Universitat Autònoma
de Barcelona

Escola d'Enginyeria

Departament d'Arquitectura de Computadors
i Sistemes Operatius

PLANIFICACIÓN DE TRABAJOS EN CLUSTERS HADOOP COMPARTIDOS

Tesis doctoral presentada por **Aprigio Augusto Lopes Bezerra** para optar al grado de Doctor por la Universitat Autònoma de Barcelona, bajo la dirección del Dr. Porfidio Hernández Budé.

Bellaterra, Septiembre de 2014

PLANIFICACIÓN DE TRABAJOS EN CLUSTERS HADOOP COMPARTIDOS

Tesis doctoral presentada por **Aprigio Augusto Lopes Bezerra** para optar al grado de Doctor por la Universitat Autònoma de Barcelona. Realizado en el Departament d'Arquitectura de Computadors i Sistemes Operatius de la Escola d'Enginyeria de la Universitat Autònoma de Barcelona, dentro del programa "Computación de Altas Prestaciones bajo la dirección del **Dr. Porfidio Hernández Budé**.

Bellaterra, Septiembre de 2014

Dr. Porfidio Hernández Budé
Director

Aprigio Augusto Lopes Bezerra
Autor

“No hay ninguna razón por la que no se pueda enseñar a un hombre a pensar.”

Burrus Frederic Skinner

Agradecimientos

Deseo expresar mi agradecimiento al Dr. Porfidio Hernández Budé, por su paciencia, comprensión y constante disponibilidad para la crítica y valoración del trabajo realizado. Así como al Dr. Juan Carlos Moure y al Dr. Antoni Espinosa por sus incalculables aportes.

A Eduardo Argollo por su confianza y ayuda a mi llegada en Barcelona.

A mis compañeros de CAOS, especialmente a Sandra Mendez, Eduardo Cabrera, João Gramacho, Carlos Nuñes, Cesár Allande, Tharso de Souza, Andrea Martínez, Marcela Castro, Hugo Meyer, Pilar Gómez, Francisco Borges y Gemma Sanjuan.

A Daniel Ruiz, Javier Navarro y Gemma Roqué por el atendimiento rápido y eficiente de todas solicitaciones.

Al Governo do Estado da Bahia, a Universidade Estadual de Santa Cruz y a mis compañeros de la carrera de Ciência da Computação por todo el soporte.

Y muy especialmente, a mi esposa e hijos por la comprensión de mí prolongada ausencia.

Abstract

Industry and scientists have sought alternatives to process effectively the large volume of data generated in different areas of knowledge. MapReduce is presented as a viable alternative for the processing of data intensive application. Input files are broken into smaller blocks. So they are distributed and stored in the nodes where they will be processed. Hadoop clusters have been used to execute MapReduce applications. The Hadoop framework automatically performs the division and distribution of the input files, the division of a job into Map and Reduce tasks, the scheduling tasks among the nodes, the failures control of nodes; and manages the need for communication between nodes in the cluster. However, some MapReduce applications have a set of features that do not allow them to benefit fully from the default Hadoop job scheduling policies. Input files shared between multiple jobs and applications with large volumes of intermediate data are the characteristics of the applications we handle in our research. The objective of our work is to improve execution efficiency in two ways: On a macro level (job scheduler level), we group the jobs that share the same input files and process them in batch. Then we store shared input files and intermediate data on a RAMDISK during batch processing. On a micro level (task scheduler level) tasks of different jobs processed in the same batch that handle the same data blocks are grouped to be executed on the same node where the block was allocated.

Resumen

La industria y los científicos han buscado alternativas para procesar con eficacia el gran volumen de datos que se generan en diferentes áreas del conocimiento. MapReduce se presenta como una alternativa viable para el procesamiento de aplicaciones intensivas de datos. Los archivos de entrada se dividen en bloques más pequeños. Posteriormente, se distribuyen y se almacenan en los nodos donde serán procesados. Entornos Hadoop han sido utilizados para ejecutar aplicaciones MapReduce. Hadoop realiza automáticamente la división y distribución de los archivos de entrada, la división del trabajo en tareas Map y Reduce, la planificación de tareas entre los nodos, el control de fallos de nodos; y gestiona la necesidad de comunicación entre los nodos del cluster. Sin embargo, algunas aplicaciones MapReduce tienen un conjunto de características que no permiten que se beneficien plenamente de las políticas de planificación de tareas construídas para Hadoop. Los archivos de entrada compartidos entre múltiples trabajos y aplicaciones con grandes volúmenes de datos intermedios son las características de las aplicaciones que manejamos en nuestra investigación. El objetivo de nuestro trabajo es implementar una nueva política de planificación de trabajos que mejore el tiempo de makespan de lotes de trabajos Hadoop de dos maneras: en un nivel macro (nivel de planificación de trabajos), agrupar los trabajos que comparten los mismos archivos de entrada y procesarlos en lote; y en un nivel micro (nivel de planificación de tareas) las tareas de los diferentes trabajos procesados en el mismo lote, que manejan los mismos bloques de datos, se agrupan para ser ejecutas en el mismo nodo donde se asignó el bloque. La política de planificación de trabajos almacena los archivos compartidos de entrada y los datos intermedios en una RAMDISK, durante el procesamiento de cada lote.

Resum

La indústria i els científics han buscat alternatives per processar amb eficàcia el gran volum de dades que es generen en diferents àrees del coneixement. MapReduce es presenta com una alternativa viable per al processament d'aplicacions intensives de dades. Els arxius d'entrada es divideixen en blocs més petits. Posteriorment, es distribueixen i s'emmagatzemen en els nodes on seran processats. Entorns Hadoop han estat utilitzats per a executar aplicacions MapReduce. Hadoop realitza automàticament la divisió i distribució dels arxius d'entrada, la divisió del treball en tasques Map i Redueix, la planificació de tasques entre els nodes, el control de fallades de nodes; i gestiona la necessitat de comunicació entre els nodes del clúster. No obstant això, algunes aplicacions MapReduce tenen un conjunt de característiques que no permeten que es beneficiïn plenament de les polítiques de planificació de tasques construïdes per Hadoop. Els arxius d'entrada compartits entre múltiples treballs i aplicacions amb grans volums de dades intermedis són les característiques de les aplicacions que fem servir en la nostra investigació. L'objectiu del nostre treball és implementar una nova política de planificació de treballs que millori el temps de makespan de lots de treballs Hadoop de dues maneres: en un nivell macro (nivell de planificació de treballs), agrupar els treballs que comparteixen els mateixos arxius de entrada i processar en lot; i en un nivell micro (nivell de planificació de tasques) les tasques dels diferents treballs processats en el mateix lot, que manegen els mateixos blocs de dades, s'agrupen per ser executes en el mateix node on es va assignar el bloc. La política de planificació de treballs emmagatzema els arxius compartits d'entrada i les dades intermedis en una RAM durant el processament de cada lot.

Contenido

Capítulo 1 -INTRODUCCIÓN 1
1.1 Hadoop 5
1.1.1 Hadoop Distributed File System 5
1.1.2 Gestión de trabajos en *cluster* Hadoop 7
1.2 Aplicaciones Hadoop con características especiales 8
1.3 Revisión de la literatura 13
1.4 Objetivos 15
1.4.1 Objetivo principal 17
1.4.2 Objetivos específicos 17
1.5 Organización de la Memoria 18
Capítulo 2 -PLANIFICACIÓN DE TRABAJOS 21
2.1 Introducción a los entornos paralelos compartidos 21
2.2 Planificación de trabajos en clusters Hadoop 24
2.3 Planificadores de Trabajos Hadoop 27
2.4 Sintonización de Parámetros 33
Capítulo 3 -POLÍTICA DE PLANIFICACIÓN DE TRABAJOS PROPUESTA 39
3.1 Archivos de Entrada Compartidos 39
3.2 Gestión de datos de entrada y salida en Hadoop 42
3.3 Gestión de datos intermedios en Hadoop 45
3.4 SHARED INPUT POLICY 47
3.5 RAMDISK para gestión de datos intermedios 50
3.6 Implementación 53
Capítulo 4 -EXPERIMENTACÍON REALIZADA Y RESULTADOS 57
4.1 Entorno de experimentación 57
4.2 Aplicaciones 59
4.2.1 Aplicación MrMAQ 60
4.2.2 Intel Benchmark 73
4.3 Shared Input Policy 74
4.3.1 Localidad de las tareas Map 74
4.3.2 Monitorización de los Recursos 80

4.3.3 Disco Local y RAMDISK 82
4.3.4 Shared Input Policy con RAMDISK 85
Capítulo 5 -CONCLUSIONES 89
5.1 Líneas Abiertas 92
Referencias 93

Capítulo 1 -

INTRODUCCIÓN

Los recientes avances en las diferentes disciplinas científicas e ingeniería están generando grandes cantidades de datos que necesitan ser almacenados y analizados de manera eficiente. El creciente volumen de datos digitalizados y procesados por aplicaciones puede ser observado en simulación, sensores de datos, procesamiento de imágenes médicas y en prácticamente todas las áreas de conocimiento. La estimación de datos generados a partir de operaciones de negocios diarios realizadas en los Estados Unidos es de 45 TBytes, y los programas genómicos y proteómicos generan 100 TBytes de nuevos datos a cada día [1]; Google procesaba en 2008 20 PBytes de dados en un solo día [2]; el experimento Large Handron Collider (LHC) del Center for European Nuclear Research (CERN) genera 15 PBytes de datos para que sean procesados cada año [3].

Estos enormes volúmenes de datos deben ser procesados en un tiempo satisfactorio para posibilitar la toma de decisiones y nuevos descubrimientos. Surge así un grupo de aplicaciones con nuevos tipos de requerimientos. Aplicaciones donde la necesidad de procesamiento no es el requerimiento principal, sino el manejo en un tiempo razonable de ingentes cantidades de datos. Almacenar, transferir y procesar grandes volúmenes de datos pasan a ser factores determinantes para los sistemas que manejan estos tipos de aplicaciones, y un reto para la computación de altas prestaciones. Las aplicaciones que manejan grandes volúmenes de datos son conocidas como aplicaciones intensivas en datos y los sistemas donde se las ejecutan son conocidos como sistemas Data-Intensive Computing (DIC) o, en traducción libre, sistemas de Computación Intensiva de Datos.

Las aplicaciones informáticas intensivas en datos pueden beneficiarse de la utilización de sistemas paralelos y distribuidos de computación tanto para mejorar el rendimiento como la calidad de la gestión y análisis de datos. Sin embargo, necesitan mejores diseños de sistemas y administradores de recursos porque introducen nuevas dimensiones, como la representación y almacenamiento de datos, el acceso

descentralizado a los datos, múltiples estrategias de paralelismo y la heterogeneidad de entornos[4].

La computación de altas prestaciones tradicional está implementada mediante arquitecturas paralelas del tipo disco compartido (*shared disk*), con nodos de cómputo y nodos de datos diferenciados. Los nodos de cómputo tienen un mínimo de almacenamiento local, y están conectados entre ellos y entre los nodos de datos por redes de comunicación de altas prestaciones. Aunque los avances tecnológicos permitan redes de conexión cada vez más rápidas, el gran volumen de datos que necesita ser transferido a través de la red por las aplicaciones intensivas de datos hace que el uso de estas arquitecturas no sea adecuado.

También desde el punto de vista de la implementación, si evaluamos los entornos de programación paralelas tradicionales, los diseñadores de aplicaciones tienen una lista de responsabilidades a las que hacer frente a:

- Decidir la granularidad por la descomposición del problema en módulos;
- Asignar tareas en cada nodo del sistema;
- Coordinar la sincronización de los diferentes procesos en ejecución;
- Definir políticas de tolerancia a fallos.

Algunas de estas tareas necesitan de un conocimiento detallado de los aspectos *hardware* del sistema informático en uso. Por lo tanto, las aplicaciones paralelas están por lo general, estrechamente acopladas al hardware en el que se ejecutan; lo que hace que sea difícil portarlas a otras plataformas. Los modelos de programación paralela como MPI y OpenMP, proporcionan un cierto nivel de abstracción sobre las operaciones de comunicación y sincronización. Sin embargo, los programadores aún necesitan abordar muchas de las decisiones de diseño mencionadas anteriormente. Además, los modelos tradicionales de programación paralela se centran en los problemas relacionados a aplicaciones intensivas en CPU y no proporcionan un soporte adecuado para el procesamiento de gran cantidad de datos.

Como solución a estas necesidades, Google propuso utilizar el paradigma MapReduce [2] centrado en los problemas del procesamiento de grandes volúmenes de datos de entrada.

