3. Fase Reduce

Luego que la función Map ha establecido los datos intermedios y la fase Sort y Merge de Hadoop han terminado, es la función Reduce quien recibe todos estos resultados, en donde los valores se encuentran agrupados según su clave en tuplas (clave departamento, lista valores), donde ésta lista corresponde a un enumeración de cada uno de los valores emitidos por las funciones map anteriores. La función Reduce como puede apreciarse en la figura 4, suma cada uno de estos valores en la lista, cifras que corresponden a 0 y 1, cero en el caso que se utilice IE y uno en el caso que no se utilice, finalmente se obtiene una cifra final por cada departamento que indica el número de ocurrencias de esta falta. La etapa Reduce retorna estos resultados de la misma forma en que Map recibe los datos, a través de archivos HDFS particionados con un tamaño automático establecido por Hadoop, según se aprecia en la figura 1.

```
Reduce(clave, lista_valores) {
int suma_navegadores=0;

while(lista_valores){

suma_navegadores = suma_navegadores + lista_valores.getValor();

lista_valores.next();
}

emit(clave,suma_navegadores);
}
```

Figura 4: Función Reduce que emite el número de veces que un departamento utiliza un navegador no autorizado.

4. Hadoop

Hadoop es un software de código abierto que implementa el modelo MapReduce y permite el procesamiento distribuido en paralelo de grandes cantidades de datos a través de servidores, estos procesos distribuidos son posibles gracias al sistema de archivos que implementa hadoop, que es conocido como Hadoop Distributed Fyle System (HDFS). MapReduce es implementado en éste software y utilizado para realizar tareas sobre grandes volúmenes de datos.

4.1. Sistema de Archivos Distribuidos de Hadoop, HDFS.

Los HDFS están diseñados para almacenar grandes cantidades de datos de forma confiable, y para transmitir éstos datos a través de un gran ancho de banda a las aplicaciones del usuario. HDFS almacena de forma separada los metadatos y los datos de las aplicaciones, los metadatos los almacena en servidores dedicados llamados NameNodes y los datos de las aplicaciones los almacena en servidores llamados DataNodes, todos los servidores antes mencionados se encuentran completamente comunicados, mediante protocolo TCP. Para mantener la confiabilidad del sistema, un archivo es replicado en múltiples DataNodes, otorgando al sistema la durabilidad de los datos, además de multiplicar la velocidad de transferencia, ya que existen más oportunidades de que al momento de estar realizando un cálculo se puedan tener los datos necesarios más cerca.

Es importante destacar que todos los archivos contenidos en este sistema (HDFS) se encuentran divididos en bloques, que por lo general se encuentran entre los 64 MB y 128 MB de tamaño. esto representa un gran beneficio al momento de almacenar archivos que tal vez son tan extensos que no podrían ser almacenados en un solo disco local.

4.2. Arquitectura del Sistema de Archivos Distribuido de Hadoop

Un Cluster HDFS posee dos clases de nodos en funcionamiento, namenodes (Master) y datanodes (Workers).

4.2.1. NameNodes

El namenode, anteriormente mencionado como el "Master"del sistema, mantiene el árbol de archivos de sistema y los metadatos de todos los archivos y directorios del árbol, el cual almacena registros como: permisos, tiempos

de acceso y modificación, además de cuotas de uso de espacio en el disco, toda ésta información es almacenada en un disco local. El namenode conoce específicamente en que datanodes se encuentran los bloques de memoria de cierto archivo requerido por el sistema, sin embargo ésta información no la maneja de forma persistente dado a que es reconstruida desde los mismo datanodes en cada inicio del sistema. Debido a todas éstas importantes características o funciones que posee el namenode se hace imposible el funcionamiento del sistema solo con datanodes, ya que no existiría registro alguno de como reconstruir los archivos desde los bloques en los que fueron inicialmente divididos. [10]

4.2.2. DataNodes

Los datanodes, que anteriormente se mencionaron como "Workers"dentro del cluster, se encargan de recibir y enviar bloques de archivos, en el momento en que el cliente se lo indique, o en su defecto el namenode. Todos estos movimientos o cambios son reportados periódicamente al namenode a través de listas en las que se indica que bloques específicamente están almacenando.

El namenode resulta fundamental para el funcionamiento del sistema, es por esto que debe poseer una alta tolerancia a los fallos, y Hadoop se encarga de esto a través de dos métodos:

- Respaldar: Hadoop puede ser configurado de modo que el namenode registre de forma constante su estado en un disco local o en dispositivos NFS, de ésta forma se mantienen los archivos que permiten la persistencia de los metadatos del sistema.

- NameNode Secundario: Otra opción es el ejecutar un namenode secundario, el cual, en estricto rigor, no se comporta como un namenode, este se ejecuta en una máquina independiente del sistema debido a la gran cantidad de recursos que consume. En el caso que se produzca una falla del namenode principal, se copian los metadatos que fueron sincronizados hacia un disco local o dispositivo NFS y son traspasados al namenode secundario, convirtiéndose así en el namenode principal.

4.3. Flujo de Datos en HDFS

A continuación se presentan los pasos o etapas que existen en la interacción del cliente con el Sistema de Archivos Distribuidos de Hadoop, con una introducción sobre la topología de la red de nodos en Hadoop.

4.3.1. Topología de una red en Hadoop

En el contexto del procesamiento de grandes volúmenes de datos, uno de los factores limitantes es la velocidad con la que se pueden transmitir datos entre nodos, ya que el ancho de banda es un bien escaso dentro de la red. [10]

La idea es utilizar el ancho de banda entre dos nodos como una medida de distancia, pero ésto resulta en la práctica altamente complejo, por tanto Hadoop realiza una aproximación, en donde la red está representada como un árbol y la distancia entre dos nodos es la suma de sus distancias hacia sus nodos inmediatamente anteriores que poseen en común. Éste árbol no posee niveles predefinidos, pero es normal tener niveles que corresponden al data center, al rack y al nodo en el que se encuentra ejecutándo un proceso. La idea es que el ancho de banda disponible para cada uno de los posibles escenarios sea cada vez menor: procesos en el mismo nodo, nodos diferentes en el mismo rack, nodos en racks diferentes en el mismo data center, nodos en diferentes data centers.

En la figura 5 se ilustra como Hadoop ve las distancias entre los nodos, a modo de ejemplo se han establecido ciertas variables y reglas para dimensionar las distancias en los 4 escenarios presentados anteriormente, las variables utilizadas corresponden a n_x(nodo x), r_y(rack y) y d_z (data center z), las reglas son las siguientes:

- distancia($/d_1/r_1/n_1$, $/d_1/r_1/n_1$) = 0 (procesos en el mismo nodo)
- distancia($/d_1/r_1/n_1$, $/d_1/r_1/n_2$) = 2 (nodos diferentes en el mismo rack)
- distancia($/d_1/r_1/n_1$, $/d_1/r_2/n_3$) = 4 (nodos en racks diferentes en el mismo data center)
- distancia($/d_1/r_1/n_1$, $/d_2/r_3/n_4$) = 6 (nodos en diferentes data centers)

Hadoop por defecto asume en un principio que la red es plana (todos los nodos están en el mismo rack en el mismo data center) ya que no es capaz de saber por sí mismo cual es la red del sistema en el cual se encuentra inserto, es por esto que es necesario que el usuario configure la red del sistema en Hadoop.

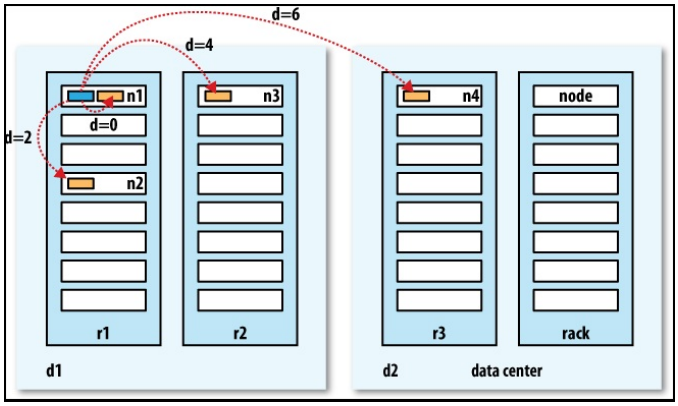

Figura 5: Distancias de red en Hadoop[3]

[3]Tom White. Hadoop, The Definitive Guide, capítulo 3, página 65. 2 edition, oct 2010

4.3.2. Lectura de Archivos

A continuación se desarrollarán cada una de las etapas que existen en la lectura de archivos desde el cliente hacia el HDFS, todas las etapas que se explican pueden ser visualizadas en la figura 6: [10]

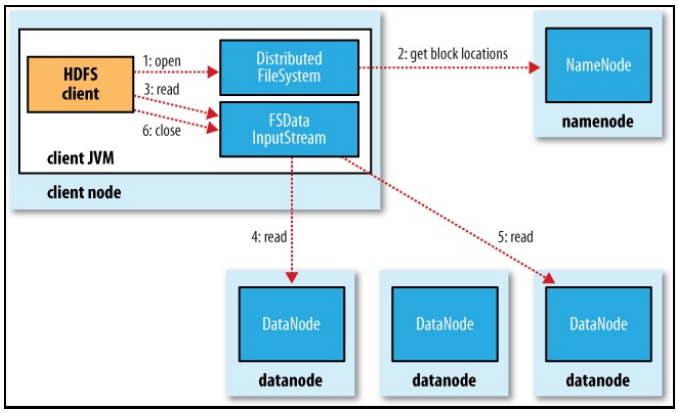

Figura 6: Lectura de Archivos en HDFS.[4]

[4]Tom White. Hadoop, The Definitive Guide, capítulo 3, página 63. 2 edition, oct 2010

- Paso 1: El Cliente abre el archivo que desea leer, invocando al método open() en el Objeto Sistema de Archivos, que resulta ser una instancia del Sistema de Archivos Distribuido.

- Paso 2: El sistema de Archivos Distribuidos llama al namenode, usando llamado a procedimiento remoto, con el fin de determinar las ubicaciones exactas de los primeros bloques del archivo requerido por el cliente. Por cada bloque el namenode retorna la ubicación de los datanodes que poseen una copia, en el caso que el cliente en sí sea un datanode, realizará la lectura sobre si mismo y confirmará si ha almacenado una copia del bloque. Luego el Sistema de Archivos Distribuido retorna un Archivo que soporta búsquedas y recibe los Flujos de Datos de Entrada (FSDIS), de este modo el cliente lee los datos desde éste archivo. FSDIS contiene también un archivo que maneja las entradas y salidas, tanto del namenode como de los datanodes, el cual recibe el nombre de Archivo de Búsqueda de Datos sobre los Flujos de Entrada (DFSIS)

- Paso 3: El cliente realiza el método read() sobre el flujo DFSIS que había almacenado las direcciones de los datanodes que poseían los primeros bloques del archivo.

- Paso 4: Luego se conecta al datanode más cercano en búsqueda de un bloque del archivo. Los datos son transmitidos desde el datanode hacia el cliente, el que a su vez realiza constantes métodos de lectura, en caso que el DFSIS encuentre un error en uno de los datanodes, buscará el bloque en el datanode siguiente y además recordará que ese datanode posee un error.

- Paso 5: Cuando se ha alcanzó el final del bloque se cierra la conexión entre el DFSIS y el datanode seleccionado, luego se vuelve a buscar el mejor datanode para el siguiente bloque de archivos.

- Paso 6: Los bloques son leídos según el DFSIS va abriendo conexiones hacia los datanodes, a medida que el cliente lee los mismos bloques a través del flujo, llama al namenode para que reciba las ubicaciones de los datanodes para el siguiente lote de bloques. Una vez que el cliente ha terminado de leer, invoca al método close() en el FSDIS.

Todo éste proceso se realiza a ojos del cliente, de forma continua sin interrupciones.

4.3.3. Escritura de Archivos

A continuación se detallarán y explicarán las etapas que existen en la creación de un archivo, escribir datos en él y luego cerrarlo, como se indica

en la figura 7. [10]

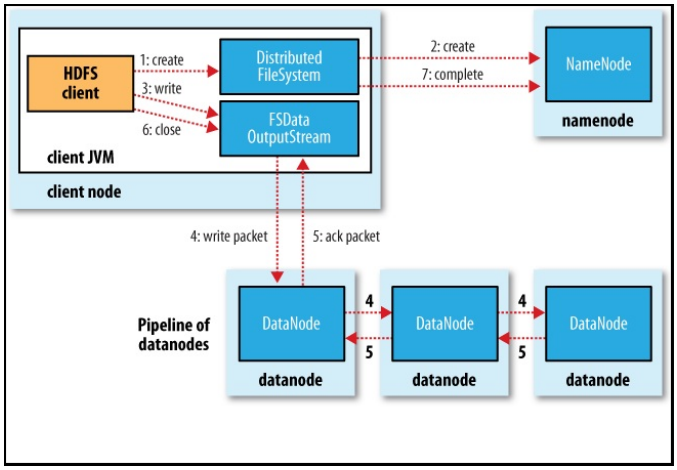

Figura 7: Escritura de Archivos en HDFS.[5]

[5]Tom White. Hadoop, The Definitive Guide, capítulo 3, página 66. 2 edition, oct 2010

- Paso 1: El cliente crea el archivo invocando al método create() en el Sistema de Archivos Distribuido (DFS).
- Paso 2: El sistema de Archivos Distribuido realiza una llamado a procedimiento remoto hacia el datanode para crear un nuevo archivo en el espacio de nombres del sistema de archivos, sin bloques asociados al nuevo archivo.
- Paso 3: El namenode realiza varias comprobaciones para lograr establecer que el archivo no existe previamente en el sistema, y además comprueba que el cliente posea los permisos necesarios para crear el archivo. Si las comprobaciones anteriores resultan positivas el namenode crea un registro del nuevo archivo, de lo contrario la creación del archivo falla y se arroja una excepción hacia el cliente. En el caso satisfactorio el Sistema de Archivos Distribuido retorna un FS de Flujo de Datos de Salida (FSDOS) para que el cliente comience a escribir los datos, así como en el caso de la lectura, el FSDOS posee un Archivo de Búsqueda de Datos sobre los Flujos de Salida (DFSOS).
- Paso 4: Mientras el cliente escribe los datos en el nuevo archivo, el DFSOS divide al archivo en paquetes que se almacenan un una cola interna, llamada Cola de Datos. La Cola de Datos es utilizada por

el Flujo de Datos, cuya responsabilidad es el consultarle al namenode donde ubicar los nuevos bloques, seleccionando entre una lista de bloques adecuados en donde se puedan almacenar estas réplicas. Ésta lista de datanodes conforman una serie, en donde en el caso que el nivel de replicación sea 3, se tendrán 3 nodos en la secuencia. El Flujo de Datos lleva los paquetes al primer datanode de la secuencia, el cual almacena el paquete y lo reenvía al segundo datanode de la secuencia, igualmente el segundo datanode almacena el paquete y lo reenvía al tercero y último datanode (en éste caso) de la secuencia.