Las aplicaciones desarrolladas bajo el paradigma MapReduce, son gestionadas de forma automática, y pueden ser ejecutadas en un *cluster* de ordenadores de manera sencilla. En tiempo de ejecución, el *framework* que implementa MapReduce proporciona soluciones para cada una de las necesidades de las aplicaciones paralelas. Por ejemplo, se aplica una política predefinida para establecer la partición de los datos de entrada, y la división del trabajo en tareas, una política de planificación de las tareas en los diferentes nodos de computación, una política de gestión de tolerancia a fallos, y un patrón de comunicación transparente entre las diferentes tareas de la aplicación. Con este modelo, los programadores sin mucha experiencia en entornos distribuidos pueden efectivamente utilizar un sistema potencialmente grande.

MapReduce es un modelo de programación que permite el procesamiento y la gestión de grandes volúmenes de datos. Los archivos de entrada son divididos en bloques más pequeños y son distribuidos entre los nodos de cómputo. El objetivo principal es acercar los datos de los nodos donde serán procesados y disminuir el volumen de datos transferidos por la red; que es un gran problema para los sistemas intensivos en datos. Llamamos a este tipo de arquitectura *shared-nothing*, donde un nodo tiene doble funcionalidad: cómputo y almacenamiento.

Una vez que el *framework* que implementa MapReduce se encarga de las tareas complejas de gestión del entorno, el programador puede centrarse en el desarrollo de su código. La implementación de una aplicación MapReduce debe especificar la construcción de dos funciones desarrolladas por el programador: Map y Reduce. El programador define una estructura de datos del tipo <clave, valor> y la función Map es aplicada, de forma paralela y distribuida, sobre cada bloque en que los datos de entrada hayan sido divididos. El objetivo de la función Map es recorrer cada bloque de datos buscando las coincidencias con la <clave> definida por el programador. Para cada coincidencia encontrada por la función Map es generada una tupla, también en el formato <clave, valor> que es almacenada de forma temporal. La función Map genera una lista de tuplas para cada bloque de datos en que la entrada haya sido dividida. Estas listas son entonces agrupadas y procesadas a continuación por la función Reduce.

La función Reduce actúa sobre los datos temporales que han sido generados por la función Map, realizando el procesamiento de los valores que poseen la misma <clave>. La salida de la función Reduce también es una lista de <clave, valor> representando el resultado final de la aplicación.

Para ilustrar el paradigma MapReduce usamos una aplicación de tipo WordCount que recibe como entrada un fichero de datos y genera como salida el número de ocurrencias de cada palabra encontrada, como muestra la Figura 1.1. El archivo de entrada (*input*) es dividido en bloques más pequeños (*splits*) que son distribuidos entre los nodos de procesamiento. Cada bloque es procesado por una instancia de Map. En este ejemplo de WordCount, la función Map recorre el bloque y para cada palabra encontrada (*word*), genera como salida una tupla (*word*, 1). La salida de Map es almacenada en el sistema de archivos local del nodo. A continuación empieza la fase *Shuffle* que es la fase donde una copia del archivo con la salida de Map es distribuida entre los nodos que procesarán la función Reduce. Después de recibir las tuplas (*word*, 1) de todos los nodos donde han sido procesados por Map, la función Reduce suma el valor 1 para cada palabra (*word*) de las tuplas recibidas.

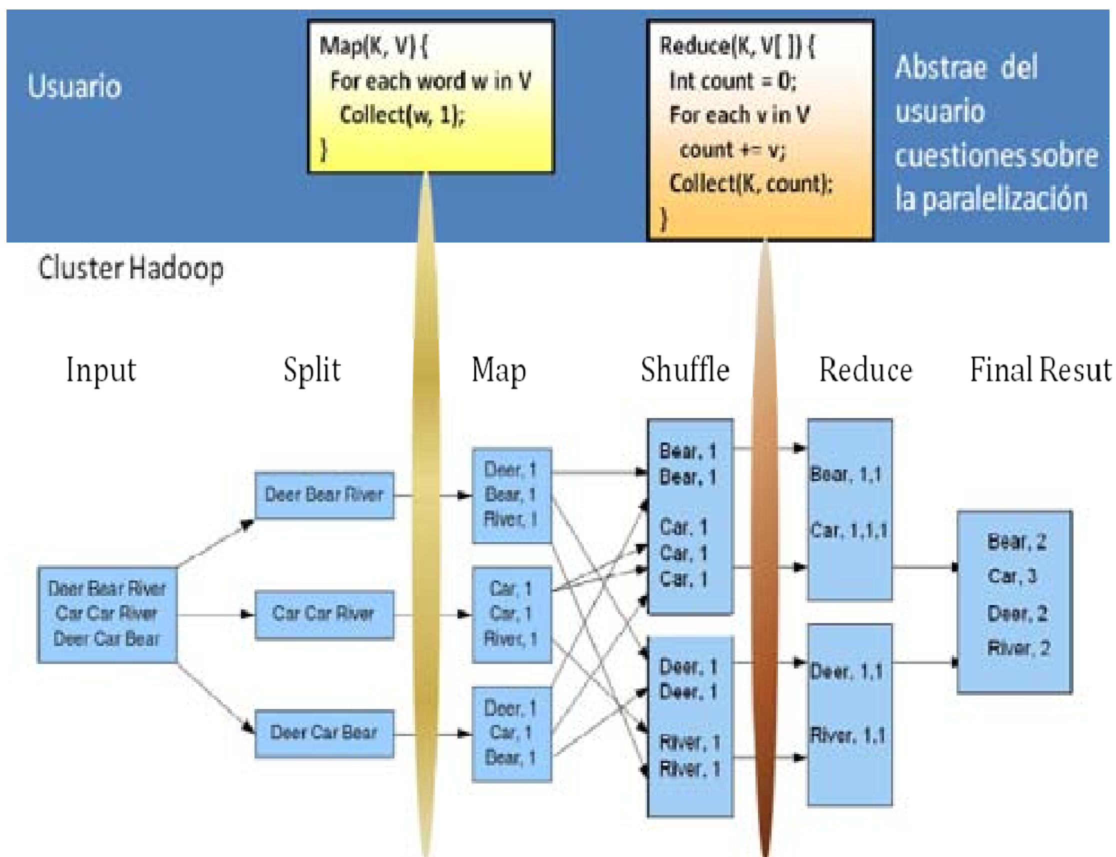

Figura 1.1- Aplicación WordCount en MapReduce.

El uso de MapReduce para aplicaciones intensivas en datos está atrayendo cada vez más industria y mundo académico. Actualmente hay una gran variedad de implementaciones MapReduce. Hadoop es un *framework* configurable, desarrollado en el proyecto Apache y que implementa MapReduce inspirado en la propuesta de Google [2] y [5]. Se trata de un sistema de código abierto, e implementado en Java. Otras implementaciones del paradigma MapReduce han aparecido en la literatura para diferentes arquitecturas como la Cell BE [6], para las GPU [7], [8] y para los procesadores multi/*manycore* [9], [10] y [11].

1.1 Hadoop

Hadoop es un *framework* que implementa MapReduce desarrollado como un sistema de código abierto, e implementado en lenguaje de programación Java. Está constituido de dos subsistemas principales: HDFS (Hadoop Distributed File System) que es el sistema de archivos distribuido de Hadoop, y el planificador de trabajos Hadoop que es el sistema que ejecuta y gestiona las tareas que forman un trabajo del tipo Hadoop. Ambos están diseñados utilizando una arquitectura maestro/trabajador (*master*/*worker*), donde un maestro coordina las tareas realizadas por los trabajadores.

A continuación, presentamos los puntos principales del sistema de archivos distribuidos de Hadoop y cómo el planificador de trabajos Hadoop planifica y gestiona la ejecución de un trabajo en su entorno.

1.1.1 Hadoop Distributed File System

HDFS es el sistema de archivos distribuidos implementado por Hadoop. Se monta sobre el sistema de archivos de cada máquina del *cluster*, y está construido para permitir el soporte a diversos sistemas de ficheros. La gestión de los datos se realiza a través de dos demonios: *NameNode* (en el nodo *master*) y *DataNode* (en cada nodo *worker*). Como se aprecia en la Figura 1.2.

Cuando se carga un archivo en el sistema, HDFS divide el archivo en bloques menores con tamaño modificable por el usuario (por defecto cada bloque tiene el tamaño de 64 MB). A continuación, los bloques son distribuidos entre los nodos de cómputo. El *NameNode* mantiene una tabla para establecer una correspondencia entre cada bloque del fichero y su ubicación en el correspondiente nodo del *cluster*. La Figura 1.2 muestra como HDFS hace la distribución de los bloques en el *cluster* utilizando la política de distribución por defecto, que es aleatoria; y como mantiene el mapeo de los mismos en *NameNode*.

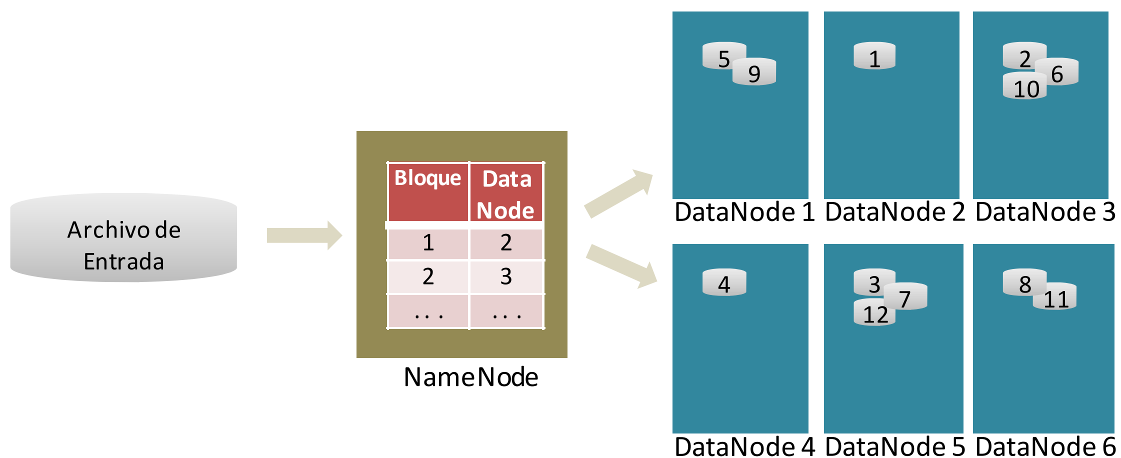

Figura 1.2- Mapeo y distribución de bloques de entrada en HDFS.

NameNode gestiona lo que llamamos *namespace*, es decir, la estructura jerárquica de directorios y los metadatos para todos los archivos y directorios en HDFS. Mientras *DataNode* realiza correctamente las operaciones de lectura, escritura y modificación de datos en cada nodo. Los *DataNodes* también mantienen el *NameNode* actualizado sobre sus estados. Durante el proceso de inicialización, cada *DataNode* informa *NameNode* sobre cuáles bloques mantiene almacenados. A continuación, los *DataNodes* siguen notificando *NameNode* sobre los cambios que se producen en los bloques que mantiene y reciben instrucciones del *NameNode* para crear, mover o borrar los bloques que gestiona.

NameNode mantiene todo el *namespace* en la memoria. Cuando un cliente necesita acceder a un archivo, primero consulta al *NameNode* sobre la ubicación de los bloques que componen el archivo y luego empieza a interactuar directamente con los *DataNodes* donde se almacenan los bloques. El acceso a los datos se realiza a través de *streaming* mediante el modelo de coherencia conocido como *write-once*, *read-many-*

times. Este modelo de acceso a datos prioriza el *throughput* en lugar de la latencia porque en la mayoría de los casos, los análisis ocurren sobre todos los datos del bloque o sobre la mayor parte de ellos, en lugar de utilizar datos con acceso aleatorio. Así el tiempo necesario para leer todo el bloque es más importante que la latencia para leer el primer registro.

Sin embargo, la política de distribución de bloques utilizada por Hadoop es aleatoria; lo que genera un desbalanceo de carga entre los nodos; teniendo algunos nodos del *cluster* un mayor número de bloques de entrada para ser procesados. En entornos *Grid* o *Cloud,* este desequilibrio es tratado mediante la creación de una serie de réplicas para cada bloque, lo que aumenta la posibilidad de obtener la localidad de datos para las tareas Map, además de permitir un control de tolerancia a fallos. Sin embargo, en los entornos en que existen limitaciones a la disponibilidad de disco, no es viable disponer de réplicas de los bloques de entrada.