- Paso 5: El DFSOS también mantiene una cola interna de paquetes que están en espera de ser reconocidos por los datanodes, ésta cola se denomina la cola ack. Un paquete es removido de la cola ack sólo cuando ha sido reconocido por todos los datanodes de la secuencia.

- Paso 6: Si se produce una falla en un datanode mientras se escriben datos en él, se llevan a cabo las siguientes acciones: primero se cierra la secuencia, y ningún paquete en la cola ack puede ser agregado al frente de la cola de datos, de éste modo los nodos que están más abajo que el datanode con fallas dentro del flujo, no perderán ningún paquete. El bloque actual en los datanodes sin fallas recibe una nueva identidad, la cual se le comunica al namenode, de modo que el bloque del datanode fallido será borrado si el datanode se recupera luego de la falla. Posteriormente el datanode con fallas es removido de la secuencia, y un registro de los datos del bloque es escrito en los otros dos datanodes restantes dentro de la secuencia. El namenode recibe la noticia que el bloque se encuentra bajo el proceso de replicación y que estará un una réplica que será creada en otro nodo. Cuando el cliente ha terminado de escribir los datos se invoca al método close().

- Paso 7: Al momento de cerrar el proceso de escritura, se eliminan todos los paquetes en secuencia hacia el datanode y se espera por el reconocimiento (ack) antes de contactar al namenode para indicarle que el archivo que se estaba escribiendo está completo.

4.4. Flujo de Datos en el Modelo MapReduce

A continuación se desarrollarán las etapas comprometidas en la ejecución de una tarea MapReduce.

4.4.1. Etapas de Ejecución

En un comienzo el modelo MapReduce que se está utilizando en el Programa Usuario, divide el archivo de entrada en M piezas, que por lo general son de 16 a 64 Megabytes (el tamaño de estas piezas puede ser establecida por el usuario), este número M también corresponde al número de funciones Map distribuidas en las diferentes máquinas como se indica en la figura 8 en el paso número 1, siempre se recomienda que el número de máquinas dentro del cluster sea menor al número de funciones Map necesarias, de éste modo se mantendrá un mínimo número de máquinas inactivas, que a su vez representan pérdidas económicas.

Luego de dividir el archivo de entrada, se ejecutarán varias copias del programa usuario en cada una de las máquinas con las que cuenta el cluster. Dentro de todas las copias del programa existe una especial denominada "Master", el resto de las máquinas corresponde a "Workers", que solo reciben tareas asignadas por el Master. A cada Worker que se encuentre inactivo le es asignada una tarea Map o una tarea Reduce según su estado. Dentro del Master se mantiene cierta estructura de datos, por ejemplo por cada tarea Map y cada tarea Reduce mantiene un estado de las mismas: inactiva, en progreso o completa, además de la identidad de cada Worker; en el caso que se traten de tareas no-inactivas. El Worker que recibe una tarea Map

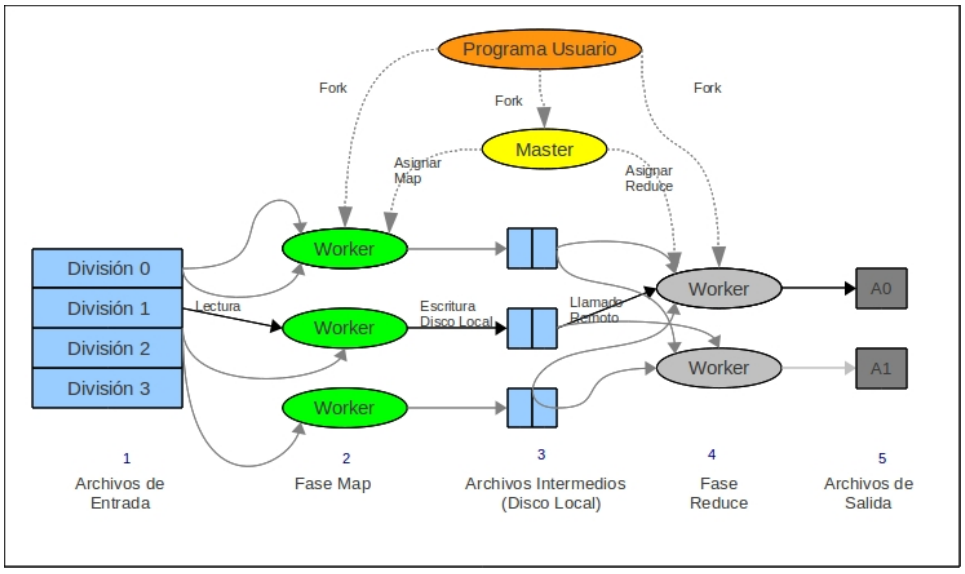

Figura 8: Ejecución de MapReduce.

se encarga primero de leer el contenido de la pieza del archivo de entrada que le fue asignada, después de leer los datos, extrae tuplas (clave, valor) y las traspasa a la función Map que el usuario estableció, una vez obtenidos los datos, la función Map cumple su cometido y genera pares intermedios (key,value) que cumplen ciertas características, según la lógica de la misma función. Finalmente estos datos intermedios son almacenados en un buffer.

Cada cierto tiempo los pares intermedios almacenados en el buffer son escritos en el disco local del Worker, el que se encuentra particionado a través de la función de partición (establecida por el usuario) en R regiones , la ubicación de cada uno de éstos pares dentro del disco local es enviada al Master, el que se encargará de transmitir estas locaciones a cada Worker Reduce. Cuando un Worker Reduce es notificado sobre las locaciones de los pares intermedios en el disco local, utiliza llamadas a procedimientos remotos para poder leer estos datos obtenidos desde la función Map anterior (ver figura 8 , paso 3). Luego que el Worker Reduce ha concluido la lectura de todos los datos intermedios, éstos son ordenados según sus llaves intermedias correspondientes, como se explicó anteriormente, de modo que agrupa las ocurrencias con la misma llave intermedia [4]. Existen oportunidades en que los datos intermedios son demasiados y el Worker Reduce debe almacenarlos en una memoria externa. Después de agruparlos, el Worker Reduce itera sobre los datos intermedios y cada vez que encuentra una clave (única), traspasa ésta y todos los datos asociados a la misma, hacia la función Reduce establecida por el usuario.

Luego que el par (clave, valor) ha sido traspasado a la función Reduce, ésta agrega su resultado al archivo final de salida de ésta misma partición.

Al concluir todas las tareas Reduce y todas las tareas Map, el Master activa al Programa Usuario, en ésta parte es cuando el modelo MapReduce cesa su trabajo y la atención se vuelca netamente al código de usuario.

4.4.2. Tolerancia a Fallas

Como MapReduce procesa una gran cantidad de datos, por lo general utiliza cientos de máquinas y en cada una de ellas está latente la posibilidad de que se produzca una falla, por lo tanto MapReduce debe saber manejar y solucionar estas situaciones.

- Falla de un Worker

  Durante la constante ejecución de MapReduce, la máquina denominada Master está periódicamente enviando un ping a cada Worker, si en una determinada cantidad de tiempo no se recibe una respuesta del

Worker, se le estima como defectuoso, por lo tanto cualquier tarea Map completada por éste Worker es considerada también como defectuosa y es reiniciada a su estado original, de éste modo puede volver a ser asignada a cualquier otro Worker que se encuentre disponible.

Una situación que caracteriza a una falla que afecta a un Worker Map, es el que la tarea debe ser necesariamente re-ejecutada, debido a que su salida está almacenada en un disco local de la misma máquina fallida, que resulta inaccesible en estos casos. Al contrario de las tareas Map las tareas Reduce no son re-ejecutadas debido a que su salida se almacena en un archivo global del sistema, es un resultado completamente independiente de la falla de un Worker. En cualquiera de los casos anteriores cuando una tarea fallida A ejecutada por un Worker A es re-ejecutada por un Worker B, debe informarse a todo el sistema sobre esta situación de modo que el Worker que desee leer la salida del Worker A sobre la tarea A sabrá que ahora debe realizar la misma lectura pero sobre el Worker B.

- Falla del Master

  En caso de una falla del Master, es necesario establecer puntos o estados de falla del mismo, para lograr ésto es necesario configurar el Master de modo que emita "*checkpoints*" o puntos de control de forma periódica, de éste modo en caso de falla del mismo, es posible re-ejecutar una copia. Éste método resulta altamente efectivo, solo existe el problema en que el Master es único, así mismo es su falla, la cual imposibilita el continuar con la ejecución de MapReduce. La única opción resultante es que el cliente verifique ésta situación y re-ejecute la operación MapReduce desde el último "*checkpoint*" del Master. [4]

4.4.3. Tareas de Respaldo

Una de las causas comunes que retrasa una operación Map o Reduce son los Rezagados, se denomina de éste modo a una máquina que toma o utiliza un tiempo mayor de lo usual para completar cierta tarea, éste tipo de situación se puede deber a: problemas en el disco local de la máquina, el cual puede disminuir su velocidad de lectura desde los 30 megabytes por segundo hasta 1 megabyte por segundo en caso de daños, otro factor que podría generar un Rezagado es que el sistema de programación central del cluster tenga programadas otras tareas en la máquina, lo que causaría lentitud en la ejecución del código MapReduce, debido a que está en constante competencia por el uso de los recursos del sistema (memoria temporal, disco local, etc.).

Existen formas de poder controlar el efecto de estos Rezagados: en el momento que una operación MapReduce esté a punto de terminar, el Master programa ejecuciones de respaldo sobre las tareas en progreso, luego la tarea es marcada como completa cuando la ejecución primaria o la de respaldo concluyen, éste mecanismo ha disminuido el tiempo de ejecución total de una tarea MapReduce hasta en un 40 % con el sistema de respaldo activado. [4]

4.5. Ejecución de MapReduce en el entorno Hadoop

Al momento de ejecutar una tarea MapReduce en Hadoop, existe una gran cantidad de procesos que resultan invisibles hacia el usuario, a continuación se indicarán cuales son los procesos paso a paso acompañados de la figura 9 que muestra cada una de las etapas involucradas, antes que nada las entidades involucradas dentro de éstos procesos son las siguientes: [10]

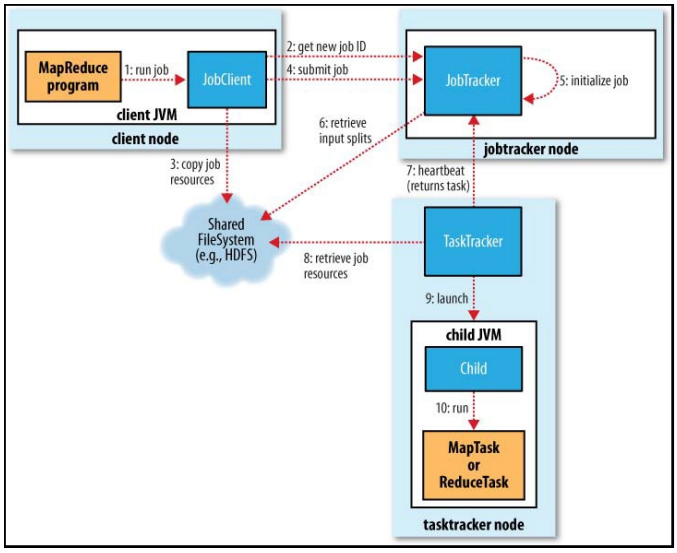

Figura 9: Trabajo MapReduce ejecutado en Hadoop.[6]

[6] Tom White. Hadoop, The Definitive Guide, capítulo 6, página 168. 2 edition, oct 2010.

- El Cliente: Es el encargado de presentar la tarea MapReduce.
- JobTRacker: Se encarga de coordinar el trabajo y corresponde a una aplicación Java cuya clase principal se denomina JobTracker.

- Tasktracker: Se encarga de ejecutar las tareas en las que se divide el trabajo completo, y corresponden a aplicaciones Java, cuya clase principal se denomina TaskTracker.

- El Sistema de Archivos Distribuidos: Por lo general corresponde al sistema de archivos HDFS de Hadoop, y se utiliza para compartir archivos del trabajo entre las otras entidades.

4.5.1. Envío del Trabajo

- Etapa 1: El método runJob() en la clase JobClient() es el método encargado de crear una instancia JobClient() y llamar a submitJob() que se encuentra inserto en él. Después de haber presentado el trabajo, el método runJob() realiza revisiones del progreso una vez por segundo, y lo reporta hacia la consola sólo en el caso que exista un cambio desde el último reporte. Una vez que el trabajo es realizado completamente se despliegan los contadores relacionados con el mismo, esto en el caso exitoso, en el caso que se produzca un error, éste es informado por consola.

- Etapa 2: Se realiza la presentación del trabajo, implementada por la clase JobClient() y su método submitJob(), que realizan las siguientes acciones:

  - sub-etapa 2.1: Consulta al jobtracker por un nuevo id de trabajo (invocando el método getNewJobId() en el JobTracker)
  - sub-etapa 2.2: Comprueba la especificación de la salida del trabajo, por ejemplo: si el directorio de salida del trabajo ya existe con anterioridad o no está especificado, el trabajo no es presentado y se indica un error hacia el programa MapReduce.
  - sub-etapa 2.3: Calcula las divisiones (partes o secciones) de entrada para el trabajo, en el caso que éste número de secciones no pueda ser calculado, el trabajo no es presentado y se indica un error al programa MapReduce.
  - sub-etapa 2.4: Copia los recursos necesarios para ejecutar el trabajo, incluido el archivo Jar del trabajo, el archivo de configuración y el número de secciones de entrada calculadas anteriormente, en el sistema de archivos del jobtracker en un directorio nombrado después de la identificación del trabajo. El archivo Jar es copiado con un factor de replicación alto (por defecto es 10), por lo tanto

existen muchas copias de éste archivo en el cluster, para que los tasktrackers puedan acceder a él en las etapas siguientes.

- sub-etapa 2.5: Le indica al jobtracker que el trabajo está listo para la ejecución (invocando a submitJob en el jobTracker).

- Etapa 3: Corresponde a la sub-etapa 2.4.
- Etapa 4: Corresponde a la sub-etapa 2.5.

4.5.2. Inicialización del Trabajo

- Etapa 5: Cuando el JobTracker() recibe un llamado a su método submitJob(), lo deja en una cola interna desde donde el planificador de trabajo podrá tomarlo e inicializarlo. La inicialización contempla el crear un objeto que represente al trabajo en ejecución, que encapsule sus tareas y acumule información para permitir el rastreo del estado y progreso de las tareas.
- Etapa 6: Para crear la lista de tareas a ejecutar, el planificador de trabajo recupera las divisiones de entrada calculadas por el JobClient(), desde el sistema de archivos compartido. Luego de esto crea una tarea Map por cada división, en el caso del número de tareas Reduce, es un valor entregado por el método setNumReduceTasks hacia el planificador de tareas, el cual se encarga de crear tantas tareas Reduce como el valor entregado por éste método lo indique. En éste punto es cuando se le asignan identificadores a las tareas.

4.5.3. Asignación de Tareas

- Etapa 7: Los tasktrackers ejecutan un ciclo sencillo que periódicamente envía "*pulsos*" al jobtracker. Los pulsos le indican al jobtracker que el tasktracker está activo, y además sirven como un canal para mensajes. Como parte del pulso, el tasktracker indicará si está listo para ejecutar una nueva tarea, y si lo está, el jobtracker le asignará una tarea, el cual se lo comunicará usando un pulso como valor de retorno. Antes de que pueda seleccionar una tarea para un tasktracker, el jobtracker debe elegir un trabajo desde donde seleccionar una tarea, para esto existen muchos algoritmos de planificación, pero por defecto se tiene una lista de prioridades de trabajos.

  Los tasktrackers poseen un número fijo de entradas para tareas Reduce y para tareas Map, pudiendo darse el caso en que se ejecuten dos tareas

Map y dos tareas Reduce en el mismo tasktracker. Por otra parte el planificador de trabajo siempre llenará las entradas Map vacías antes que las entradas Reduce vacías.

4.5.4. Ejecución de una Tarea

- Etapa 8: Una vez que se le ha asignado una tarea al tasktracker, el siguiente paso es que ejecute la tarea. Lo primero que hace es localizar el archivo jar del trabajo, copiándolo desde el sistema de archivos compartidos hacia su sistema de archivos. Luego crea un directorio local de trabajo para la tarea, y extrae los contenidos del archivo jar en éste directorio, finalmente instancia al método taskrunner() para ejecutar la tarea.

- Etapa 9: El taskrunner inicia una máquina virtual de java (JVM).

- Etapa 10: Se corre cada tarea en la máquina virtual de java creada anteriormente. De ésta manera cualquier error en la función Map o la función Reduce no afectará al tasktracker. El proceso hijo se comunica con el padre a través de la conexión central de la interfaz y le informa el progreso de la tarea cada cierto tiempo, hasta que la tarea se completa. Algo importante de destacar es el que la máquina virtual de java puede ser reutilizada por otras tareas.

4.5.5. Manejo de Fallas

En la realidad el código de usuario posee fallas, algunos procesos se caen, y las máquinas pueden fallar. Uno de los beneficios de Hadoop es el poder manejar éstas y otras fallas y poder entregar un resultado satisfactorio sobre el trabajo requerido.

- Falla de Tareas

  Uno de las fallas que se pueden producir es que una tarea hija falle, esto se puede producir cuando el código usuario en una tarea Map o Reduce arroja una excepción sobre la ejecución, cuando esto ocurre el JVM hijo reporta el error de regreso hacia el tasktracker padre, antes de que exista. El error se escribe en los registros de usuario y finalmente el tasktracker marca el intento de realizar la tarea como fallido, liberando una posición del mismo para ejecutar otra tarea. [10]

Otro caso son las tareas "*colgantes*", en donde el tasktracker informa que no ha recibido notificación alguna de progreso durante un tiempo considerable y procede a marcar la tarea como fallida, el proceso JVM hijo es automáticamente eliminado después de éste periodo de tiempo (normalmente corresponde a 10 minutos, pero puede ser configurado). En el caso que se establezca a cero el tiempo de espera, una tarea colgante nunca liberará su posición y por tanto podría retrasar el trabajo del cluster, por lo tanto al momento de establecer el tiempo de espera es necesario saber que se considera como progreso dentro de la ejecución de una tarea MapReduce. En el caso que una tarea falle en un gran número de ocasiones, no se volverá a intentar ejecutarla, el número de ocasiones es completamente configurable, pero por defecto es cuatro.

- Falla en el TaskTracker

  Otra forma de falla es la del tasktracker, al momento de que se produzca la falla dejará de enviar pulsaciones al jobtracker (o las enviará con una baja frecuencia), el jobtracker recibirá información en dónde se especificará que un tasktracker ha dejado de enviar pulsaciones y lo sacará de su lista de tasktrackers disponibles para ejecutar una tarea. El jobtracker organizará que las tareas map que se ejecutaron completamente de forma satisfactoria se ejecutarán otra vez en el mismo tasktracker, debido a que los datos intermedios de salida se encuentra en el archivo de salida del tasktracker fallido, el cual podría resultar inaccesible para las tareas reduce.

- Falla en el JobTracker

  Ésta resulta ser una de las fallas más importantes y serias. Actualmente Hadoop no posee ningún sistema o mecanismo para tratar con ésta clase de fallas, por lo tanto en éste caso el trabajo completo resulta fallido, sin embargo existen bajas probabilidades que se presente ésta clase de falla dado a lo difícil que es que falle una máquina en particular. Una posible solución sería el tener más de un jobtracker, pero mantener uno solo como el primario.

4.5.6. Mezclar y Ordenar

MapReduce garantiza que cada entrada a la función Reduce está ordenada según una clave, el proceso en que el sistema desarrolla éste ordenamiento y traspasa las salidas de las funciones Map como entradas a las funciones

Reduce se denomina Shuffle (mezclar), éste resulta ser un importante procedimiento para que el modelo MapReduce cumpla su cometido.

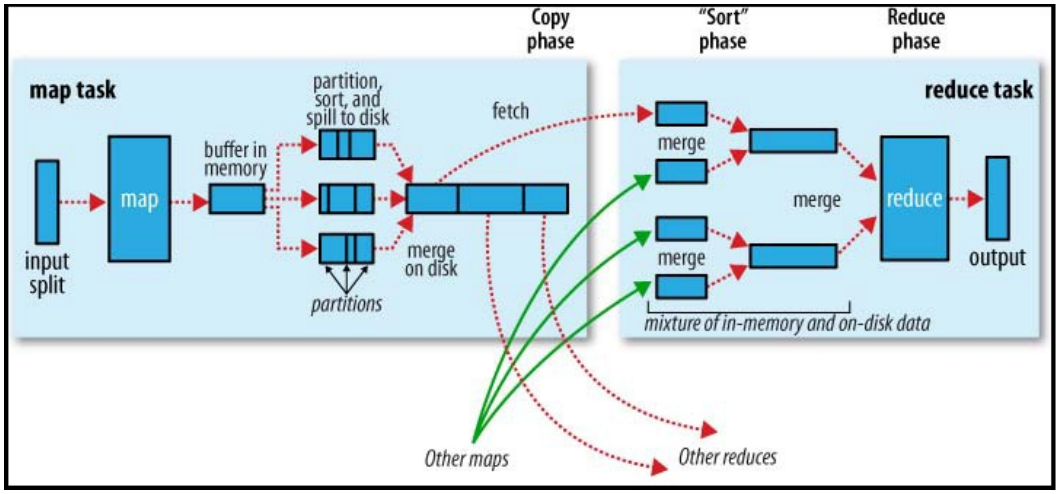

Figura 10: Shuffle y Sort en MapReduce.[7]

[7] Tom White. Hadoop, The Definitive Guide, capítulo 6, página 178. 2 edition, oct 2010.

- En Map

  Cuando la función map comienza a producir archivos de salida, no resulta ser una simple escritura hacia el disco, el proceso es mucho más complejo y utiliza memoria temporal e incluso un pre-ordenamiento que incrementa la eficiencia de todo el proceso, según se refleja en la figura 10. [10] Cada tarea Map posee una memoria temporal en donde escribir sus salidas (memoria por defecto de 100 mb.), cuando el tamaño del espacio ocupado en el temporal alcanza cierto tamaño, un proceso secundario comenzará a traspasar éstos contenidos al disco. Las salidas de Map continuarán escribiéndose en el temporal, mientras que por otro lado se siguen traspasando éstos contenidos al disco, en el caso que el temporal se llene, se detendrá la función Map hasta que se terminen de traspasar todos los datos. Luego estos datos son enviados a directorios específicos según el trabajo, utilizando round-robin. Antes de escribir éstos datos al Disco, el proceso divide los datos en particiones, correspondientes al número de reducers al que le serán enviados los datos. Dentro de cada partición, un proceso secundario realiza un ordenamiento por llave (clave).

Cada vez que la memoria temporal llega a su tope, se crea un nuevo archivo de traspaso, por lo tanto luego de escribir la última salida de la función Map podrían existir muchos archivos de traspaso. Antes que la tarea termine, los archivos de traspaso son combinados en un sólo archivo de salida (particionado y ordenado).

- En Reduce

  El archivo de salida de la función Map se encuentra en el disco local del tasktracker que ejecutó la tarea Map, pero ahora es requerido por el tasktracker que ejecutará la tarea Reduce sobre ésta partición. Las tareas Map pueden ir terminando en diferentes intervalos de tiempo, por lo tanto las tareas reduce copiarán sus salidas en el momento exacto en que las anteriores terminen, ésta fase se conoce como la fase de copia de las tareas Reduce. Las tareas Reduce poseen un número limitado de procesos que se encargan de realizar éstas copias, por lo tanto pueden realizarse de forma paralela sobre diferentes salidas Map.

  Las salidas de las tareas Map son copiadas en la memoria de los tasktrackers Reduce, y en el caso que no exista espacio suficiente se copian al disco. En el momento en que la memoria interna del tasktracker llega a su tope, se traspasan todos los datos al disco local de una sola vez. A medida que las copias que se acumulan en el disco, un proceso secundario las va uniendo en un archivo ordenado. Una vez que todas las salidas Map han sido copiadas, las tareas Reduce comienzan la etapa de ordenamiento, la que une las salidas de Map manteniendo su orden. En la fase Reduce, la función Reduce es invocada por cada llave (clave) en el archivo ordenado, la salida de ésta fase es directamente traspasada al sistema de archivos HDFS.

5. Pig

Pig eleva el nivel de abstracción en el procesamiento de grandes conjuntos de datos, ya que mientras MapReduce le exige al usuario el especificar y definir una función Map y una función Reduce, Pig no lo hace, sus estructuras de datos son más fuertes, casi siempre multievaluadas y anidadas, y además es posible realizar múltiples transformaciones a los datos, como una operación join. Es sabido que en MapReduce resulta altamente tedioso el implementar éste tipo de transformaciones. [10]

Pig posee dos componentes principales:

- Pig Latin Language.
- El ambiente de Ejecución (JVM o Hadoop).

El primero es el lenguaje que utiliza para expresar los flujos de datos, denominado Pig Latin, y el segundo componente es el ambiente de ejecución de programas escritos en Pig Latin, puede ser una ejecución local utilizando una JVM o de forma distribuida utilizando un cluster Hadoop. Un programa Pig Latin se compone de una serie de operaciones o transformaciones que se aplican sobre los datos de entrada. Muy por debajo de la visión de un programador, Pig convierte estas transformaciones en una serie de trabajos MapReduce, los cuales a su vez son ejecutados en un cluster Hadoop.