1.1.2 Gestión de trabajos en *cluster* Hadoop

La gestión de los trabajos en un *cluster* Hadoop se realiza por el planificador de trabajos Hadoop a través de dos demonios: *jobtracker* (en el nodo *master*) y *tasktracker* (en los nodos *workers*). Los trabajos MapReduce presentados al *cluster* son inyectados al sistema en la cola de trabajos gestionada por el *jobtracker,* como se puede observar en la Figura 1.3. El orden de ejecución de estos trabajos es definido por la política de planificación de trabajos de Hadoop. Por defecto esta política es basada en una cola del tipo FIFO – *first in first out*.

Los trabajos encolados son divididos por Hadoop en un conjunto de tareas Map y tareas Reduce. Para cada tarea Map se asocia un bloque de entrada. Y como Hadoop mantiene una tabla con la ubicación de cada bloque en el *cluster*, se define así donde preferiblemente se procesará la tarea Map. El mapeo es gestionado por el *jobtracker.*

Estas tareas son asignadas a cada uno de los *tasktrackers*. Para cada tarea recibida, el demonio local crea una máquina virtual Java para ejecutarla. Cuando el nodo local finaliza su tarea, el *tasktracker* finaliza la máquina virtual e informa de su estado al *jobtracker* a través de un mecanismo basado en *heartbeat* que genera la asignación de una nueva tarea. La Figura 1.3 presenta la planificación de trabajos en Hadoop.

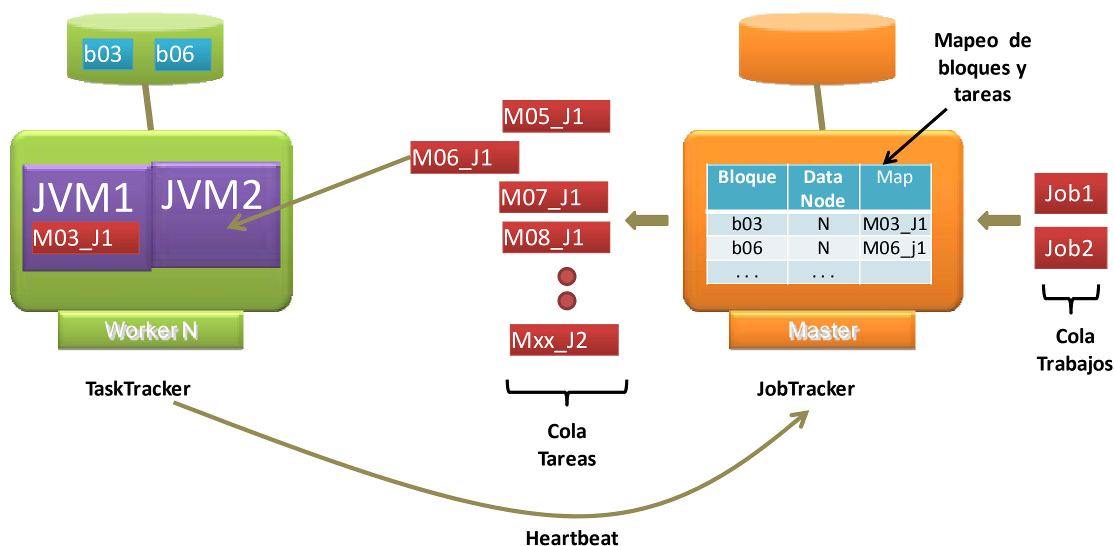

Figura 1.3 - Gestión de trabajos en Hadoop.

1.2 Aplicaciones Hadoop con características especiales

El carácter automático que ofrece Hadoop en las tareas de particionamiento de datos de entrada, de división del trabajo en tareas, de planificación de las tareas en los diferentes nodos de computación, y de gestión de la tolerancia a fallos; facilita el uso de entornos paralelos por los no expertos en computación de alto rendimiento y ha permitido el desarrollo de nuevas aplicaciones en diversos campos del conocimiento. Sin embargo, muchas aplicaciones tienen características de ejecución diferentes de las aplicaciones que se realizan tradicionalmente en un entorno Hadoop. La Bioinformática es un ejemplo de área donde se han desarrollado un gran número de aplicaciones con características muy particulares. Zou Quan, et al [12] presentó un mapa de aplicaciones bioinformáticas desarrolladas en MapReduce. Hay ejemplos de aplicaciones en la alineación de secuencias genéticas como CloudBLAST [13] y bCloudBLAST [14] que se basan en Hadoop y utilizan Basic Local Alignment Search Toll (BLAST); y GRAMMAR [15], que es una aplicación que combina BLAST con Gene Set Enrichment Analysis (GSEA) sobre Hadoop. Hay ejemplos de aplicaciones en mapping y en assembly como BlastReduce con CloudBurst [16], SEAL [17], CloudAligner [18], HPC-MAQ [19], y MrMAQ [20]. También hay ejemplos de aplicaciones con enfoque Hadoop en el análisis de la expresión génica y en el problema de análisis de variantes de tipo SNP como Myrna [21] y CloudTSS [22].

Las aplicaciones bioinformáticas del tipo "read-mapping" comparan las secuencias de ADN almacenados en archivos de lectura con un genoma almacenado en un archivo de referencia. Es común que los trabajos con diferentes archivos de lectura compartan el mismo archivo de referencia, como muestra la Figura 1.4. Sin embargo, datos de entrada compartidos no son manejados eficientemente por Hadoop causando accesos innecesarios a disco y tráfico en la red.

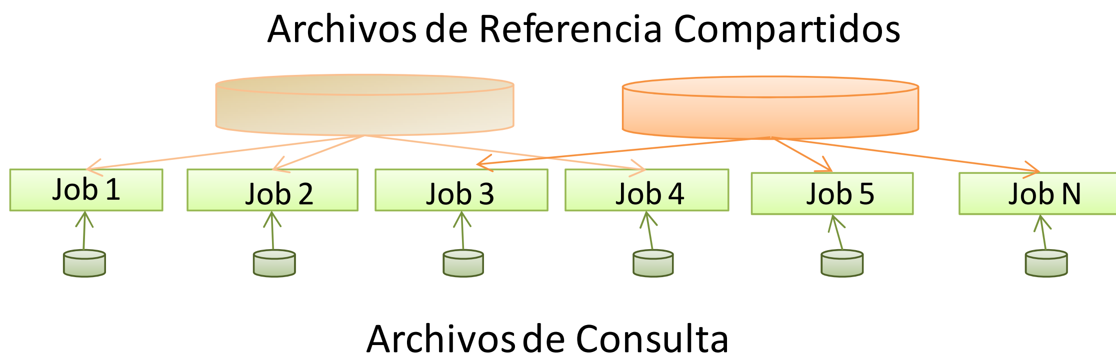

Figura 1.4 - Conjunto de datos de entrada en aplicaciones compartidas del tipo "read-mapping".

Los archivos compartidos de entrada son divididos en bloques y distribuidos entre los nodos de cómputo. Sin embargo, la política de distribución de bloques utilizada por Hadoop es aleatoria; lo que genera un desbalanceo entre los nodos, como se puede apreciar en la Figura 1.5. Puede ocurrir que, cuando un nodo con menos bloques termine de procesarlos, todavía queden bloques para ser procesados en otros nodos. En estos casos, Hadoop hace una copia del bloque desde el nodo donde esté ubicado hasta el nodo que está libre. Este problema se agrava cuando hay en la cola muchos trabajos que comparten el mismo conjunto de entrada. En estos casos puede ser imprescindible transferir el mismo bloque hacia distintos nodos; lo que genera tráfico innecesario en la red. Como muestra la Figura 1.5, en que el bloque 12 del fichero de entrada es procesado por tareas Map asignadas a nodos diferentes, entonces Hadoop necesita copiarlo para estos nodos de procesamiento.

Esto no es un problema exclusivo de aplicaciones de bioinformática sino que también está presente en aplicaciones cuyos trabajos distintos comparten el mismo archivo de entrada en un *cluster* Hadoop, como por ejemplo trabajos que hacen las búsquedas en una misma base de datos. El problema se complica cuando se ejecutan

estos trabajos en los entornos del tipo *cluster* Hadoop compartido, o en entornos en los que hay limitaciones de almacenamiento porque en estos casos no se puede disponer de réplicas para los bloques de entrada. La necesidad de transferir repetidas veces el mismo bloque de datos a distintos nodos genera tráfico en la red, que además de no ser deseado es innecesario.

Mapeo de bloques y tareas

Bloque	Data Node	Map
12	5	M12_J1
12	5	M12_J2
12	5	M12_J3

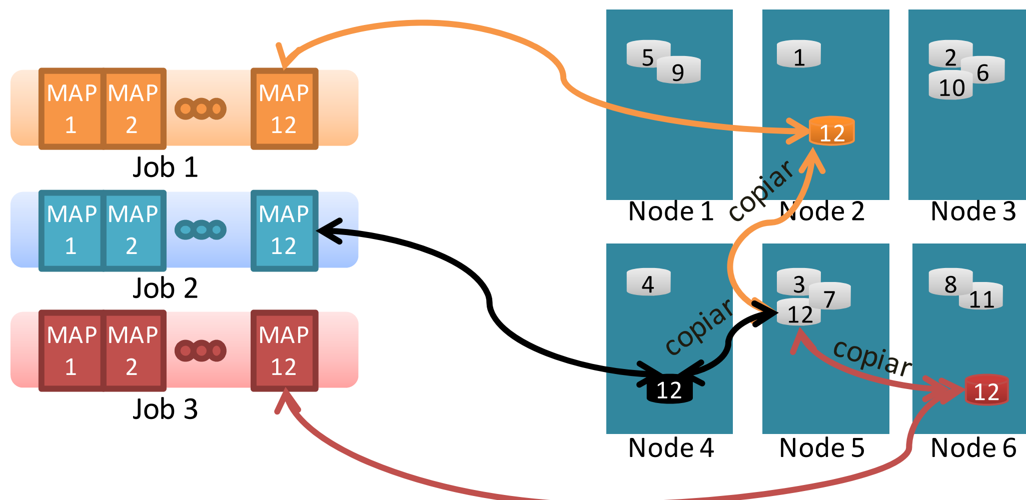

Figura 1.5 - Copias innecesarias de bloques de datos provocadas por el desbalanceo de carga en cluster Hadoop.

Otra característica de algunas aplicaciones Hadoop que nos interesan en especial, como ocurre en la aplicación MrMAQ, es el gran volumen de datos intermedios generados a partir de la fase Map para la fase Reduce. Hadoop no utiliza HDFS para almacenar estos datos. Hadoop guarda los datos intermedios generados entre las fases Map y Reduce en el sistema de archivos local de cada nodo, como muestra la Figura 1.6.

Los problemas causados por el gran volumen de datos de entrada ya están bien manejados por Hadoop. Los conjuntos de datos de entrada se dividen en bloques pequeños (*splits*) y los datos se acceden mediante *streaming*. A nivel de red, la política de planificación de tareas tiene por objeto garantizar la localidad de datos en la fase Map disminuyendo el volumen de datos transferidos.