Una de las mayores críticas que posee MapReduce es en que el tiempo de desarrollo es demasiado extenso, escribiendo las funciones Map y Reduce, compilando y empaquetando el código, enviando los trabajos y recibiendo resultados, es un proceso complejo que consume demasiado tiempo. El potencial de Pig es el procesar terabytes de datos utilizando sólo una media docena de líneas de código, es más, es posible ejecutar el mismo análisis de datos sólo sobre una muestra de los datos totales, con el fin de establecer posibles errores en el procesamiento, antes de realizar el trabajo sobre todos los datos de entrada.

Pig fue diseñado para ser ampliable y configurable, es así como la búsqueda, el almacenamiento, filtrado, agrupamiento y unión de los datos, son procesos que pueden ser definidos por el usuario (User Defined Functions), que resultan ser altamente reutilizables.

5.1. Ejecutar un Programa Pig

Existen 3 formas de ejecutar un programa escrito en Lenguaje Pig, todas trabajan de forma local o a través de MapReduce (Hadoop).

- Script:

  Pig puede ejecutar un script que contenga comandos pig. Por Ejemplo: "*pigscript.pig*" ejecuta el comando en el archivo local "*script.pig*". Por otra parte para scripts muy cortos basta con usar la opción "$-e$" para ejecutar un script especificado como un string en la línea de comandos.

- Grunt:

  Es un shell interactivo para ejecutar comandos pig. Grunt se puede iniciar aun cuando no se especifique ningún archivo pig para ejecutarse, y la opción "$-e$" no se utiliza. También es posible ejecutar scripts pig en grunt con los comandos "*run*" and "*exec*".

- Embebido:

  También es posible ejecutar programas pig desde Java, así mismo como se utiliza JDBC para ejecutar programas SQL desde Java.

5.2. Pig y MapReduce

Para ejecutar programas escritos en Pig Latin sobre MapReduce, lo primero que se realiza es el transformar todas las consultas Pig en trabajos MapReduce que luego se ejecutarán sobre un cluster Hadoop, éste cluster puede ser pseudo o completamente distribuido. Los trabajos realizados de ésta forma disminuyen las líneas de código empleadas, así como el tiempo necesario para su desarrollo, según indica la figura 11. [9]

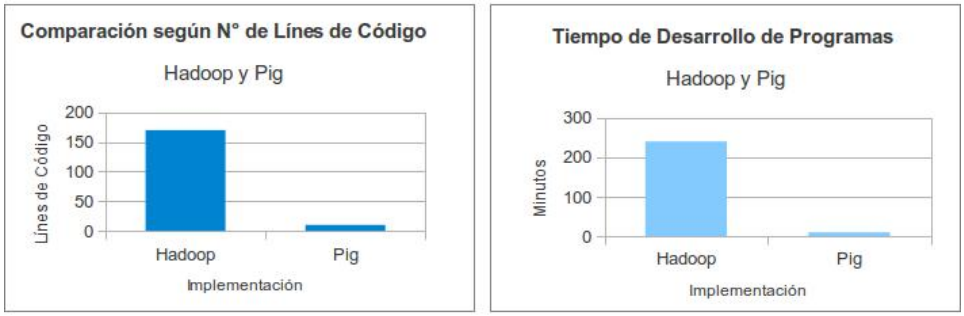

Figura 11: Gráficos Comparativos, según líneas de código y tiempo necesario para implementar un trabajo tanto en Hadoop como en Pig.

6. MongoDB

MongoDB es un sistema de base de datos NoSQL, multiplataforma y orientado a documentos, está escrito completamente en lenguaje c++, por lo tanto posee una gran cercanía hacia los recursos de hardware de la máquina, es por esto su alta velocidad de consulta, posee licencia libre y como se dijo anteriormente es multiplataforma, puede instalarse en variados sistemas operativos, como Linux, Solaris, y Windows. Como también se mencionó antes, uno de los grandes potenciales de MongoDb es su alta velocidad y además destaca la simplicidad de su sistema de consulta, con el cual es posible realizar consultas que resultarían complejas en una base de datos relacional, y además con una alta velocidad de respuesta. [6]

Éstos "*documentos*" mencionados en un inicio, corresponden a la denominación que recibe un conjunto de datos en MongoDB, éstos documentos a su vez se agrupan en colecciones, que corresponderían a una tabla de una base de datos relacional, la diferencia radica en que en las colecciones es posible almacenar documentos con formatos muy diferentes entre sí. En estos documentos es posible agregar, eliminar, modificar o renombrar nuevos campos, ya que no existen esquemas predefinidos.

MongoDB guarda la estructura de los datos en documentos de tipo JSON, con un esquema dinámico, que finalmente se denominan BSON. Los documentos de MongoDB poseen una estructura simple que se compone de pares (clave: valor), siguiendo el formato de JSON, a continuación en la figura 12 es posible apreciar ésta descripción.

Figura 12: Almacenamiento de Datos de Usuario en MongoDB.

MongoDB se utiliza en las siguientes situaciones o casos: [7]

- Almacenamiento y Registro de Eventos
- Para sistemas de manejo de documentos y contenido
- Comercio Electrónico

- Juegos
- Problemas de alto volumen
- Aplicaciones móviles
- Almacén de datos operacional de una página Web
- Manejo de estadísticas en tiempo real

  Existen muchas empresas que utilizan MongoDB en alguna de éstas situaciones, algunas de ellas son MTV, Craiglist, Foursquare y Disney.

6.1. Arquitectura de MongoDB

La arquitectura de MongoDB posee básicamente dos componentes importantes, que corresponden a mongod y mongos, la descripción de cada uno de estos a continuación:

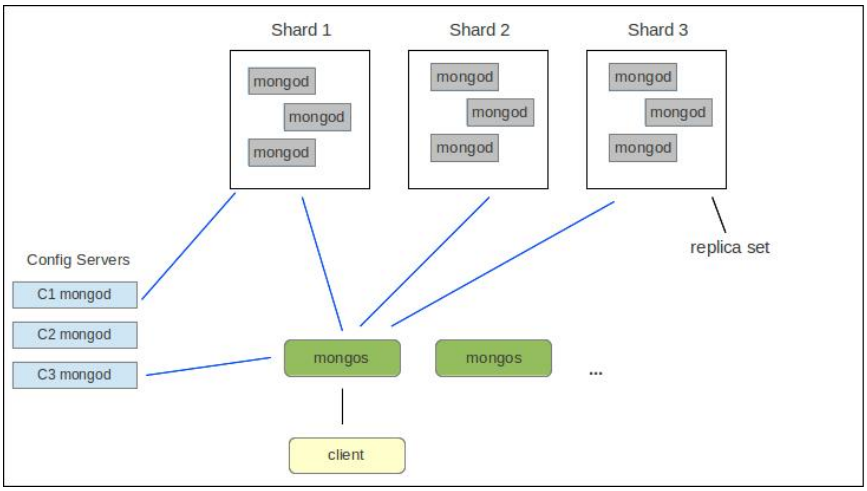

Figura 13: Arquitectura de MongoDB.

- mongod: Corresponde al motor o núcleo central de la base de datos, similar a mysqld en MySQL, pude funcionar de 3 formas diferentes: standalone server, config server y como shard partition (ver figura 13).
- mongos: Los mongos son los controladores del particionamiento de los datos, además se encargan de enviar y recibir consultas o datos y realizar el balanceo de los datos dentro de cada shard.

En la figura 13 se puede apreciar como los mongos funcionan como una especie de enrutadores de los datos desde o hacia los mongod que funcionan como config server o los que trabajan dentro de cada particion de los datos (shard partition), además en la parte inferior de la figura se visualiza el cliente (client) que es donde finalmente mongos envía las respuestas a las peticiones realizadas en un inicio.

6.2. MongoDB y MapReduce

MongoDB permite utilizar funciones MapReduce escritas en Javascript para seleccionar atributos dentro de los datos y trabajar con ellos como se desee (agregar, unificar, simplificar), a pesar que esta clase de consultas resulta habitual en muchos sistemas SQL, MongoDB simplifica el trabajo necesario para realizar éste tipo de tareas, incluso no es necesario recurrir a MapReduce en algunos de éstos casos. Una búsqueda clásica dentro de MongoDB puede verse reflejada en la figura 14, en dónde se realiza una búsqueda de los posts etiquetados con "*MongoDB*"

```
db.posts.find({'tags' : 'MongoDB'})
```

Figura 14: Búsqueda de valores en MongoDB.

7. Implementación Algoritmo Join Utilizando MapReduce

7.1. Datos a Analizar por los Algoritmos

Los datos que se utilizan dentro de todas las implementaciones consisten básicamente en dos tablas en el caso de las implementaciones en Hadoop y Pig y a dos colecciones en el caso de MongoDB, los campos y valores que poseen estas tablas y colecciones son los mismos, solo cambia su formato. A continuación las tablas implicadas en las implementaciones:

- Empleado: Consta del rut y el código del departamento al cual pertenece dentro de la empresa (ver figura 15).
- Departamento: Contiene el código del departamento y el nombre del mismo (ver figura 16).

La operación Join en todos los casos se realiza sobre el código del Departamento que es el que conecta a las dos tablas anteriores.

RUT	COD_DEPTO
16403093	D1
16200545	D1
16444768	D2
16887584	D2
18777876	D3
12345678	D3

Figura 15: Tabla de Empleados de la Empresa.

COD_DEPTO	NOMBRE_DEPTO
D1	RRHH
D2	LOGISTICA
D3	INFORMATICA

Figura 16: Tabla de Departamentos de la Empresa.

7.2. Algoritmo Join

Una operación Join se define como el operador que permite combinar registros de dos o más tablas dentro una Base de Datos Relacional. Dentro del lenguaje SQL existen 3 tipos de Join: Interno, Externo y Cruzado, cada uno de éstos maneja los registros de diferentes formas y entrega así mismo diferentes resultados sobre las mismas tablas.

Con el fin de optimizar las operaciones que realiza un sistema MapReduce sobre los datos de ingreso ('*Logs*') dentro de una plataforma establecida, es muchas veces necesario realizar consultas o referencias hacia otras tablas, que contienen por ejemplo datos personales del usuario, datos sobre conexiones previas, etc. La utilización de un comando Join resulta imposible en éste caso, ya que Hadoop no posee este comando y la técnica MapReduce no fue diseñada ni pensada para realizar éste tipo de tareas sobre los datos, sin embargo es posible una operación Join en Hadoop utilizando MapReduce a través de ciertas técnicas.

El intentar implementar este tipo de operaciones sobre MapReduce representa cierta complejidad, debido a que MapReduce no fue diseñado para combinar información de dos o más fuentes de datos. Por lo general los programadores de MapReduce utilizan algoritmos para imitar el comportamiento de un Join sobre los datos, pero éstos resultan ineficientes y engorrosos.

Existen ciertas estrategias para implementar algoritmos Join dentro de MapReduce, la idea siempre es mantener la estructura central de MapReduce y no alterarla bajo ninguna circunstancia, con el fin de mantener la alta tolerancia a fallas y la alta velocidad de procesamiento, inherentes en esta metodología. Las estrategias que pueden ser implementadas son [3]:

- Repartition Join
- Broadcast Join
- Semi-join
- Per-Split Semi-Join

Los dos algoritmos que se implementarán en este trabajo corresponden a Repartition Join y al Semi Join, a continuación se profundizará en las estrategias mencionadas:

- Repartition Join:

  Se cuenta con una tabla de Empleado y una tabla de Departamento de dos columnas (ver figuras 15 y 16). Donde cada una de ellas es

particionada, y donde cada una de estas particiones se transforma en las entradas de la fase Map.

En la fase Map, cada función Map trabaja sobre una división de Empleados o Departamentos, para identificar de cuál de las dos tablas proviene ésta entrada, cada tarea Map guarda una etiqueta con su tabla de origen, luego de esto, la salida de la función Map será una tupla (clave, valor) como es sabido, pero en éste caso los pares se conforman de una clave y un valor compuesto, el cual corresponde a la etiqueta de la tabla de donde proviene la partición y un registro del resto de valores que poseen las columnas de la tabla. Luego estos pares son ordenados y agrupados por su clave, que es este caso corresponde al número del departamento al cual pertenece el Empleado, antes de ser enviados a la función Reduce.

Luego por cada clave, la función Reduce separa y almacena los archivos de entrada en dos partes, de acuerdo al indicador de tabla. Finalmente realiza un producto cruzado entre los registros de éstos conjuntos.

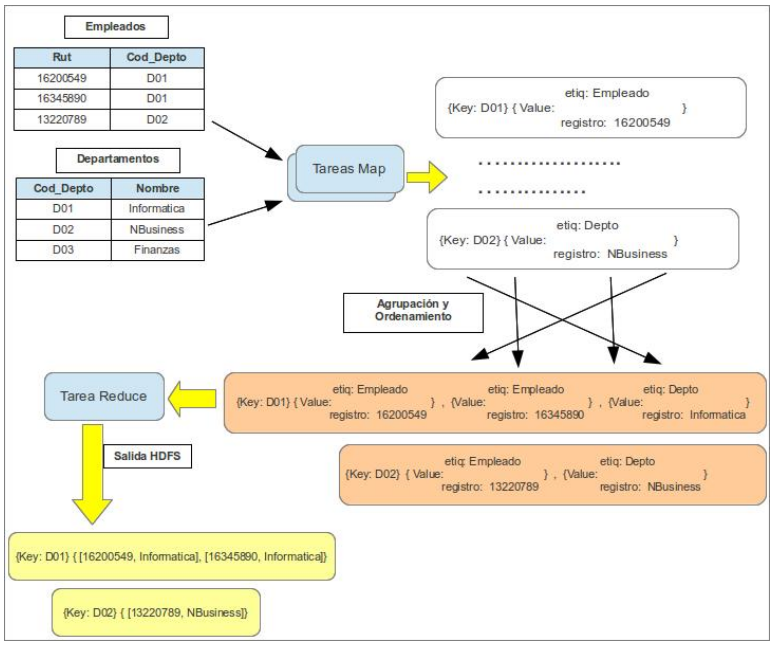

Figura 17: Repartition Join en MapReduce.

- Semi-Join:

  En más de una ocasión una de las dos tablas es más grande que la otra, en éste caso en específico la tabla Empleado es más grande que la tabla Departamento, la técnica semi-Join posee 3 fases, en donde cada una corresponde a un trabajo MapReduce:

  - Primera Fase: Al principio se ejecuta un trabajo MapReduce completo. En la función Map una tabla hash en memoria principal se utiliza para determinar el conjunto de claves en una parte de la tabla Departamento. Enviando las claves únicas como salida de Map, se reduce la cantidad de datos que deben ser ordenados. Luego la salida de la función es simplemente la consolidación de estas claves únicas en un solo archivo final (ver Fase I, figura 18).
  - Segunda Fase: Ésta fase es similar al método Broadcast Join, en donde se ejecuta sólo una fase Map. Inicialmente se carga la salida de la Primera fase en una tabla hash en memoria principal (que corresponde a las claves de la tabla Departamento), luego de esto la función Mapper itera sobre cada registro de la tabla Empleado y en el caso de encontrar una clave que se encuentre en la tabla hash , se realiza una salida. La salida final de ésta fase corresponde a una lista de archivos R_i, uno por cada división de R (corresponden solo a los empleados que poseen una relación con la tabla Departamento).
  - Tercera Fase: Finalmente todos los archivos R_i se unen a la tabla Departamento utilizando la metodología Broadcast Join, la cual básicamente consiste en obtener el resultado de la fase número 2 y almacenarlos en una tabla Hash para luego compararlos con cada lectura de la fase Map sobre la tabla Departamento, de éste modo se emiten directamente las salidas de la función map sin realizar ninguna clase de ordenamiento ni agrupación por parte de Hadoop, disminuyendo considerablemente el tiempo de ejecución. La salida final será el Rut de cada empleado con el nombre del departamento al cual pertenecen dentro de la empresa.

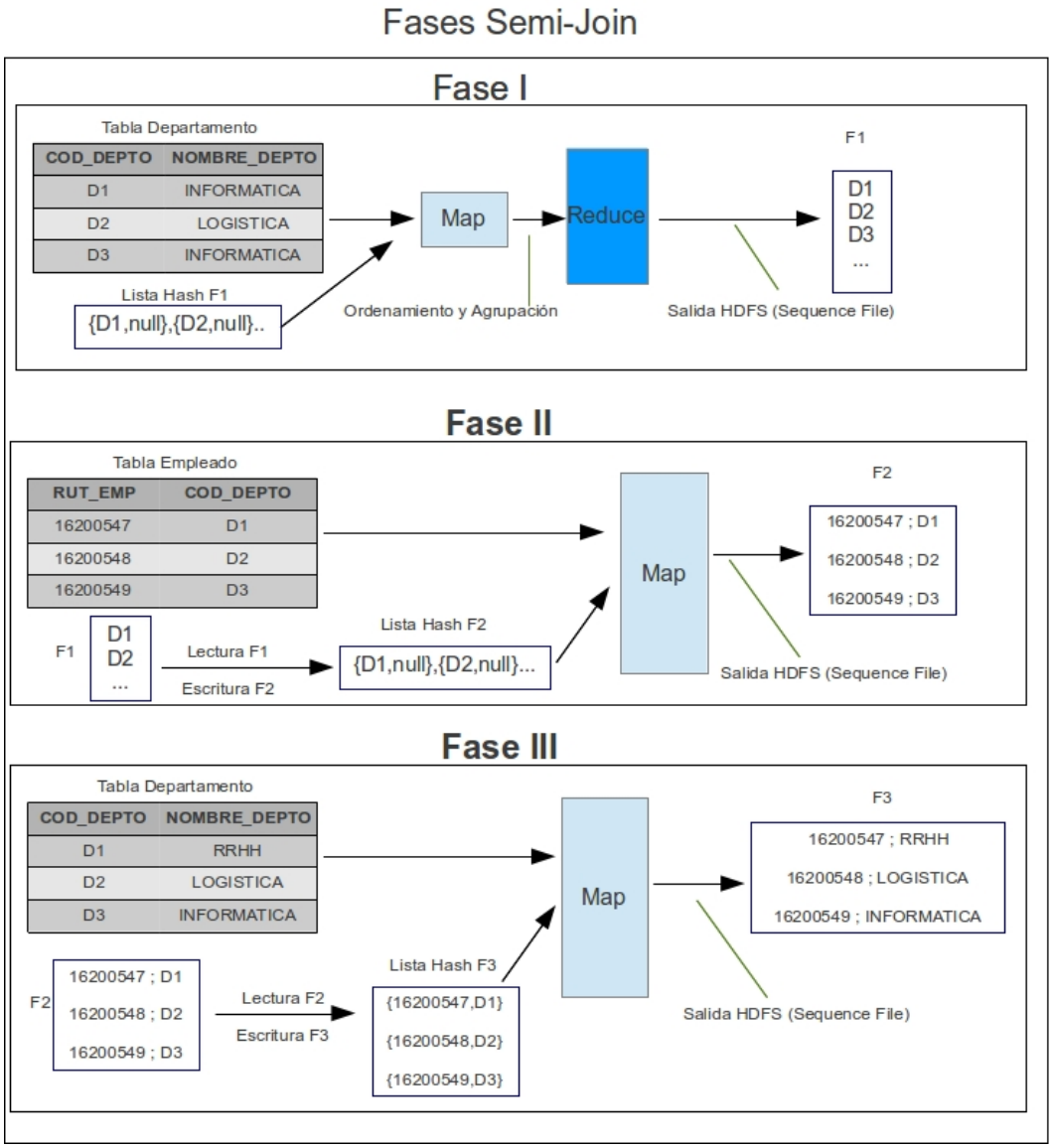

Figura 18: Semi Join en MapReduce.

7.3. Implementación en Hadoop

7.3.1. Datos Técnicos Codificación

La codificación de los algoritmos se desarrolló en lenguaje de programación java, utilizando la plataforma Eclipse versión 3.3.2 Europa sobre el sistema operativo Ubuntu 12.04. La Plataforma Eclipse utiliza un plug-in que incorpora Hadoop dentro de su sistema, especialmente diseñado para crear proyectos MapReduce sobre ésta plataforma.

7.3.2. Datos Técnicos Ejecución de Programas

A continuación se desarrollarán los detalles ténicos implicados dentro de la ejecución del programa repartition join side y semi join:

- Sistema Operativo: El sistema operativo como se mencionó anteriormente es Ubuntu 12.04 de 32 bits de direccionamiento, posee un kernel Linux 3.2.0 publicado el 27 de julio de 2012.
- Máquina: La máquina sobre la que se ejecuta el programa corresponde a un Notebook con procesador Intel Core I5-2430M de 3.00GHz de 64 bits de direccionamiento, posee una memoria principal de 6 gb y una memoria secundaria de 640 gb.
- Hadoop: La instalación de Hadoop sobre la máquina se encuentra de forma semi distribuida, donde los procesos secundarios de Hadoop simulan como si trabajasen en un cluster, la versión de Hadoop corresponde a la 1.0.3 lanzada el 16 de Mayo del presente año.
- Java: Corresponde a la versión 1.6.0.35 de Sun MicroSystem, debido a que según los desarrolladores de Hadoop es la óptima para un buen desempeño del framework, ésta versión fue descargada de los servidores de Oracle debido a la reciente adquisición de Sun MycroSystems por parte de Oracle.
- Datos de Entrada: Las dos tablas que se utilizan están codificadas en texto plano UTF8.

7.3.3. Implementación Repartition Join

Como una prueba inicial se implementó el algoritmo Join utilizando la técnica Repartition Join, la cual se basa en realizar la unión de los registros en la etapa Reduce tal como muestra la figura 17, en ésta implementación se trabajó con dos tablas, con una función Map y una función Reduce.

1. **Clases Java Involucradas**

   Existen dos clases principales dentro del programa, una relacionada con la operación Map y la otra con Reduce:

   - Map: Clase que recibe la tupla proveniente de una de las dos tablas implicadas, la cual puede estar formada por (rut empleado, código departamento) o (código departamento, nombre departamento), según sea la tabla recibida, luego de esto envía una tupla que se compone por la clave sobre la que se realizará el join, que en este caso corresponde al codigo del departamento, y a su vez una clave compuesta por la etiqueta de la tabla desde la que se realizó la lectura y el valor de la columna restante, esto se puede ver reflejado en la figura 19. En el caso específico de la función Map sobre la tabla Empleado la clave de salida estará compuesta de la siguiente forma: $(codigo_departamento, (\text{"}etiquetatabla\text{"}, rutempleado))$.

```
Map (clave, valor: un registro de una división de una tabla)
{
etiqueta_tabla = identificador_tabla;
clave = clave principal de la tabla;
campo_1 = se extrae el resto del contenido de la tabla;
Emitir (clave, (etiqueta_tabla,campo_1));
}
```

   Figura 19: Pseudocódigo Método Map Repartition Join.

   - Reduce: Esta clase recibe las salidas de la función Map agrupadas y ordenada según clave (codigo departamento), luego de esto se encarga de recorrer la lista de valores que posee cada clave y realiza una salida a través de un producto cruz sobre los campos que posean diferente etiqueta de tabla y que además posean el mismo código de departamento, esto se puede visualizar en el pseudocódigo de la figura 20.
   - Driver: Define el formato de salida del programa MapReduce, establece los archivos de entrada hacia la funcion Map. Además establece los nombres de las clases encargadas de las operaciones Map y Reduce y finalmente ejecuta el programa.

```
Reduce (clave, lista_valores: lista que contiene la etiqueta
de una tabla y un campo de la misma){

//crear temporales por cada etiqueta
    For (t : cada registro en lista_calores)
   //asignar el valor a el temporal correspondiente

//Join sobre los dos temporales
   For (a : producto_cruz_temporales){

    Si (temporales son de diferente tabla)

  Emite(temporal.campo(1),temporal.campo(2));
   }
}
```

Figura 20: Pseudocódigo Método Reduce Repartition Join.

2. **Código Programa Repartition Join**

Como se mencionó anteriormente el código utilizado fue desarrollado completamente sobre una plataforma java y consta de 2 clases principales, además de la clase encargada de unir el trabajo de las dos anteriores, en la figura 21 es posible apreciar los sectores importantes del método map contenido en la clase Map, donde la sección destacada en azul indica la forma en que se realiza la diferencia de salida según sea la tabla con la que se está trabajando.

```
public void map(LongWritable key, Text value, OutputCollector<Text, Text> output,
                    Reporter reporter) throws IOException
{

   String line = value.toString();
   String splitarray[] = line.split(",");

       clave_intermedia=splitarray[0].trim();
       valor_intermedio=splitarray[1].trim();

     if(clave_intermedia.contains("D")){

      fileTag="depto~";
      output.collect(new Text(clave_intermedia), new Text(fileTag+valor_intermedio));

     }else{

      clave=valor_intermedio;
      valor=clave_intermedia;
      fileTag="empleado~";

       output.collect(new Text(clave), new Text(fileTag+valor));
        }
}
```

Figura 21: Extracto Código Java Método Map Repartition Join.

Luego se tiene el método Reduce en la figura 22, éste método resulta

un tanto más complejo debido a que es necesario almacenar los valores para luego ir emitiendo las salidas correspondientes, para estos efectos es necesario crear temporales por cada tabla, que luego se leerán uno a uno y se compararán de modo de establecer que posean la misma clave, que en este caso corresponde al código del departamento. Finalmente se obtendrá una salida compuesta por un Rut de empleado y un nombre de departamento.

```
public void reduce(Text clave, Iterator<Text> values, OutputCollector<Text, Text>
                output, Reporter reporter) throws IOException
  {

     while (values.hasNext()){

           String valor_actual = values.next().toString();
           String division_tupla_valor[] = valor_actual.split("~");

         if(division_tupla_valor[0].equals("empleado")){

           emp.add(division_tupla_valor[1].trim()+"~"+clave);

           }else{
          dep.add(division_tupla_valor[1].trim()+"~"+clave);
           }
     }

      for(i=0;i<emp.size();i++){
          for(j=0;j<dep.size();j++){

              //DEFINICION DE VARIABLES
              if  (campo1[1].equals(campo2[1])){

                output.collect(new Text (campo1[0]),new Text (campo2[0]));

       }}}}
```

Figura 22: Extracto Código Java Método Reduce Repartition Join.

Dentro de cada uno de estas clases existen variables especiales de la librería MapReduce sobre Hadoop:

- Clase Text: se utiliza para almacenar datos en codificación UTF8 [8] y provee métodos internos que permiten trabajar a nivel de bytes con los datos.
- Interfaz OutputCollector: Es la encargada de recolectar los datos con formato de tuplas ya sea desde el maper o el reducer.
- Interfaz Reporter: Proporciona datos en tiempo de ejecución y estados sobre el programa MapReduce.

[8] UTF-8 (8-bit Unicode Transformation Format) es un formato de codificación de caracteres Unicode e ISO 10646 utilizando símbolos de longitud variable. UTF-8 fue creado por Robert C. Pike y Kenneth L. Thompson.