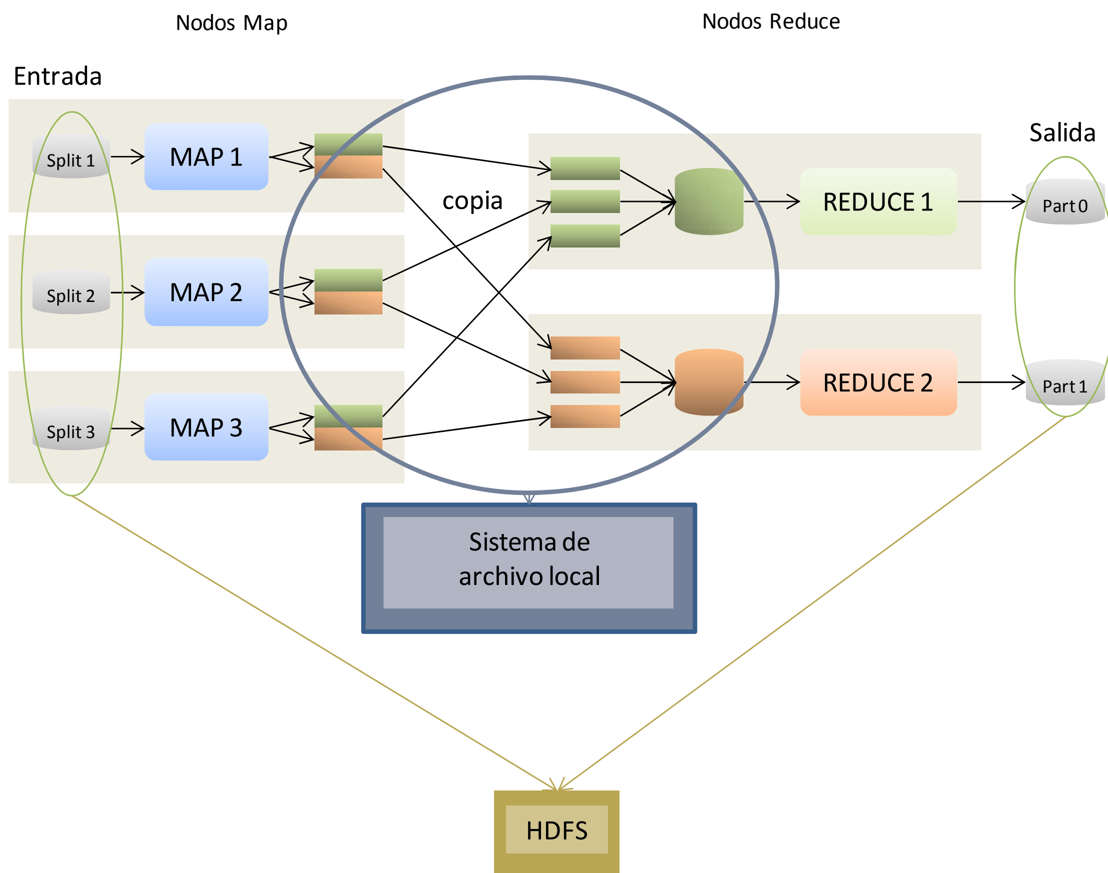

Figura 1.6 - Almacenamiento de datos en cluster Hadoop.

Sin embargo Hadoop no considera la localidad de datos al asignar tareas Reduce. No hay una implementación en la misma línea de intentar proporcionar la localidad para las tareas Map que sea aplicable en la fase Reduce. Las opciones disponibles en Hadoop para gestionar los datos intermedios son el uso de una función *combiner* (utilizada como un paso de filtración o agregación) en la salida de Map para disminuir el número de datos intermedios o el uso de alguna política de compresión de datos para reducir la cantidad de datos transferidos a las tareas Reduce. Sin embargo, estas opciones no son suficientes para aquellos casos en que el volumen de datos

intermedios es realmente grande. En la implementación de Hadoop, los datos intermedios son almacenados en *buffers* temporales mientras son generados por las tareas en ejecución, como se muestra en la Figura 1.7. Cuando estos *buffers* están llenos, los datos son cargados en el disco. Sin embargo, la velocidad de acceso a discos es considerablemente más baja que la velocidad de acceso a los *buffers* que están en memoria. Además, las aplicaciones intensivas en datos Intermedios sobrecargan el disco y causan largos tiempos de espera. Como consecuencia ocurre un aumento en el tiempo de ejecución de las aplicaciones.

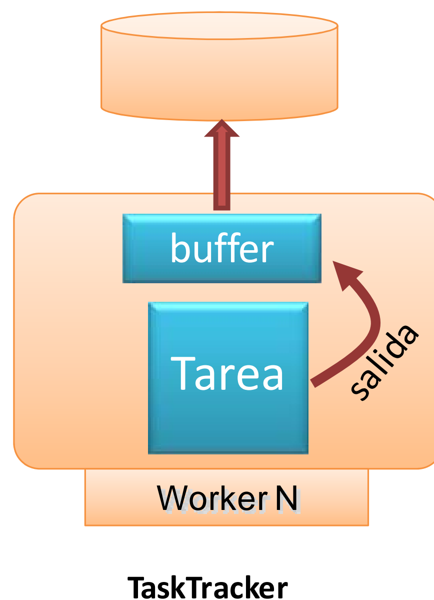

Figura 1.7 - Buffer intermediario para salida de datos de tareas Hadoop.

Tanto en la fase Map cuanto en la fase Reduce, las aplicaciones que manejan grandes volúmenes de datos intermedios vuelcan muchos datos en el disco, y esto hace con que el disco se sobrecargue.

En el caso de que los datos intermedios quepan en el buffer de memoria, la política de persistencia de datos definida por Hadoop determina que estos datos deben ser volcados a disco al final de la fase Map. La razón por la que Hadoop debe volcar los datos en el disco al final de la tarea Map, es para prevenir fallos. Si uno de los nodos que ejecuta tareas Reduce falla, no hace falta volver a procesar la tarea Map para generar los datos otra vez. Los datos simplemente son transferidos desde el disco

del nodo donde se procesó la tarea Map hasta el nodo asignado para ejecutar la nueva tarea Reduce.

1.3 Revisión de la literatura

Una revisión de la literatura nos indica que el problema de la planificación de trabajos que comparten un mismo conjunto de entradas, y el problema de la gestión de grandes volúmenes de datos intermedios, son tratados usando enfoques distintos y las soluciones para los problemas no están integrados en una misma política de planificación de trabajos.

Teniendo en cuenta que para el algoritmo FIFO, que es el algoritmo clásico utilizado en Hadoop para la planificación de trabajos, el problema de la localidad no está bien resuelto [23]; uno de los enfoques utilizados para resolver el problema es aplicar técnicas destinadas a reducir el uso de recursos de disco y de red, tratando de mejorar la localidad de los datos y los accesos a disco. Como propuso Zaharia et al [24] en el algoritmo de planificación de trabajos *Delay Scheduling* para aumentar localidad de datos en aplicaciones desarrolladas bajo el paradigma MapReduce. El algoritmo retrasa la planificación de tareas en ***K*** unidades de tiempo cuando una tarea no encuentra los datos en el mismo nodo que esté libre. Sin embargo, su uso no es adecuado para tareas de tiempo de ejecución largos porque pueden causar degradación del rendimiento en los trabajos que tengan sus tareas retrasadas. Además, el valor de ***K*** debe estar configurado para obtener buenos resultados. Establecer este valor depende del tipo de carga y el entorno de ejecución. Los valores bajos no pueden garantizar la localidad de datos y los valores demasiado altos pueden causar problemas de posposición indefinida (*starvation*).

Seo et al. [25] propuso una política de planificación de trabajos basada en técnicas de *prefetching* para *clusters* con múltiplos racks. La política hace *prefetch* de datos y aquellos bloques más requeridos son copiados entre los *raks* del *cluster*. Sin embargo, la política no garantiza un aumento de la localidad de datos para las tareas Map, sino que los datos estarán en el mismo rack donde serán procesados.

Otro enfoque que se puede dar al problema de planificación de trabajos con entradas compartidas, es la propuesta del trabajo de Zhenhua et al. [26], que recomienda la necesidad de tomar decisiones globales relacionadas con la asignación de los bloques de archivos de datos, cuando se necesita compartir un conjunto de archivos de entrada entre trabajos distintos. Su trabajo trata de resolver globalmente el problema de la asignación utilizando técnicas de programación linear entera. Sin embargo, esta propuesta diverge del diseño original de Hadoop donde las tareas son planificadas de forma individual y tratando de asignar dinámicamente las nuevas tareas.

Especialmente para los clústeres compartidos, la comunidad propuso algoritmos de planificación de trabajos como Fair-Share [27] y Capacity Scheduler [28], que intentan garantizar equidad de recursos entre distintos usuarios, o requerimiento de aplicaciones que se ejecutan en el *cluster*. Sin embargo, en muchas situaciones estos objetivos de equidad de recursos y localidad de datos se convierten en objetivos contradictorios para lograrse.

Aunque estas propuestas puedan mejorar la localidad de datos cuando se procesa solo un trabajo cada vez, no tratan el problema específico de localidad cuando muchos trabajos comparten el mismo conjunto de datos de entrada. También muchas de estas propuestas proponen cambios al diseño original de Hadoop o en la aplicación.

Para el problema de manejo de grandes volúmenes de datos intermedios, encontramos en la literatura trabajos con enfoques en la utilización de técnicas para mejorar la gestión de estos datos; o en el uso de mecanismos de almacenamiento con velocidad de acceso más altas una vez que el bajo rendimiento y la alta latencia de los dispositivos de disco duro se agravan en entornos compartidos con muchos accesos concurrentes.

Ma et al [29] propuso limitar el número de accesos concurrentes en el disco a través de un mecanismo de coordinación de las solicitudes de E/S basado en la prioridad de los trabajos. Rasmussen et al [30] propuso Themis, una implementación de MapReduce que lee y escribe registros de datos en el disco exactamente dos veces. Sin embargo Themis realiza decisiones de diseño diferentes de aquellos realizados en la implementación Hadoop. Moise et al [31] modificó el *framework* Hadoop para introducir una capa de almacenamiento de datos intermedios sin almacenar datos en el sistema de archivos local. Sin embargo, el mecanismo propuesto está más centrado en el manejo de fallos generando un impacto mínimo en el tiempo de ejecución.

Algunos trabajos proponen el almacenamiento en *cache* de datos para acelerar el acceso a los mismos. Sin embargo, a menudo estas propuestas requieren modificaciones en la aplicación y/o en el flujo de trabajo; o requieren un mecanismo sistemático de almacenamiento en *cache*, para la manipulación de archivos a través de múltiples dispositivos de almacenamiento. Casi siempre estas propuestas generan *overhead* mediante el uso de APIs. Zhao et al [32] propusieron HyCache; un sistema de archivos a nivel de usuario que gestiona los dispositivos de almacenamiento heterogéneos. Específicamente, un sistema de almacenamiento híbrido con discos duros mecánicos (HDD) y unidades de estado sólido (SDD). HyCache proporciona interfaces POSIX estándar a través de FUSE [33], y lleva a una sobrecarga causada por la necesidad del uso de sus APIs.

En la misma línea de cache de datos, Kim et al [34] propusieron SplitCache. Un mecanismo de *cache* distribuida que proporciona un almacenamiento de [clave-valor] en memoria. SplitCache mejora el rendimiento de las aplicaciones intensivas en datos de estilo OLAP, reduciendo entradas redundantes en un *framework* MapReduce. Sin embargo SplitCache sólo maneja los datos de entrada.

Luo et al [35] en su trabajo proponen el uso de una RAMCloud para un almacenamiento en línea de uso general. Sin embargo, su propuesta se centró en aplicaciones que hacen accesos aleatorios de lectura para entornos del tipo *cloud*. Anteriormente, Outerhout et al [36] propuso el uso de RAMCloud como un sistema de almacenamiento de datos para aplicaciones Web a gran escala.

1.4 Objetivos

Para solucionar los problemas enfrentados por aquellos trabajos que comparten un mismo conjunto de entradas en *clusters* Hadoop compartidos y el problema provocado por las aplicaciones que generan un gran volumen de datos intermedios, propusimos el diseño y implementación de una única solución, que por un lado gestione de manera más eficiente la planificación de las tareas Map que procesan los mismo bloques de entrada, y por otro lado maneje los datos intermedios utilizando construcciones de memoria más adecuadas.

En este trabajo construimos una nueva política de planificación de trabajos para *clusters* compartidos del tipo Hadoop denominada *Shared Input Policy* y que puede sustituir las políticas de planificación de trabajos tradicionales de Hadoop sin necesidad de cambios en el diseño original de Hadoop, ni el uso de APIs.

Shared Input Policy tiene dos focos de actuación: la gestión de los trabajos que comparten el mismo conjunto de datos de entrada a través de la planificación de las tareas Map que procesan los mismos bloques, y la gestión de los datos intermedios a través de la construcción de una RAMDISK.

Por su parte, la gestión de los trabajos que comparten el mismo conjunto de datos de entrada está estructurada en dos niveles. En el primer nivel, la política agrupa los trabajos que comparten los mismos archivos de entrada y los procesa en lotes. Y en un segundo nivel, las tareas de los diferentes trabajos procesados en el mismo lote, y que manejan los mismos bloques de datos de entrada, son agrupadas para ser ejecutadas en el mismo nodo. Este enfoque permite aumentar la localidad de datos para las tareas Map, lo que propicia un menor volumen de datos transferido por la red. También proporciona menor acceso al disco, dado el hecho de que cuando existe la necesidad de transferir los bloques de entrada entre los nodos, estos bloques son leídos desde el disco duro del nodo donde están ubicados y escritos otra vez en los nodos que los reciben.

Para la gestión de los datos intermedios propusimos introducir una RAMDISK para el almacenamiento temporal de los datos, entre los buffers intermedios creadas por Hadoop y el disco duro. Los buffers creados por Hadoop siguen existiendo. Sin embargo, en nuestra propuesta, cuando los buffers alcanzan su capacidad de almacenamiento, Hadoop vuelca los datos intermedios desde los buffers hacia la RAMDISK, y no hacia el disco duro como ocurre en la implementación por defecto. Utilizamos la RAMDISK para volcar datos intermedios tanto en los nodos donde se ejecutan las tareas Map como en los nodos donde se ejecutan las tareas Reduce. Como la velocidad de acceso a la memoria RAM es más alta que la velocidad para acceder al disco duro en algunos órdenes de magnitud, este enfoque reduce el coste de acceso a grandes volúmenes de datos intermedios, generados por las aplicaciones Hadoop intensivas en datos. Sin embargo, como la memoria utilizada para construir la RAMDISK tiene un tamaño limitado y es volátil, definimos una política de sustitución de su contenido, y una política de persistencia para el caso de tolerancia de fallos.

1.4.1 Objetivo principal

El objetivo principal de este trabajo es diseñar, implementar y evaluar una nueva política de planificación de trabajos para *clusters* compartidos del tipo Hadoop, que mejore el tiempo de *makespan* necesario para procesar lotes de trabajos que, por un lado, compartan el mismo conjunto de datos de entrada, y que por otro lado, generan un gran volumen de datos intermediarios entre las fases Map y Reduce. Las estrategias utilizada para resolver este problema han sido la de disminuir el volumen de datos transferido entre los nodos aumentando la localidad de datos durante la planificación de tareas Map que comparten los mismos datos de entrada, y disminuir el volumen de datos intermedios volcados al disco durante las fases Map y Reduce.

1.4.2 Objetivos específicos

Para alcanzar el objetivo principal definimos los siguientes objetivos específicos.

- Caracterizar aplicaciones intensivas de datos, e identificar los posibles cuellos de botella en un *cluster* Hadoop compartido. Elegimos la aplicación de bioinformática MrMAQ del tipo *short mapping* que comparte el conjunto de datos de entrada entre distintos trabajos; además de generar grandes volúmenes de datos intermedios entre las fases Map y Reduce.

- Diseñar y implementar una política de planificación de trabajos Hadoop que aumente la localidad de datos para tareas Map de distintos trabajos procesados en lote, y que compartan el mismo conjunto de datos de entrada.

- Construir una RAMDISK que pueda ser utilizada por la política de planificación de trabajos como un mecanismo de almacenamiento temporal de datos intermedios.

- Establecer una política de sustitución para los datos almacenados en la RAMDISK.

- Evaluar el tiempo de *makespan* obtenido por la política de planificación de trabajos Hadoop construida.

1.5 Organización de la Memoria

Este documento está organizado en cinco capítulos.

Capítulo 1 - Describimos la problemática asociada al problema de planificación de trabajos que comparten el mismo conjunto de datos de entrada, y al problema de gestión de grandes volúmenes de datos intermedios. También presentamos, como resultado de una revisión de la literatura, diferentes enfoques para solucionar a estos problemas. Definimos el objetivo de nuestro trabajo y ofrecemos una descripción de nuestra propuesta de solución.

Capítulo 2 - Hacemos una breve introducción a cerca de los entornos paralelos compartidos. Presentamos las principales características relacionadas con la planificación de trabajos en *clusters* Hadoop, y la importancia de la sintonización de sus parámetros de configuración. También presentamos algunos ejemplos de planificadores de trabajo Hadoop con diferentes enfoques de planificación.

Capítulo 3 - Presentamos como el *framework* Hadoop lee y escribe datos en los nodos del *cluster*; y como gestiona los datos intermedios. Proponemos una nueva política de planificación de trabajos para *clusters* Hadoop compartidos denominada *Shared Input Policy*. Describimos sus características de diseño e implementación.

Capítulo 4 - Ofrecemos una descripción de los experimentos realizados. Describimos las aplicaciones utilizadas en los experimentos, el entorno en los que fueron realizados y los indicadores de rendimiento que fueron aplicados. También mostramos los

resultados obtenidos mediante la ejecución de los experimentos y realizamos un análisis de los mismos.

Capítulo 5 - Presentamos las conclusiones, las publicaciones que han sido generadas, y proponemos las líneas futuras hacia la continuación de este trabajo de investigación.

Capítulo 2 -

PLANIFICACIÓN DE TRABAJOS

Las necesidades de cómputo de las aplicaciones han crecido a lo largo del tiempo de manera ininterrumpida. Para atender estas demandas, la investigación en computación ha seguido principalmente por dos caminos: incrementar la capacidad de cómputo de los ordenadores, y desarrollar técnicas de cómputo distribuido y paralelo. Por otro lado, han surgido nuevas aplicaciones paralelas con requerimientos más estrictos de tiempo de retorno (*turnaround*), de calidad de servicios (QoS) y con acceso a volúmenes de datos cada vez más grandes.

Estas aplicaciones tienen necesidades propias de recursos de memoria, CPU y ancho de banda de red y disco. Gestionar la ejecución de estas aplicaciones paralelas, cada vez más complejas, en entornos paralelos compartidos, es un reto para los investigadores en computación de alto rendimiento.

2.1 Introducción a los entornos paralelos compartidos

A continuación, nos proponemos situar los *cluster* de naturaleza compartida, dentro de los entornos de cómputo paralelo más habituales. Esta caracterización, nos permitirá fijar aspectos críticos de los mismos, en relación a los tipos de planificadores más adecuados en cada caso.

La Figura 2.1 presenta una taxonomía para los sistemas paralelos propuesta por Hanzich [37], basada en las taxonomías de Flynn y Tanenbaum, donde es posible clasificar diferentes sistemas disponibles para el cómputo distribuido y paralelo.

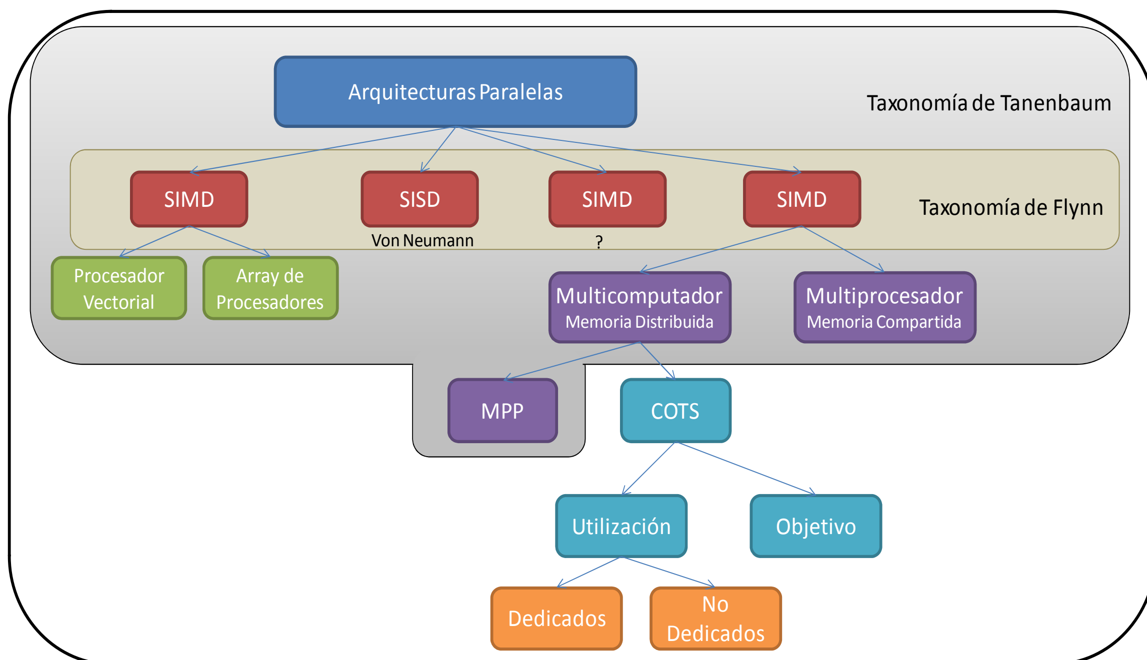

Figura 2.1 - Taxonomía aplicada a los sistemas paralelos. Extraído y adaptado de [37].

En el ámbito de los multicomputadores una opción son las máquinas tipo Massive Parallel Processors (MPP). Estos computadores están basados en máquinas con un gran número de procesadores y un alto nivel de acoplamiento entre ellos, que proporcionan elevada capacidad de cómputo. Para su funcionamiento, estas máquinas necesitan de *hardware* y *software* específico con costos económicos elevados; por este motivo ha disminuido la utilización de estos computadores de alto rendimiento basados en máquinas MPPs en los últimos años. La Figura 2.2, presenta la distribución de los distintos sistemas paralelos, dentro de la lista del Top500, en las dos últimas décadas.

La utilización de computadores de alto rendimiento basados en *clusters* ha aumentado en los últimos años. Es posible construir *clusters* a partir de componentes comerciales fácilmente accesibles y de bajos costos económicos. Estos son *clusters* del tipo COTS (Commodity Of-The-Shelf) cada vez más comunes pero tienen niveles de acoplamiento más bajos que las MPPs.

Los *clusters* tipo COTS pueden ser clasificados, a partir de la taxonomía propuesta, de acuerdo con la utilización de sus recursos o de acuerdo con su objetivo. Y de acuerdo con la utilización de sus recursos, los *clusters* tipo COTS pueden ser

clasificados en dedicados o no-dedicados. Los *clusters* dedicados poseen como característica la ejecución solamente de cargas controladas por el sistema de gestión del *cluster* y los recursos sintonizados para una única aplicación. Los *clusters* no-dedicados poseen como característica la ejecución de cargas no controladas por el sistema de gestión del *cluster*. En estos *clusters* las aplicaciones que se ejecutan comparten los recursos disponibles entre ellas, como muestra la Figura 2.3. Esto hace que los *clusters* compartidos tengan costo muy económico cuando son comparados a los *clusters* dedicados.

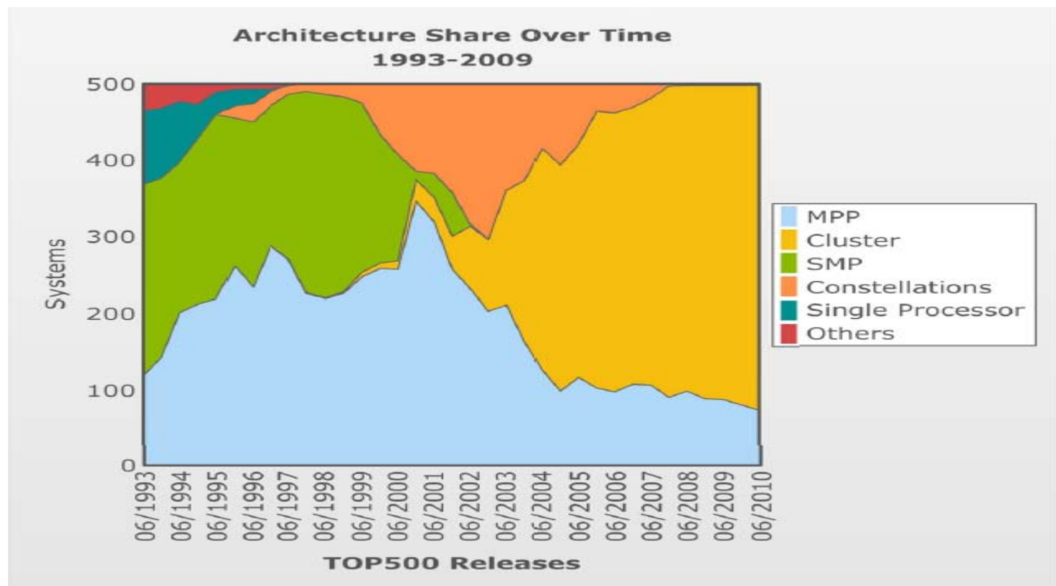

Figura 2.2 – Distribución de la arquitectura de los computadores en la lista Top500 de las últimas dos décadas

La posibilidad de que los *clusters* no-dedicados ejecuten aplicaciones con distintos requerimientos exige la investigación de técnicas que permitan la combinación de estas cargas sin comprometer estos requerimientos, ni las prestaciones de las aplicaciones. En la literatura podemos encontrar técnicas basadas en servicios de ejecución remota [Piranha [38] y Process Server[39]]; las basadas en migración de tareas [Mosix [40], Condor [41], Stealth [42] y Sprite[43]]; la construcción de máquinas virtuales – MV [[44] y [37]] y la utilización de técnicas de planificación de trabajos, que permitan la coexistencia de diferentes tipos de aplicaciones con prestaciones adecuadas.