- Interfaz JobConf: Es el modo de indicar un trabajo MapReduce hacia la plataforma Hadoop.
- Clase JobClient: Es la Interfaz primaria que el usuario posee para interactuar con los estados del trabajo que se está ejecutando sobre Hadoop, reporta errores, avance, etc.
- Clase JobClient.runjob: Utilidad dentro de la clase JobClient que presenta el trabajo ante Hadoop y muestra su progreso hasta que se realice completamente. [1]

3. **Resultados Obtenidos**

   Luego de realizar la ejecución del programa sobre la instalación de Hadoop semi distribuida los resultados obtenidos se ven reflejados en la siguiente tablas, en donde la tabla 1 muestra los resultados obtenidos tras procesar 10 empleados, mientras que la tabla 2 se basa en procesar 10000 empleados:

Item	**Valor**
N° Tareas Map Ejecutadas	3
N° Tareas Reduce Ejecutadas	1
N° Bytes de Entrada Map	267
N° Registros de Salida Map	20
N° Registros de Entrada Reduce	20
N° Grupos de Entrada Reduce	10
N° Registros de Salida Reduce	10
Tiempo Uso de CPU en Map	1070 ms
Tiempo Uso de CPU en Reduce	3860 ms
Tiempo Uso de CPU Total	4930 ms

Tabla 1: Tabla de Resultados ejecución Repartition Join en MapReduce sobre 10 empleados.

Como se puede apreciar en la tabla 1 el tiempo resulta mínimo (milisegundos), claro está que la cantidad de datos es pequeña, pero permite visualizar la forma de trabajo de MapReduce sobre la implementación del algoritmo Join. Una segunda ejecución sobre 10000 empleados es descrita en la tabla 2 con sus valores correspondientes.

Los pares obtenidos se pueden apreciar en la figura 23, en donde finalmente los pares obtenidos resultan en una unión perfecta de las dos

Item	Valor
N° Tareas Map Ejecutadas	3
N° Tareas Reduce Ejecutadas	1
N° Bytes de Entrada Map	130137
N° Registros de Salida Map	10010
N° Registros de Entrada Reduce	10010
N° Grupos de Entrada Reduce	10
N° Registros de Salida Reduce	10010
Tiempo Uso de CPU en Map	3570 ms
Tiempo Uso de CPU en Reduce	4350 ms
Tiempo Uso de CPU Total	7920 ms

Tabla 2: Tabla de Resultados ejecución Repartition Join en MapReduce sobre 10000 empleados.

tablas que pertenecen al caso de estudio, donde se forma una nueva tabla que consiste en el Rut del empleado de la empresa y el nombre del departamento al cual pertenece.

RUT	NOMBRE_DEPTO
16403093	RRHH
16200545	RRHH
16444768	LOGISTICA
16887584	LOGISTICA
18777876	INFORMATICA
12345678	INFORMATICA

Figura 23: Tabla Formada con Repartition Join utilizando MapReduce.

7.3.4. Implementación Semi Join

Esta implementación consta de cinco clases java en total, dos clases representan a la Fase 1 descrita en la figura 18 que consiste en un trabajo MapReduce completo, otras dos clases representan a la Fase II y III que consisten en dos fases Map únicamente, y la quinta clase corresponde al Driver encargado de llevar a cabo los diferentes trabajos MapReduce sobre Hadoop.

1. **Clases Java Involucradas**

   Las clases java y su funcionamiento básico se describen a continuación:

   - Map1: Esta clase se encarga de recibir como tuplas los datos de la tabla Departamento, se recibe el código de cada uno de los departamentos y su nombre asociado, una de las principales características de esta clase, es el contar con una lista Hash, que se utiliza en el caso que algún código de departamento se encuentre repetido dentro de la tabla (en el caso que no sea una Primary Key por supuesto), ya que al mismo tiempo que la función Map va leyendo las tuplas, va almacenando el código del departamento en la lista hash, no sin antes buscar dentro de la misma por posibles coincidencias, en cuyo caso el dato no es emitido hacia la función Reduce siguiente, como indica la figura 24.

```
Map1(clave:null,valor:un registro de la tabla Departamento)
{
clave_join = extraer la clave del depto;

If (clave_join no esta en lista_hash_1){

lista_hash_1.agregar(clave_join,null);

emit(clave_join,null);
}
}
```

   Figura 24: Pseudocódigo Función Map 1 Semi Join.

   - Reduce1: Es la terminación del único trabajo MapReduce completo en este algoritmo, recibe cada una de las claves de unión emitidas anteriormente por Map1 y las escribe sobre un HDFS final, según indica la figura 25.

```
Reduce1 (clave: clave_join, valor:null){
emit (clave_join,null);
}
```

   Figura 25: Pseudocódigo Función Reduce 1 Semi Join.

   - Map2: Es en esta parte en donde comienza la ejecución solo de tareas Map, la primera de ellas es Map2 que realiza un análisis sobre la tabla Empleado, recibiendo cada uno de sus registros. El

análisis que marca la diferencia corresponde a una lista hash que posee todos los datos emitidos anteriormente por Reduce1, de ésta forma Map2 solo emitirá al empleado que posea o esté dentro de un departamento de la empresa, para comprobarlo, inicialmente se carga el Archivo de Texto del HDFS [9] dentro del hashmap, que ahora posee todos los códigos de los departamentos de la empresa, los cuales son comparados con los códigos que posee cada uno de la empleados, si alguno de ellos no corresponde a un código dentro del hashmap, se descarta, de lo contrario se emite hacia un archivo HDFS final que contendrá el registro completo de la tabla Empleado (Rut empleado y código depto.), esto puede verse reflejado en la figura 26.

```
Init()
lista_hash = cargar salida reduce1;

Map2 (clave:null, valor: un registro de la tabla Empleado){
clave_union = cod_depto de cada Empleado;

If(clave_union esta en lista_hash){
emitir(null , valor);
}
}
```

Figura 26: Pseudocódigo Función Map 2 Semi Join.

- Map3: Fase final, en donde la tabla de entrada corresponde a la tabla Departamento, anteriormente todos los empleados fueron almacenados en un archivos de Texto HDFS, ahora éstos datos deben ser almacenados en una lista hash para luego realizar la unión de los mismos con los nombres de los departamentos a los que correspondan. El paso central es realizar una comparación entre el código del departamento de la tabla Departamento y el código del departamento que posee cada uno de los empleados almacenados en la lista hash, en el caso que éstos coincidan, se emitirá el Rut del empleado junto con el nombre del departamento al cual pertenece, en un archivo de Texto. En la figura 27 se refleja el pseudocódigo de la operación descrita anteriormente.

[9] Archivo de Texto HDFS (TextOutputFormat) es un formato de salida de Hadoop que permite almacenar los datos como String en tuplas (clave,valor) en base a este formato es como deben ser leídos.

```
Init()
lista_hash = cargar salida map2;

Map3 (clave:null, valor: un registro de la tabla
Departamento){
clave_union = cod_depto de cada Departamento;

While( lista_hash ){
If(clave_union esta en lista_hash.clave){

emitir(lista_hash.rut , valor.nombre_depto);
}}
}
```

Figura 27: Pseudocódigo Función Map 3 Semi Join.

2. **Código Programa Semi Join**

   Como se mencionó anteriormente la implementación consta de tres clases Map, una clase Reduce y una clase Driver encargada de coordinar los trabajos en la ejecución, las últimas dos fases del algoritmo constan sólo de trabajos Map, en éstas clases existe otra función que se encarga de almacenar datos en una lista hash, según se puede apreciar en la figura 29 en el segundo texto destacado en azul, en donde el método init() recupera la salida de la Fase 1 (ver figura 28) del algoritmo para poder luego trabajar con ella.

   Luego de realizar la recuperación de la salida de la fase 1 en el método init() se procede a realizar dentro de la misma clase Map Phase 1 el método, que trabaja tanto con la lista hash creada anteriormente, como con la tabla leída directamente por el Map, que en éste caso es la tabla Empleado, ésta función se refleja en la figura 30.

   La Fase tres y final del algoritmo también posee un método init() encargado de recuperar la salida del método Map de la fase dos y almacenarlo en una lista hash, luego el análisis final es realizado por un último método map que recibe la tabla Departamento y lee cada uno de sus registros a la vez que realiza una comparación con los datos del hashmap, como se explicó en la sección 5.2.5 punto número 1 Map3, el resultado es la salida de las dos tablas unidas según su clave de unión en un archivo HDFS de texto (ver figura 31).

   Una de las complejidades del algoritmo consiste en realizar múltiples trabajos MapReduce en sólo una implementación o codificación, por lo tanto la clase Driver juega un papel fundamental dentro de éste programa, se encarga de definir los diferentes trabajos y cada una de las

```
 public class Map_Phase_1 extends MapReduceBase implements Mapper<LongWritable, Text,
Text, Text>
//este map se encarga de procesar la tabla departamento//
{
      private String clave_join =null;
      private String nombre_depto = null;
      private static Map<String,String> Lista_Hash= new HashMap<String,String>();
      private int bandera_igualdad = 0;

  public void map(LongWritable key, Text value, OutputCollector<Text, Text> output,
Reporter reporter) throws IOException {
      //se debe extraer la clave del depto

        String linea = value.toString();
        String splitarray[] = linea.split(",");
        clave_join=splitarray[0].trim();
        nombre_depto = splitarray[1].trim();

 //luego se debe verificar que la clave sea unica y que no este en el hashmap Lista_Hash

        Iterator iter = Lista_Hash.entrySet().iterator(); //para recorrer el hashmap

        while (iter.hasNext()) {
            Map.Entry e = (Map.Entry)iter.next();
            if (e.getKey().toString().equals(clave_join)){  //se recorre el hashmap
              bandera_igualdad = 1;
   //si la clave esta en el hashmap se levanta la bandera
            }
        }

        if (bandera_igualdad == 0){
            Lista_Hash.put(clave_join,null); //se almacena la clave en el hashmap

            output.collect(new Text(clave_join),new Text(nombre_depto));
            //se envia la clave join hacia el reduce phase1

        }
        }
    }
```

Figura 28: Clase Map Phase 1, Método Map() Semi Join.

```
public class Map_Phase_2 extends MapReduceBase implements Mapper<LongWritable, Text,
Text, Text>
/*esta funcion map se encarga exclusivamente de la tabla empleados
inicialmente carga los valores obtenidos en el reduce anterior en un nuevo hashmap
 en el metodo init*/
{

      private static Map<String,String> Lista_Hash_2= new HashMap<String,String>();
//nueva lista hash
      private String clave_join = null;
      private String clave_hashmap = null;
      private String valor_hashmap = null;
      private String linea = null;
      int i=0;

    public void init(int numero_archivo) throws IOException{
      //aqui se crea un objeto hdfs para luego acceder al sistema de archivos

      Configuration config = new Configuration();
      config.addResource(new Path("/usr/local/hadoop/conf/core-site.xml"));
        config.addResource(new Path("/usr/local/hadoop/conf/hdfs-site.xml"));
        config.addResource(new Path("/usr/local/hadoop/conf/mapred-site.xml"));
        String uri = "hdfs://localhost:9000/semi_join/fase_1/part-0000"+numero_archivo;
        //archivo HDFS
        FileSystem fs = FileSystem.get(config);
        FileStatus[] files = fs.globStatus(new Path(uri));
        if (files.length!=0){
        for(FileStatus file:files){ //recorrer el archivo
            if(file.getLen() > 0){  //si el archivo tiene contenido
                FSDataInputStream in = fs.open(file.getPath());
                BufferedReader bin = new BufferedReader(new InputStreamReader(in));
                String s = bin.readLine(); //se lee una linea del archivo
                while(s!=null){
                  Lista_Hash_2.put(s, null); //se almacena los cod_deptos en el hashmap
                  s = bin.readLine();
                }
                in.close();
    }}}}
```

Figura 29: Clase Map Phase 2, Método Init() Semi Join.

```
public class Map_Phase_2 extends MapReduceBase implements Mapper<LongWritable, Text,
Text, Text> {
      private static Map<String,String> Lista_Hash_2= new HashMap<String,String>();
//nueva lista hash
      private String clave_join = null;
      private String clave_hashmap = null;
      private String valor_hashmap = null;
      private String linea = null;
      int i=0;
public void map(LongWritable key, Text value, OutputCollector<Text, Text> output,
Reporter reporter) throws IOException
    {
      String line = value.toString();
      int bandera_archivo = 1;
        String splitarray[] = line.split(",");
        clave_join = splitarray[1].trim();
      /*separamos los datos obtenidos de la tabla empleado donde el segundo valor
       es el codigo del departamento en este caso la clave_join*/
        Map_Phase_2 objeto = new Map_Phase_2();
        while(bandera_archivo == 1){
        try {
                  objeto.init(i);
                  i++;  //recorre los supuestos archivos secuenciales
            } catch (Throwable e1) {

                  bandera_archivo=0;
                  e1.printStackTrace();
            }
        }
Iterator iter = Lista_Hash_2.entrySet().iterator(); //para recorrer el hashmap

            while (iter.hasNext()) { //se recorre el hashmap

            Map.Entry e = (Map.Entry)iter.next();
            linea = e.getKey().toString();
            String division[] = linea.split("~");
            clave_hashmap = division[0].trim();//codigo depto
            if (clave_hashmap.equals(clave_join)){
                  output.collect(null, value);
//si la clave esta en el hashmap se emite el empleado::value =(rut_empleado,codigo_depto)
}}}}
```