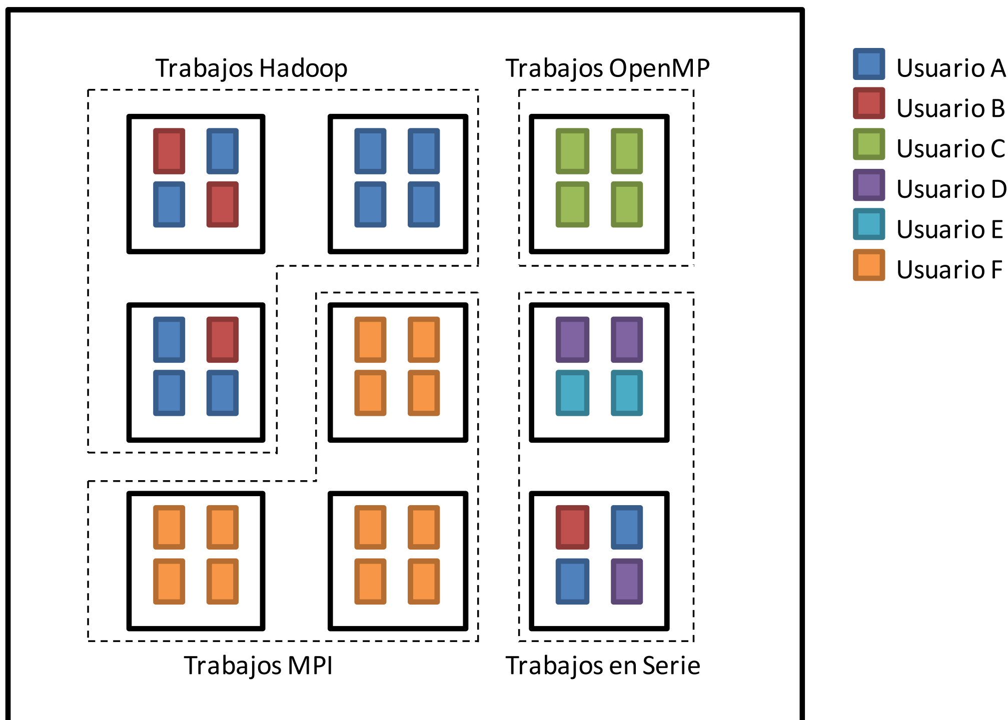

Figura 2.3 - Representación de un cluster compartido.

En este documento proponemos una nueva política de planificación de trabajos para permitir que *clusters* Hadoop puedan ser compartidos entre trabajos distintos. La política propuesta se inscribe en la planificación de trabajos que comparten ficheros de entrada, y que el manejo de los datos de entrada y de los datos intermedios generados entre las fases Map y Reduce, requieren una gestión más eficiente.

2.2 Planificación de trabajos en clusters Hadoop

Hadoop es un *framework* desarrollado como un proyecto Apache y que implementa MapReduce inspirado en la propuesta de Google. Es un sistema de código abierto y después de su implementación original para *clusters* han sido propuestas implementaciones para varias arquitecturas específicas, incluso para Cell B.E, GPUs y procesadores multi/*manycore*.

El *framework* Hadoop realiza de forma automática la fragmentación y distribución de los archivos de entrada, la división de un trabajo en tareas Map y Reduce, la planificación dinámica de las tareas frente al modelo clásico de planificadores estáticos, realiza el control de fallos de los nodos y gestiona la necesidad de comunicación entre los nodos del *cluster*.

Hadoop se ejecuta sobre un sistema de archivos distribuidos, Hadoop Distributed File System – HDFS que es capaz de almacenar archivos de tamaños grandes a través de muchas máquinas. La fiabilidad es obtenida por la replicación de los dados a través de las máquinas del *cluster* sin necesidad de implementación de Raid.

El desarrollo de Hadoop está basado en el modelo *master/worker* donde es implementado el motor MapReduce: *JobTracker* (el nodo *master*) y *TaskTracker* (los nodos *workers*).

Los trabajos MapReduce son inyectados al sistema en una cola gestionada por el *JobTracker*. Los trabajos encolados son divididos por Hadoop en un conjunto de tareas. Estas tareas son asignadas a cada uno de los *TaskTrackers*. Cuando un *TaskTracker* finaliza su tarea informa de su estado al *JobTracker* que realiza la asignación de una nueva tarea. La Figura 2.4 presenta el sistema de cola de trabajos de Hadoop.

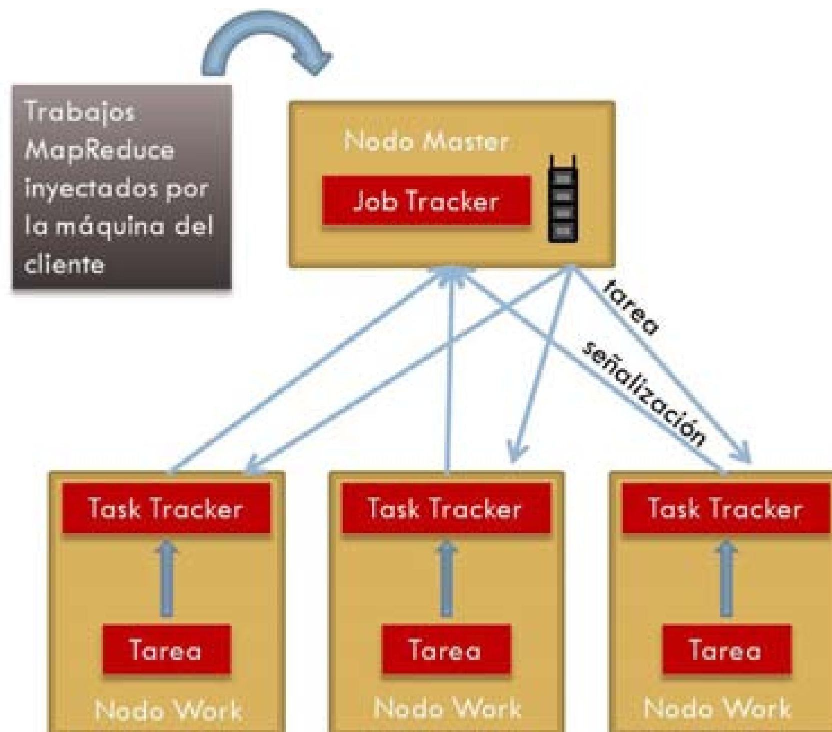

Figura 2.4 – Componentes del Planificador en Hadoop

Las primeras tareas asignadas a los *TaskTrackers* son las tareas de Map. Estas tareas aplican la función Map a los fragmentos del archivo de datos de entrada que están distribuidos por los nodos *workers* del *cluster*. Hadoop hace la distribución de las tareas a los *TaskTrakers*, de modo que estén más cerca de los nodos cuanto sea posible. Las tuplas <clave, valor> generadas por cada tarea Map, son almacenadas temporalmente en los discos locales de cada *TaskTraker*.

Cuando la primera tarea Map haya terminado, el *JobTracker* bajo petición de los *TaskTrakers*, empieza a distribuir las tareas Reduce. Las tareas Reduce reciben las salidas de las tareas Map y las almacenan en el sistema de archivo local. Cuando todas las tareas Map hayan terminado, las tareas procesan las listas de tuplas recibidas y generan la salida final del trabajo. La Figura 2.5 ilustra una ejecución MapReduce en el *framework* Hadoop.

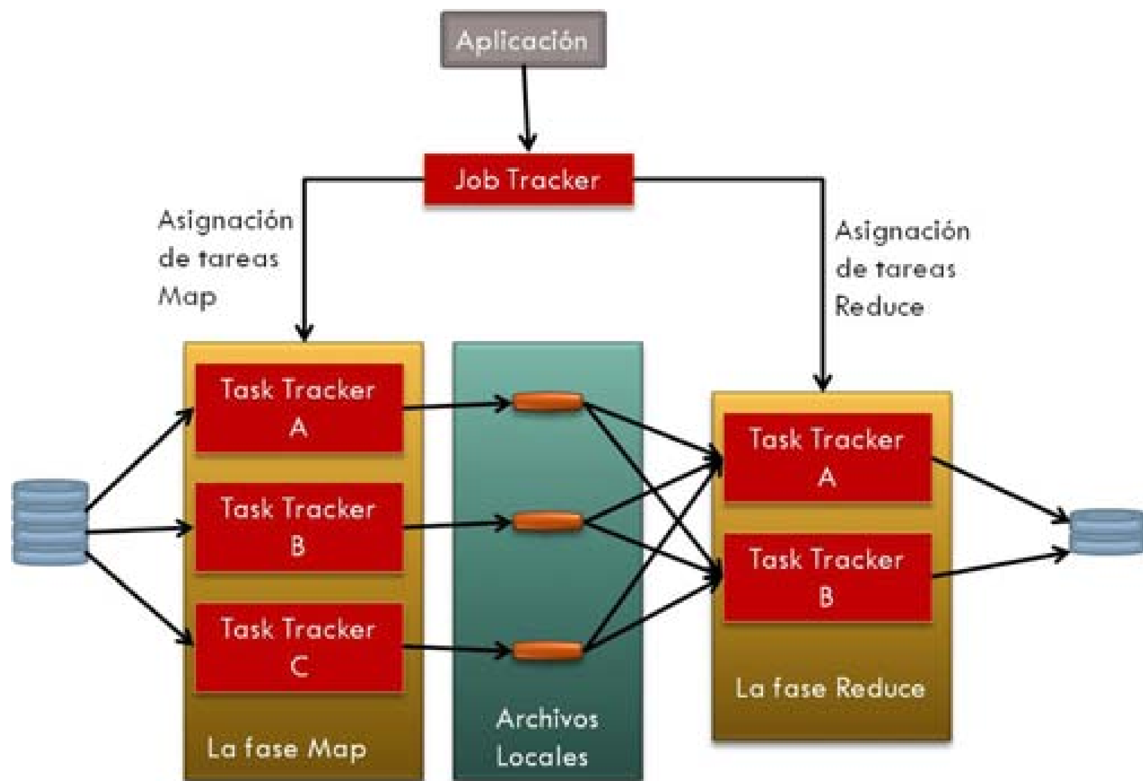

Figura 2.5 – Ejecución de Hadoop

Hadoop ejecuta varias tareas Map y Reduce al mismo tiempo en cada *worker* para que se solapen la computación y E / S. Cada *TaskTracker* le dice al *JobTracker*

cuando éste ha finalizado sus tareas. El planificador entonces asigna nuevas tareas al *TaskTracker*. La planificación de las tareas se realiza de forma dinámica.

Como MapReduce es altamente paralelizable, Hadoop necesita realizar el control de fallos. El nodo *master* hace la verificación periódica de la finalización de las tareas a ejecutar. En el caso de que una tarea a realizar, no conteste al *master* en un intervalo de tiempo determinado, esta es marcada como en estado de fallo. Todas las tares de Map que han sido concluidas, pero no tengan enviado toda la salida hacia el nodo Reduce, serán planificadas una vez más en caso de que el nodo donde están haya fallado. Además, todas las tareas de Map y Reduce en ejecución en el nodo que ha fallado, también serán planificadas otra vez. La necesidad de volver a planificar las tareas de Map ya concluidas, es porque los datos intermediarios generados por la función Map, son almacenados en el disco local de cada máquina. Así, en caso de que haya ocurrido fallo en una máquina que hay procesado una tarea Map, no es posible acceder a los datos en su disco.

Las versiones más recientes de Hadoop, también realizan *checkpoints* periódicos de las estructuras de datos existentes en el *master*. En caso de que ocurra un fallo en el *master*, una nueva ejecución a partir del punto de *checkpoint* puede ser comenzada en un nuevo *master* elegido por el *JobTracker*.

2.3 Planificadores de Trabajos Hadoop

El uso crecente de Hadoop como *framework* para ejecutar aplicaciones MapReduce, ha fomentado el surgimiento de nuevas propuestas de algoritmos de planificación de trabajos, para atender diferentes tipos de requerimientos, y con enfoques de solución muy variados. La facilidad del diseño original de Hadoop, implementado como un conjunto de clases Java jerárquicas, permite que nuevas políticas de planificación de trabajo sean fácilmente acopladas al *framework* a través de la proposición de nuevas clases de planificación.

La política de planificación de trabajo por defecto en Hadoop está basada en la orden de presentación de los trabajos al *cluster* e implementa un algoritmo do tipo *First In First Out*. La simplicidad de esta política no suprime el reto de planificar trabajos cada vez más complejos, que manejan volúmenes de datos crecentes; y con los recursos del *cluster* siendo utilizados de manera compartida entre trabajos con diferentes tipos de requerimiento.

En este apartado presentemos algunos ejemplos de planificadores de trabajos Hadoop propuestos en la literatura. No pretende ser un estudio completo o detallado sobre el tema, sino una ilustración de la variedad de algoritmos de planificación de trabajos y sus enfoques. La Figura 2.6 presenta una taxonomía para planificadores de trabajos Hadoop y tiene como objetivo posicionar la política propuesta en esta investigación entre el grupo de políticas de trabajos Hadoop para *clusters* no dedicados.

El *framework* Hadoop dispone de implementaciones para arquitecturas específicas, como por ejemplo *Phonix++* [11] y *Metis* [45] desarrollados para entornos *multicore* con memoria compartida, o *Dandelion* [46] propuesto para arquitecturas hibridas constituidas por CPUs y GPUs. Para las arquitecturas tradicionales del tipo *cluster*, hay implementaciones del *framework* que permiten la ejecución de múltiples instancias de Hadoop, como por ejemplo *Hadoop on Demand* [47] que permite la construcción de *clusters* virtuales sobre *clusters* físicos. La administración de los recursos en estos casos se puede hacer utilizando gestores de recursos como *SGE 6.2* [48], *Condor* [49] o *Torque* [50]. También pueden existir para las arquitecturas tradicionales, políticas de planificación para entornos con una única instancia de Hadoop en ejecución. Estas políticas pueden estar dirigidas para *clusters* dedicados, como es el caso de la política por defecto de Hadoop, la política del tipo *FIFO*. O bien estas políticas pueden estar direccionadas para entornos no-dedicados. Hay ejemplos de políticas de planificación de trabajos para *clusters* no-dedicados que intentan asegurar una distribución justa y equitativa de los recursos de *cluster* entre los diferentes trabajos y usuarios: como por ejemplo *Fair Scheduler*, *Capacity Scheduler* y *Delay Scheduler*. También existen ejemplos de planificadores de trabajo Hadoop que realizan procesamiento especulativo para intentar obtener mejores prestaciones: como *LATE Scheduler* y *SAMR Scheduler*. Otro ejemplo de planificador de trabajos para *clusters* no-dedicados es *Dynamic Proportional Scheduler*, cuya política de planificación está basada en un control dinámico de los recursos del *cluster* y permite que el usuario adquiera u ofrezca recursos dinámicamente.

Una revisión en la literatura nos permite la ampliación de esta taxonomía de manera sencilla y rápida. A continuación presentamos de manera resumida algunos de estos ejemplos de planificadores para *clusters* Hadoop no-dedicados. También presentamos la política FIFO. Aunque ésta sea una política de planificación de trabajos para *clusters* dedicados, como es la política por defecto de Hadoop muchas veces también es utilizada en entorno no-dedicados.

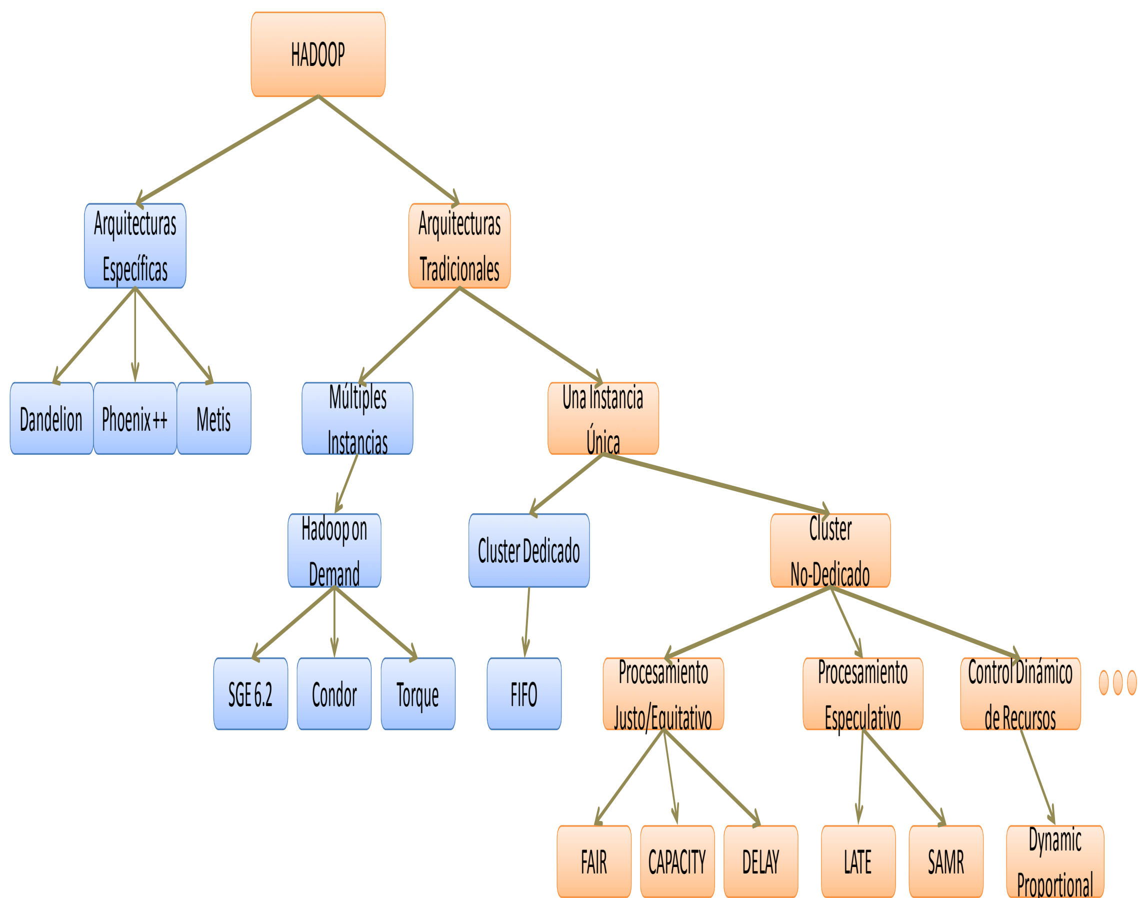

Figura 2.6 - Taxonomía HADOOP

FIFO (First In First Out)

Esta es la política de planificación de trabajos estándar de Hadoop y se basa en una cola FIFO [51] y [52]. Los trabajos se ejecutan en el mismo orden en el que entran a la cola del planificador. En la práctica, todas las tareas Map de un trabajo deben ser procesadas antes de que comience el siguiente trabajo. Cuando un slot en un nodo está libre y todavía hay tareas Map del trabajo que se está ejecutando a la espera de que se inicie, el demonio *JobTracker* enviará una de estas tareas para el nodo. Incluso si ninguna de las tareas pendientes tiene datos de entrada situados en el nodo donde está el slot libre. Esto hace difícil el mantenimiento de la localidad de datos. Además de aumentar el tráfico en la red, ya que los datos de entrada deben ser copiados desde el nodo donde están almacenados hasta el nodo donde se realiza la tarea Map.

FIFO es una política de planificación de trabajos muy simple y su uso permite obtener buenos resultados de planificación cuando los trabajos planificados tienen las mismas características. Sin embargo, cuando los trabajos que se ejecutan en el *cluster* tienen características diferentes entre ellos, como por ejemplo tamaño de los datos de entrada o tiempo de ejecución, esta política de planificación de trabajos muchas veces no genera las prestaciones adecuadas [23]. Los trabajos pequeños que tienen tiempos de respuesta cortos necesitan esperar la finalización de todos los trabajos que se encuentran en las primeras posiciones de la cola. Esto puede causar cierto retraso en el tiempo de respuesta de los trabajos cortos sobre todo si los trabajos que están en posiciones iniciales de la cola son muy largos.

PLANIFICADORES FAIR Y CAPACITY

En la distribución Apache de Hadoop otras dos políticas de planificación de trabajos están disponibles: *FAIR Scheduler* [27] y *Capacity Scheduler* [28]. Son políticas de planificación de trabajos que intentan compartir de manera justa el uso de los recursos del *cluster* (tiempo de CPU, memoria, almacenamiento en disco, etc) entre los trabajos presentados.

FAIR Scheduler fue propuesto por Facebook. Los recursos del *cluster* se dividen en *pools*. Cada *pool* cuenta con una cantidad mínima garantizada de los recursos, y se ha fijado el número de tareas que pueden ejecutarse simultáneamente. Los usuarios asignan el trabajo a un *pool* específico. Los *pools* que están más sobrecargados reciben más recursos, siempre que se respeten las cantidades mínimas de recursos de los demás *pools*. Con el tiempo, esto permite que los diferentes tipos de trabajos reciban una cantidad similar de recursos.

Capacity Scheduler fue propuesto por Yahoo. La política de planificación de trabajos organiza los recursos de *cluster* en colas. Las colas pueden tener una estructura jerárquica y cada una de ellas posee su capacidad de recursos predefinida. Los usuarios del *cluster* asignan los trabajos a su propia cola. Por lo tanto, los recursos se distribuyen de manera justa entre los usuarios, independientemente del número de trabajos que cada usuario haya asignado al *cluster*. Sin embargo, los trabajos en una misma cola son planificados a través de la política de planificación de trabajos FIFO (estándar de Hadoop).

DELAY SCHEDULING

Zaharia et al [24] propuso el algoritmo de planificación de trabajos *Delay Scheduling* para aumentar la localidad de datos en aplicaciones desarrolladas bajo el paradigma MapReduce. La motivación para construcción del planificador *Delay Scheduling* fue la dificultad en conciliar planificación justa de la propuesta *Fair Scheduler* con localidad de datos de la política FIFO.

La técnica de planificación de trabajos propuesta en *Delay Scheduling* es relativamente simple. Cuando el trabajo que debe ser ejecutado a continuación (según la política que busca equidad de recursos entre los trabajos) no puede iniciar una tarea con datos locales, se espera una pequeña cantidad de tiempo, dejando que otros trabajos lancen tareas en su lugar. Hay por lo tanto una relajación en la política de distribución justa de los recursos para que los trabajos aumenten la posibilidad de obtener localidad de datos en sus tareas Map.

Sin embargo trabajos cortos pueden tener elevados tiempos de espera cuando permiten ser remplazados por trabajos largos en *cluster* que utilizan *Delay Scheduling* como política de planificación de trabajos.

Dynamic Proportional Scheduler

Sandholm y Lai [53] propusieran el planificador *Dynamic Proportional Schedule*r que permite a los usuarios del *cluster* controlar los recursos que tienen disponibles de manera dinámica a lo largo del tiempo.

El diseño de *Dynamic Proportional Scheduler* establece un mecanismo que posibilita a los usuarios comprar u ofrecer slots de procesamiento, tanto de tareas Map cuanto de tareas Reduce dinámicamente. Los recursos asignados a un usuario son proporcionales a su capacidad y disposición de pagar por ellos, e inversamente proporcional a los valores globales. Cuando un recurso es asignado a un usuario, sus créditos son consumidos y restados del presupuesto del usuario.

Dynamic Proportional Scheduler posibilita que el planificador decida de manera más eficiente qué trabajos y usuarios deban ser priorizados. Además permite que los

propios usuarios optimicen los recursos que tienen disponibles de acuerdo con la prioridad y requerimiento de sus trabajos.

LATE Scheduler

LATE (Longest Approximate Time to End) propuesto por Zaharia el al [54] es un planificador de trabajos especialmente diseñado e implementado para *cluters* donde se ejecute procesamiento especulativo para maximizar las prestaciones. Los *clusters* que utilizan máquinas virtuales como Amazon's Elastic Compute Cloud (EC2) [55] son ejemplos de los entornos adecuados para esto tipo de planificador.

Los algoritmos de planificación de trabajos estándares de Hadoop que realizan procesamiento especulativo, intentan identificar los nodos con retraso tan pronto como sea posible, e inicializan el procesamiento especulativo de tareas utilizando una heurística sencilla de comparar el progreso de una tarea con el promedio del progreso de todas las demás tareas. Aunque esta técnica funcione relativamente bien en *clusters* homogéneos puede llevar a resultados erróneos en un *cluster* con nodos heterogéneos. En nodos con características distintas puede resultar difícil identificar si el procesamiento es ligeramente más lento que la media o si está realmente retrasado. El coste de inicializar una tarea especulativa equivocada es alto porque estas tareas compiten por los recursos del *cluster* con las demás tareas que se están ejecutando.