Figura 30: Clase Map Phase 2, Método Map Semi Join.

```
public class Map_Phase_3 extends MapReduceBase implements Mapper<LongWritable, Text,
Text, Text>
{
      private static Map<String,String> Lista_Hash_3= new HashMap<String,String>();
      private String clave_join = null;
      private String nombre_depto = null;
      private String clave_hashmap = null;
      private String valor_hashmap = null;
      private String linea = null;
      private int i = 0;

public void init(int numero_archivo) throws Throwable{
...
//Metodo init similar al método de Map Phase 2
...
}
 public void map(LongWritable key, Text value, OutputCollector<Text, Text> output,
Reporter reporter) throws IOException{
      //este map recibe los datos de la tabla deptos
      String line = value.toString();
        String splitarray[] = line.split(",");
        int bandera_archivo = 1;
            //separamos los datos obtenidos de la tabla deptos donde el segundo valor
           //es el nombre_del depto y el primero es el codigo
        clave_join=splitarray[0].trim(); //este es el codigo del depto
        nombre_depto=splitarray[1].trim(); //este es el nombre del depto
        Map_Phase_3 objeto = new Map_Phase_3();
        while(bandera_archivo==1){
                  try {
                      objeto.init(i);
                        i++;
                  } catch (Throwable e1) {
                        e1.printStackTrace();
                  }}

        Iterator iter = Lista_Hash_3.entrySet().iterator(); //para recorrer el hashmap

        while (iter.hasNext()) {

            Map.Entry e = (Map.Entry)iter.next();
            linea = e.getKey().toString();
            String division[] = linea.split(",");
            valor_hashmap = division[0].trim();//este es el rut del empleado
            clave_hashmap = division[1].trim();//codigo depto del empleado

            if (clave_hashmap.equals(clave_join)){  //se recorre el hashmap

                  output.collect(null,new Text(valor_hashmap+","+nombre_depto));
}}}}
```

Figura 31: Clase Map Phase 3, Método Map Semi Join.

salidas y entradas de éstos, que luego serán utilizadas por el siguiente trabajo, así sucesivamente hasta obtener un resultado final, cada uno de los trabajos puede identificarse en la figura 32 en el texto destacado en azul y en verde cada una de las salidas y entradas.

```
public class Driver_Semi_Join{
      public static void main (String[] args) throws Exception {
            //primer jobconf
           JobConf conf = new JobConf(Driver_Semi_Join.class);
           conf.setJobName("fase1"); //nombre del trabajo

           conf.setOutputKeyClass(Text.class);
           conf.setOutputValueClass(Text.class);

           conf.setMapperClass(Map_Phase_1.class);
           conf.setReducerClass(Reduce_Phase_1.class);

           conf.setOutputFormat(TextOutputFormat.class);

           FileInputFormat.setInputPaths(conf,new Path(args[0]));//entrada map1

           FileOutputFormat.setOutputPath(conf,new Path ("/semi_join/fase_1"));
//archivos de salida reduce 1 en hdfs
           JobClient.runJob(conf);
//segundo jobconf
          JobConf conf2 = new JobConf(Driver_Semi_Join.class);
          conf2.setJobName("fase2"); //nombre del trabajo

          conf2.setOutputKeyClass(Text.class);
          conf2.setOutputValueClass(Text.class);

          conf2.setMapperClass(Map_Phase_2.class);
          conf2.setNumReduceTasks(0);
          conf2.setNumMapTasks(1);
          conf2.setOutputFormat(TextOutputFormat.class);

          FileInputFormat.setInputPaths(conf2,new Path(args[1]));//entrada map2

          FileOutputFormat.setOutputPath(conf2,new Path("/semi_join/fase_2"));
//archivos de salida map2 en hdfs
          JobClient.runJob(conf2);
          //tercer jobconf
          JobConf conf3 = new JobConf(Driver_Semi_Join.class);
          conf3.setJobName("fase3"); //nombre del trabajo

          conf3.setOutputKeyClass(Text.class);
          conf3.setOutputValueClass(Text.class);

          conf3.setMapperClass(Map_Phase_3.class);
          conf3.setNumReduceTasks(0);

          conf3.setOutputFormat(TextOutputFormat.class);

          FileInputFormat.setInputPaths(conf3,new Path(args[2]));//entrada map3

          FileOutputFormat.setOutputPath(conf3,new Path("/semi_join/fase_3"));
//archivos de salida map3 en hdfs
          JobClient.runJob(conf3);
 }}
```

Figura 32: Clase Driver Semi Join, Método Main.

3. **Resultados Obtenidos**

   Los resultados de ejecutar el programa descrito anteriormente que implementa el algoritmo Semi join sobre Hadoop se ven descritos en las siguientes tablas, en donde la tabla 3 muestra la ejecución sobre 10 empleados y la tabla 4 sobre 10000 empleados como entrada al programa.

Item	**Fase 1**	**Fase 2**	**Fase 3**	**Total Fases**
N° Tareas Map Ejecutadas	2	1	2	5
N° Tareas Reduce Ejecutadas	1	0	0	1
N° Bytes de Entrada Map	137	130	137	404
N° Registros de Salida Map	10	10	10	30
N° Registros de Entrada Reduce	10	0	0	10
N° Grupos de Entrada Reduce	10	0	0	10
N° Registros de Salida Reduce	10	0	0	10
Tiempo Uso de CPU en Map	960 ms	1390 ms	2260 ms	4610 ms
Tiempo Uso de CPU en Reduce	3160 ms	0 ms	0 ms	3160 ms
Tiempo Uso de CPU Total	4120 ms	1390 ms	2260 ms	7710 ms

Tabla 3: Tabla de Resultados ejecución Semi Join en MapReduce sobre 10 empleados.

Item	**Fase 1**	**Fase 2**	**Fase 3**	**Total Fases**
N° Tareas Map Ejecutadas	2	1	2	5
N° Tareas Reduce Ejecutadas	1	0	0	1
N° Bytes de Entrada Map	137	130000	137	130274
N° Registros de Salida Map	10	10000	10000	20010
N° Registros de Entrada Reduce	10	0	0	10
N° Grupos de Entrada Reduce	10	0	0	10
N° Registros de Salida Reduce	10	0	0	10
Tiempo Uso de CPU en Map	1260 ms	79870 ms	5740 ms	86870 ms
Tiempo Uso de CPU en Reduce	3040 ms	0 ms	0 ms	3040 ms
Tiempo Uso de CPU Total	4300 ms	79870 ms	5740 ms	89910 ms

Tabla 4: Tabla de Resultados ejecución Semi Join en MapReduce sobre 10000 empleados.

7.4. Implementación en Pig

La implementación del algoritmo Repartition Join se realizó de forma satisfactoria, mas no la del algoritmo Semi Join, que resulta imposible de implementar a través del lenguaje PigLatin, debido a la abstracción que posee el mismo sobre niveles inferiores de manejo de los datos, es necesario acceder a niveles inferiores de programación para desarrollar el algoritmo Semi Join de forma fiel al algoritmo definido.

7.4.1. Datos Técnicos Codificación

La codificación de los algoritmos se desarrolló en lenguaje de programación Pig Latin, utilizando un plug-in Pig-Pen sobre la plataforma Eclipse versión 3.3.2 Europa sobre el sistema operativo Ubuntu 12.04. El plug-in es capaz de reconocer comandos específicos del lenguaje Pig Latin y de cierta formar orientar al programador, además de permitir crear un enlace con la instalación de Pig presente en la Máquina. Por otra parte todos los trabajos MapReduce se ejecutan a través de Hadoop, así como la fase de agrupación y ordenamiento.

7.4.2. Datos Técnicos Ejecución de Programas

A continuación se desarrollarán los detalles ténicos implicados dentro de la ejecución del programa repartition join side y semi join:

- Sistema Operativo: El sistema operativo es Ubuntu 12.04 de 32 bits de direccionamiento, posee un kernel Linux 3.2.0 publicado el 27 de julio de 2012.
- Máquina: La máquina sobre la que se ejecuta el programa corresponde a un Notebook con procesador Intel Core I5-2430M de 3.00GHz de 64 bits de direccionamiento, posee una memoria principal de 6 gb y una memoria secundaria de 640 gb.
- Pig: Version Pig 0.10, que trabaja a su vez con Hadoop para procesar los datos en tareas MapReduce.
- PigLatin: Lenguaje de programación de Pig.
- Hadoop: La instalación de Hadoop sobre la máquina se encuentra de forma semi distribuida, donde los procesos secundarios de Hadoop simulan como si trabajasen en un cluster, la versión de Hadoop corresponde a la 1.0.3 lanzada el 16 de Mayo del presente año.

- Datos de Entrada: Las dos tablas que se uilizan están codificadas en texto plano UTF8.

7.4.3. Lenguaje Pig Involucrado en la Implementación

Dentro de la codificación de los algoritmos existen palabras reservadas de Pig que ameritan ser desarrolladas en mayor profundidad, algunas de ellas son las siguientes [2]:

- Load: Se utiliza para leer información y almacenarla en el entorno de Pig, debe especificarse una variable destino y una dirección fuente.
- Join: Operador para unir datos en dos o más relaciones.
- Store: Este operador almacena los resultados en un archivo.
- PigStorage: Formato de almacenamiento propio de Pig en donde es posible indicar un separador de campos.
- Using: Se utiliza para describir el formato de almacenamiento a utilizar, por defecto el formato es Pigstorage.
- Left Outer: Operación interna del operador Join que permite que la tabla que se encuentra a la izquierda de la sentencia posea todos sus campos presentes en el resultado final del join.

7.4.4. Implementación Repartition Join

1. **Código Pig Involucrado**

   El código indicado en la figura 33 corresponde a todas las instrucciones necesarias en PigLatin para realizar una operación join sobre dos tablas utilizando el algoritmo repartition join, que como se ve reflejado en las primeras líneas de la figura, se cargan ambas tablas en dos variables utilizando un formato especial de Pig (líneas 1 y 2), identificando cada campo de la tupla, luego se realiza el producto cruzado según la clave indicada en la línea 3 del código, en donde se realiza un Join hacia la izquierda, es decir todos los valores de la tabla Empleado deben estar presentes en el resultado de la operación. Finalmente la salida de esta operación se almacena en el directorio especificado en la última línea de código, que corresponde a la carpeta "*salidapig*".

```
1 EMPLEADOS = load 'in1/Empleados.txt' using PigStorage(',') as (rut, codigo_depto);
2 DEPARTAMENTOS = load 'in1/Depto.txt' using PigStorage(',') as (codigo_depto, nombre_depto);
3 SUMA = join EMPLEADOS by codigo_depto left outer, DEPARTAMENTOS by codigo_depto;
4 STORE SUMA INTO '/salida_pig';
```

Figura 33: Código Pig Repartition Join.

2. **Resultados Obtenidos**

   Los resultados luego de ejecutar el programa descrito en la sección anterior se encuentran disponibles en la tabla 5 se encuentran los resultados de la ejecución sobre diez empleados y en la tabla 6 sobre 10000 empleados.

Item	**Valor**
N° Tareas Map Ejecutadas	2
N° Tareas Reduce Ejecutadas	1
N° Bytes de Entrada Map	513
N° Registros de Salida Map	20
N° Registros de Entrada Reduce	20
N° Grupos de Entrada Reduce	10
N° Registros de Salida Reduce	10
Tiempo Uso de CPU en Map	880 ms
Tiempo Uso de CPU en Reduce	2410 ms
Tiempo Uso de CPU Total	3290 ms

Tabla 5: Tabla de Resultados ejecución Repartition Join con Pig sobre 10 empleados.

Item	Valor
N° Tareas Map Ejecutadas	2
N° Tareas Reduce Ejecutadas	1
N° Bytes de Entrada Map	250263
N° Registros de Salida Map	10010
N° Registros de Entrada Reduce	10010
N° Grupos de Entrada Reduce	10
N° Registros de Salida Reduce	10010
Tiempo Uso de CPU en Map	2820 ms
Tiempo Uso de CPU en Reduce	3570 ms
Tiempo Uso de CPU Total	6390 ms

Tabla 6: Tabla de Resultados ejecución Repartition Join con Pig sobre 10000 empleados.

7.5. Implementación en MongoDB

La implementación del algoritmo Join en MongoDB resultó ser inviable, tanto el algoritmo Repartition Join, como el algoritmo Semi Join, ésto debido a la naturaleza misma en que MongoDB mantiene la estructura de los datos. A diferencia de una base de datos Relacional (sql), MongoDB mantiene cada una de sus tablas en colecciones y cada una de las filas de esas tablas (sql) se denominan documentos [8], los cuales no necesariamente deben poseer una estructura rígida en cuanto a los tipos de datos que los componen, por ésta razón es que una base de datos MongoDB no necesita tener datos relacionados de forma separada, todos pueden ser agrupados en un solo Documento, por lo tanto la operación Join no resulta necesaria. Ahora bien en la implementación desarrollada se crearon dos colecciones con un formato idéntico al formato utilizado en las tablas de Hadoop o Pig, sin embargo la operación Join fue imposible de realizar sobre empleados que pertenecían al mismo departamento, ésta situación se explicará en mayor detalle más adelante.

Para poder realizar el trabajo MapReduce sobre estas dos colecciones fue necesario crear dos funciones Map, una por cada colección, y una función reduce según se puede apreciar en la siguiente figura 34, 35 y 36, en total es necesario llevar a cabo dos trabajos MapReduce, que comparten la misma función Reduce, en donde cada uno de éstos trabajos realiza una salida final creando una nueva colección denominada *Join*, en dónde por defecto en la ejecución del segundo trabajo MapReduce los datos serán sobre-escritos

sobre los datos que posean la misma clave, sin embargo MongoDB presenta problemas en el caso en que se repita un código de departamento, ya que éstos son tratados como claves primarias dentro de la nueva colección creada, en éste caso el Join ya es imposible de realizar, una alternativa válida podría ser el poseer en sólo una colección las dos tablas SQL Empleado y Departamento ya que sólo sería necesario un único trabajo MapReduce y de alguna forma adaptar el formato de salida que poseen actualmente los datos, pero siendo fiel a los datos de entrada utilizados en todas las implementaciones anteriores éste trabajo es imposible de realizar de forma satisfactoria.

```
map_empleado = function(){

var codigo_depto = this.codigo_depto;

var valores = {rut_empleado:this.rut_empleado, tag: "emp"};

emit(codigo_depto, valores);
}
```

Figura 34: Función Map que trabaja sobre la Colección Empleado.

```
map_depto = function(){

var codigo_depto = this.codigo_depto;

var valores ={nombre_depto: this.nombre_depto, tag:"depto"};

emit(codigo_depto,valores);
}
```

Figura 35: Función Map que trabaja sobre la Colección Departamento.

```
reduce_final = function(k , valores){

var resultado = { rut:null, nombre:null };

   valores.forEach(function(valores){

     if(valores.tag == "emp"){
       resultado.rut = valores.rut_empleado;
      }else{
        resultado.nombre = valores.nombre_depto;
      }
   });

return resultado;

}
```

Figura 36: Función Reduce Única.

7.6. Generadores de Datos

Los datos implicados dentro de la ejecución de los algoritmos desarrollados en Hadoop y en Pig son generados por un programa desarrollado completamente en Java, mientras que en el caso de MongoDB son generados a través de procedimientos almacenados dentro del motor. A continuación una breve explicación de éstos programas:

7.6.1. Generador de Datos para Hadoop y Pig

Consta de dos clases, una de ellas se encarga de generar aleatoriamente las combinaciones de empleado con su departamento respectivo, esto puede verse reflejado en la figura 37, donde el método *cargar empleados* se encarga de ésta tarea, los archivos txt son enviados finalmente a un directorio declarado en el inicio del método, *tablas entrada*.

```
public class cargar_empleados_y_deptos {
        private String
nombre_deptos[]={"Informatica","Comercial","Finanzas","RRHH","Logistica","Contabilidad","
Comex","NBusiness","Contratista","Operativa"};
      private String
claves_deptos[]={"D01","D02","D03","D04","D05","D06","D07","D08","D09","D10"};
      private String buffer=null;
      private int subindice_general =0;
      private FileWriter fichero = null;
      private PrintWriter pw = null;

public int cargar_empleados(int numero_empleados_a_generar){
int i_rut = 0;//para iniciar a iterar los rut
int rut_inicial = 16200545; //valor por defecto
subindice_general = 0;
Random rnd = new Random();
try
{
    fichero = new FileWriter("/home/spas/Documentos/tablas_entrada/Empleados.txt");
   pw = new PrintWriter(fichero);

   while (i_rut<numero_empleados_a_generar){
         subindice_general = (int) Math.floor(Math.random()*10);//numero del 0 al 9 al
azar

         String rut = (Integer.toString((rut_inicial+i_rut)).trim()); //valor del rut a
string sin espacios

         buffer= (rut+","+claves_deptos[subindice_general]).trim();  //asignar el rut y
el codigo del departamento a un buffer

         pw.println(buffer);   //se escribe en el archivo
           i_rut++;
   }
} catch (Exception e) {
}finally{

      try{
        if( null != fichero ){   //aqui se cierra el fichero
           fichero.close();
        }
     }catch (Exception e1){
     }} return 1;}
```

Figura 37: Clase Generadora de Datos.

7.6.2. Generadores de Datos en MongoDB

Para generar las tablas, en el caso de MongoDB consisten en Colecciones, es necesario emplear procedimientos almacenados para realizar ésta tarea, los procedimientos almacenados en MongoDB son relativamente parecidos a los de una base de datos relacional, en total son dos procedimientos; uno encargado de crear la colección de departamentos y otro encargado de generar la colección de empleados, éste último resulta ser el más importante y es el que al igual que el programa Generador de Hadoop debe realizar combinaciones aleatorios entre el Rut de un empleado y el código de un departamento, el resultado final de éstos procedimientos es una colección nueva dentro de MongoDB. La figura 38 muestra el procedimiento encargado de crear la colección Empleado.

```
cargar_empleados = function (numero_empleados){

var rut = 16200600;
var i = 0;
var depto_aleatorio = 0;
var cod_depto = null;
var deptos=["D01","D02","D03","D04","D05","D06","D07","D08","D09","D10"];

while(i < numero_empleados){
i++;
rut = rut +1;
depto_aleatorio = Math.floor(Math.random()*10);//generador de numero aleatorios
cod_depto = deptos[depto_aleatorio]; //seleccionar un codigo del arreglo anterior

documento = { codigo_depto : cod_depto , rut_empleado : rut };
db.empleado.save(documento); //creación de la coleccion

}
return true;
}
```

Figura 38: Procedimiento Almacenado en MongoDB, Generador de Empleados.

8. Análisis de Resultados

Finalmente luego de realizar las implementaciones del algoritmo Join, utilizando dos algoritmos específicos, que en este caso corresponden a Repartition Join y Semi Join, se puede establecer tal y como se refleja en la tabla 7 que no en todas las plataformas planteadas fue posible desarrollar estas implementaciones. Sin embargo existen casos en que es posible realizar alguna clase de comparaciones de desempeño, por ejemplo entre el mismo algoritmo Repartition Join implementado en Hadoop y en Pig, como lo ilustra la tabla 8.

Otra importante comparación se establece en la tabla 9 en donde es posible visualizar las diferencias entre diferentes algoritmos desarrollados en la misma plataforma y que además trabajan con los mismo datos de entrada, por supuesto cada uno de ellos se enfoca en mejorar ciertas características que podrían llevar a marcar una diferencia en cuanto a rendimiento.

Plataformas	**Repartition Join**	**Semi Join**
HADOOP	Implementado	Implementado
PIG	Implementado	No es Posible
MONGO DB	No es Posible	No es Posible

Tabla 7: Tabla Resúmen Implementación Algoritmo Join.

Item	**Hadoop**	**Pig**
N° Tareas Map Ejecutadas	3	2
N° Tareas Reduce Ejecutadas	1	1
N° Registros de Salida Map	20	20
N° Registros de Entrada Reduce	20	20
N° Registros de Salida Reduce	10	10
Tiempo Uso de CPU en Map	1070 ms	880 ms
Tiempo Uso de CPU en Reduce	3860 ms	2410 ms
Tiempo Uso de CPU Total	4930 ms	3290 ms

Tabla 8: Tabla comparativa Repartition Join sobre 10 empleados en Hadoop y Pig.

Existe una gran diferencia en tiempos de ejecución entre el algoritmo desarrollado en Hadoop y en Pig, esto puede deberse a los ciclos que existen en el desarrollo del Repartition Join implementado en Hadoop, los cuales son

conocidos por su impacto en tiempos de ejecución, de alguna forma se podría optimizar la codificación desarrollada en Java sobre este algoritmo con el fin de mejorar los tiempos, por ahora Pig parece llevar la delantera en tiempo de ejecución, sin embargo es posible realizar un número alto de modificaciones o personalizaciones al momento de codificar la operación Join en Java a diferencia de Pig en dónde la operación está básicamente pre-definida.

Item	**Repartition Join**	**Semi Join**[10]
N° Tareas Map Ejecutadas	3	2
N° Tareas Reduce Ejecutadas	1	0
N° Registros de Salida Map	20	10
N° Registros de Entrada Reduce	20	0
N° Registros de Salida Reduce	10	0
Tiempo Uso de CPU en Map	1070 ms	2260 ms
Tiempo Uso de CPU en Reduce	3860 ms	0 ms
Tiempo Uso de CPU Total	4930 ms	2260 ms

Tabla 9: Tabla comparativa de Repartition Join sobre 10 empleados en Hadoop y Pig.

El caso presentado en la tabla 9 indica que el algoritmo Semi Join lleva la delantera en tiempo de ejecución sobre el Repartition Join, sin embargo, ésto es considerando la fase I y la fase II del Semi Join como pre-procesos, claramente en el caso que se consideren las fases anteriores y se sumen los tiempos de ejecución Repartition Join llevaría la delantera, se ha procedido a considerar las fases I y II como pre-procesos debido a que realizan un trabajo que es completamente descuidado en el algoritmo Repartition Join y que apuntan a disminuir el tiempo de ejecución por medio de la disminución de los datos, que son enviados hacia la etapa Reduce y que por lo tanto son ordenados y agrupados por Hadoop, este proceso consume mucho tiempo y es en este caso algo relativamente innecesario.

[10]Semi Join Fase III: En éste caso los datos del Semi Join se basan en los resultados de la Fase III y última de éste algoritmo, de modo que las fases I y II son consideradas como un pre-procesamiento para establecer una comparación con el trabajo realizado por el algoritmo Repartition Join.

9. Conclusiones

Luego de estudiar el algoritmo Join a implementar y comenzar a codificar el mismo, gracias al conocimiento previo sobre el funcionamiento del modelo MapReduce, es posible decir que resulta claro que MapReduce no está diseñado para realizar este tipo de operaciones en forma nativa, pero aun así es posible establecer ciertos métodos que permiten el correcto funcionamiento de éste modelo sobre el algoritmo join. Debido a que la técnica para adaptar MapReduce hacia el algoritmo depende de cada programador, teniendo el mismo caso de estudio, los resultados obtenidos deberían ser los mismos, pero no así el rendimiento de las operaciones Map y Reduce que podrían poseer procedimientos poco eficientes.

Actualmente existen métodos específicos insertos en el modelo MapReduce que podrían de alguna forma facilitar el trabajo en la resolución de operaciones Join, pero aun así dependen en gran forma del usuario que se encuentre detrás del desarrollo del programa.

La codificación en lenguaje de programación java (Hadoop) posee tanto factores negativos como positivos, ya que al tener que definir todos los procedimientos para trabajar sobre el algoritmo join es posible establecer ciertas mejoras a funciones que podrían estar ya definidas, por otra parte puede resultar en una pérdida de tiempo enorme el tener que definir todo dentro del programa, tiempo que se ve reflejado en horas de programación y por lo tanto en dinero por horas hombre.

En cuanto a los resultados obtenidos en la implementación de Pig, estos eran de esperarse, debido a la alta abstracción que posee en su lenguaje de programación PigLatin sobre niveles inferiores de programación. El algoritmo repartition join presenta una estructura similar a la estructura de la operación join establecida en pig, en cambio la implementación del algoritmo semi join resultó ser inviable debido a lo mencionado anteriormente. En cuanto a la comparación que pudo establecerse entre el algoritmo repartition join implementado tanto en Hadoop como en Pig, es claro que estos resultados son influenciados directamente por la experticia del programador y su estilo de programación.

Por otra parte, ya se estableció que la implementación de ambos algoritmos en la plataforma MongoDB resultó ser completamente inviable, de hecho la operación join básica lo es, pero lo importante dentro de todo esto es el destacar las diferencias de una base de datos SQL con una NoSQL (MongoDB) en cuanto a su arquitectura y en general la forma de tratar y organizar los datos, se concluye que MongoDB no necesita realizar una operación Join, o por lo menos no de la misma forma que una base de datos

SQL. En la presente investigación quedó demostrado con datos obtenidos sobre una implementación, luego de un análisis que es la misma estructura de MongoDB la que impide realizar esta operación de forma satisfactoria.

La implementación tanto de Repartition Join como de Semi Join sobre diferentes plataformas demostraron lo complejo que puede resultar muchas veces realizar una operación que para muchos es algo trivial, ninguno de los resultados obtenidos habrían sido posibles sino fuera por MapReduce, que es la columna vertebral que une las diferentes codificaciones en los diferentes lenguajes, y que permitió realizar cada ejecución sobre un número considerable de datos de entrada en un tiempo mínimo de trabajo, estas ejecuciones solamente fueron limitadas por los recursos de la máquina en la cual se encuentran insertas. MapReduce parece ser lo que promete según sus precursores, y mucho más, esto quedó demostrado en la presente investigación gracias a las implementaciones y ejecuciones en tiempo real desarrolladas.

Referencias

[1] Apache. Hadoop. http://hadoop.apache.org/docs/r0.20.0/api/org/apache/hadoop/, September 2012. [Online; último ingreso 20-Septiembre-2012].

[2] Apache. Pig. http://pig.apache.org/docs/r0.10.0/start.html#req, November 2012. [Online; último ingreso 21-Noviembre-2012].

[3] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce . volume 1. ACM Sigmod, June 2010.

[4] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. volume 1, pages 139–142. Operating Systems Design and Implementation (OSDI), December 2004.

[5] Jefrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. volume 1, pages 72–76. Communications of the ACM, January 2010.

[6] Tony Hannan. MongoDB. http://www.mongodb.org/display/DOCS/Introduction, September 2011. [Online; último ingreso 19-Junio-2012].

[7] Dwight Merriman. MongoDB. http://www.mongodb.org/display/DOCS/Use+Cases, December 2011. [Online; último ingreso 19-Junio-2012].

[8] MongoDB. MongoDB. http://docs.mongodb.org/manual/, November 2012. [Online; último ingreso 21-Noviembre-2012].

[9] Cristopher Olston, Utkarsh Srivastava, Ben Reed, et al. Pig talk. page 168, June 2008.

[10] Tom White. *Hadoop, The Definitive Guide*, chapter 3,6,11, pages 15,62–67,168–178,321. 2 edition, oct 2010.