El planificador de trabajos LATE está implementado bajo tres principios: elegir y priorizar las tareas que más tardarán en finalizar, seleccionar los nodos más rápidos para ejecutarlas, y limitar la ejecución de tareas especulativas para evitar sobrecarga en el *cluster*. Para estimar el tiempo de ejecución de una tarea LATE utiliza un algoritmo estático que define una puntuación basada en el progreso de la tarea y presume que los nodos ejecutan las tareas en una velocidad constante.

SAMR Scheduler

SAMR (Self-Adaptive MapReduce scheduling algorithm) ha sido propuesto por Chen et al. [56] y también es un planificador de trabajos que ejecuta procesamiento especulativo de tareas orientado a *clusters* heterogéneos. Está basado en los mismos principios de LATE, sin embargo calcula el tiempo de progreso de cada tarea dinámicamente y se adapta a las variaciones del entorno automáticamente.

Cuando un trabajo se inicializa, SAMR asigna las tareas Map y Reduce a un conjunto de nodos. Mientras tanto, lee el historial con informaciones de log mantenidos en cada nodo y que son actualizados después de cada ejecución. A continuación SAMR ajusta el peso del tiempo de ejecución para cada tarea Map y Reduce basado en el historial de las informaciones recogidas. Con estas informaciones más precisas SAMR define que tareas serán reinicializadas de manera especulativa. SAMR también identificada los nodos más lentos y define este conjunto de nodos lentos dinámicamente. Las tareas reinicializadas en modo especulativo no son asignadas a este conjunto de nodos lentos para evitar más retrasos. Para realimentar el algoritmo de planificación, SAMR recoge las informaciones de la tarea que se finalice primero: o la que estaba retrasada o de la que se reinicializó de manera especulativa.

2.4 Sintonización de Parámetros

Sin embargo, la política de planificación de trabajos es sólo uno de los muchos parámetros de configuración del *framework* Hadoop. La distribución 1.2.1 Apache de Hadoop utilizada en esta investigación posee más de 190 parámetros ajustables. Y el rendimiento de Hadoop se puede mejorar sin aumentar los costos de hardware, al sintonizar varios parámetros de configuración claves para las especificaciones del *cluster*.

Realizar la sintonización de estos parámetros, no es una tarea sencilla porque un único parámetro puede tener efecto sobre los demás y sobre el rendimiento del sistema. La Figura 2.7 presenta un esquema sobre los parámetros de sintonización. El sistema posee una configuración por defecto de estos parámetros, y la definición de sus valores puede ser realizada por el programador y/o administrador del sistema, para obtener prestaciones más adecuadas. En la distribución Apache 1.2.1 de Hadoop los parámetros de configuración están distribuidos entre cuatro archivos de configuración.

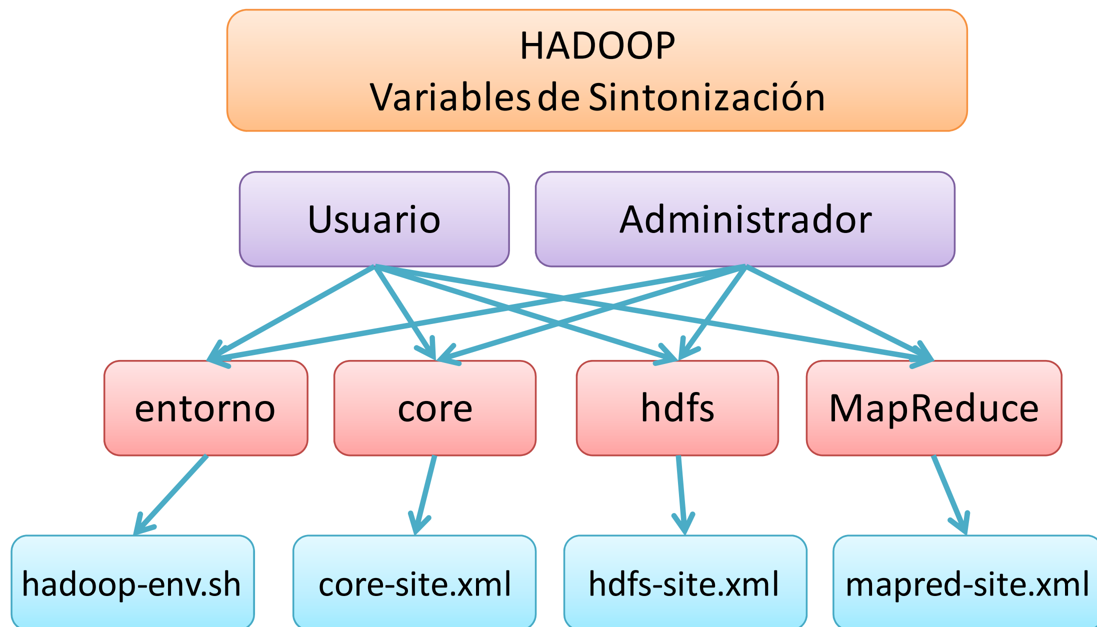

Figura 2.7 - Sintonización de Parámetros en HADOOP

En el archivo *hadoop-env.sh* están definidas las variables del entorno. Como por ejemplo la implementación JAVA utilizada (JAVA_HOME), la máxima cantidad de *heap* que se puede utilizar (HADOOP_HEAPSIZE) y el directorio donde serán almacenados los archivos de log del sistema (HADOOP_LOG_SIZE).

En el archivo *core-site.xml* está la definición del núcleo del sistema. Como por ejemplo el nombre del sistema de archivo distribuido con la autoridad que determina su implementación: servidor, el número de puerto, protocolo de comunicación entre los nodos, etc.

Las variables que definen el sistema de archivos distribuidos de Hadoop se localizan en el archivo *hdfs-site.xml*. Son variables que determinan entre otras características, el tamaño del bloque de entrada de los nuevos archivos (*dfs.block.size*) que por defecto tiene el valor de 64 MB, o el número de réplicas para cada bloque (*dfs.replication*).

En el archivo *mapred-site.xml* están las variables que definen los parámetros utilizados durante la planificación de trabajos, como por ejemplo el número de slots disponibles en cada nodo para tareas Map y para tareas Reduce; el número de tareas Map y Reduce que serán creadas para cada trabajo ejecutado en el *cluster*; o la cantidad de memoria destinado a la máquina virtual JAVA de cada tarea Map y

Reduce. Algunas definiciones importantes para este trabajo de investigación utilizan las variables de configuración localizadas en mapred-site.xml.

La variable *mapred.jobtracker.taskScheduler* define la clase responsable por planificar las tareas. Cambiamos el contenido de esta variable por la clase que establece la política de planificación de trabajos *Shared Input Policy* propuesta en este trabajo.

Si habrá procesamiento especulativo o no de las tareas Map y Reduce es determinado a través de las variables *mapred.map.tasks.speculative.execution* y *mapred.map.tasks.speculative.execution*. Establecemos que en nuestro entorno de investigación no habría procesamiento especulativo, por razón del coste en ejecutar más de una instancia de una misma tarea en paralelo, en un entorno compartido.

La gestión de los datos intermedios en Hadoop ocurre en dos etapas distintas. La primera etapa es donde los datos son generados durante el procesamiento de las tareas Map. A continuación, estos datos necesitan ser enviados para los nodos donde serán procesados por las tareas Reduce, en una segunda etapa. En ambas etapas se hace necesario construir un *buffer* temporal para almacenamiento de los datos generados por la tarea Map y otro *buffer* para los datos recibidos por la tarea Reduce.

En el archivo *mapred-site.xml* también hay un conjunto de variables para la definición y gestión de estos *buffers*. El tamaño del *buffer* para la salida de cada tarea Map es definido por la variable *io.sort.mb*, y su valor por defecto es de 100 MB. La variable *io.sortrecord.percent* determina qué porcentaje del *buffer* está reservado para los metadatos (5% por defecto), y lo que resta es destinado a almacenar los datos. La variable *io.sort.spill.percent* determina un umbral a partir del cual los datos que están en el *buffer* son volcados en el disco duro. Por defecto este valor es de 80% del tamaño del *buffer*.

El tamaño del *buffer* temporal creado por Hadoop para recibir los datos de entrada de la tarea Reduce derivados de cada tareas Map es definido por la variable *mapred.job.shuffle.input.buffer.percent*. Su valor por defecto es de 70% y establece el porcentaje máximo del *heap* que puede ser usado por el *buffer* para esto objetivo. Si el volumen de datos es superior a este umbral, los datos deben ser volcados directamente en el disco. También son definidos dos límites de ocupación del *buffer*.

La variable *mapred.job.shuffle.merge.percent* determina el tamaño máximo del *buffer* a partir del cual los datos pasan a ser volcados a disco. Por defecto su valor es de 66% del tamaño asignado al *buffer*. Y la variable *mapred.inmem.merge.threshold* determina el número máximo de entrada de diferentes tareas Map que puede ser mantenido en el *buffer*. Por defecto, si el *buffer* ya almacena entradas de más de 1000 tareas Map distintas, entonces empieza a volcar su contenido a disco.

Las variables de control de los dos *buffers* fueron utilizadas en nuestra investigación para permitirnos evaluar distintos escenarios y compararlos con el algoritmo de planificación de trabajos propuesto en este documento.

Sin embargo la sintonización de un número tan elevado de parámetros, y estando relacionado entre ellos, puede ser demasiado compleja para el usuario y/o administrador, que procura obtener rendimientos más adecuados. Para auxiliar en esta tarea compleja de sintonización se puede utilizar herramientas automatizadas. El proyecto abierto Starfish [57] es una herramienta para análisis y ajuste automático de sistemas que manejan grandes volúmenes de datos. Starfish permite definir los parámetros de Hadoop que mejor se ajustan a las necesidades de la aplicación, y así obtener mejores prestaciones.

El proyecto Starfish [58] afronta este reto mediante una combinación de técnicas de optimización de consulta a datos basada en el costo, el procesamiento robusto y adaptativo de consultas, el análisis estático y dinámico de programas, el muestreo dinámico de datos y perfiles de tiempo de ejecución, y el aprendizaje automático estadístico aplicado a la instrumentación de datos. Starfish sintoniza los parámetros de Hadoop adaptándose a las necesidades del usuario y a las cargas de trabajo del sistema para proporcionar un buen rendimiento de forma automática, sin necesidad de que los usuarios comprendan o manipulen las muchas variables de ajuste de Hadoop.

La Tabla 2.1 presenta los valores por defecto de algunos de los parámetros de de Hadoop, que fueron sintonizados durante la experimentación realizada en este trabajo de investigación.

Parámetro	Valor por defecto
HADOOP_HEAPSIZE	1000 MB
dfs.block.size	64 MB
dfs.replication	3
mapred.jobtracker.taskScheduler	FIFO
mapred.map.tasks.speculative.execution	2
mapred.map.tasks.speculative.execution	1
io.sort.mb	100 MB
io.sortrecord.percent	5% del valor de io.sort.mb
io.sort.spill.percent	80% del valor de io.sort.mb
mapred.job.shuffle.input.buffer.percent	70% del tamaño del heap
mapred.job.shuffle.merge.percent	66% del tamaño del buffer
mapred.inmem.merge.threshold	1000

Tabla 2.1 – Valores por defecto de los parámetros de sintonización de Hadoop.

Capítulo 3 -

POLÍTICA DE PLANIFICACIÓN DE TRABAJOS PROPUESTA

En este capítulo presentamos una nueva política de planificación de trabajos nombrada *Shared Input Policy* para *clusters* Hadoop compartidos. *Shared Input Policy* tiene como reto mejorar el tiempo de *makespan* para trabajos intensivos en datos que compartan un mismo conjunto de datos de entrada, y que sean procesados en lote.

El capítulo presenta los problemas a enfrentar cuando Hadoop planifica trabajos con entrada compartida, y cuando gestiona grandes volúmenes de datos de entrada/salida y datos intermedios generados entre las funciones Map y Reduce.

3.1 Archivos de Entrada Compartidos

Hadoop no maneja de manera eficiente aplicaciones que comparten el mismo conjunto de ficheros de entrada. En estos casos el diseño de Hadoop puede provocar accesos repetidos a disco porque Hadoop no realiza lecturas compartidas al disco; o bien puede generar tráfico innecesario en red porque la política de distribución de bloques de entrada de Hadoop genera desbalanceo de carga. En este trabajo de investigación nos centramos en el problema generado por el desbalanceo de carga entre los nodos del *cluster*.

Cuando un archivo es cargado en HDFS, esto archivo es dividido en bloques con tamaños predefinidos (64 MB por defecto) y distribuidos entre los nodos del *cluster*. Entonces *NameNode* mantiene los metadatos del archivo y la localización de cada bloque en el *cluster* como muestra la Figura3.1.

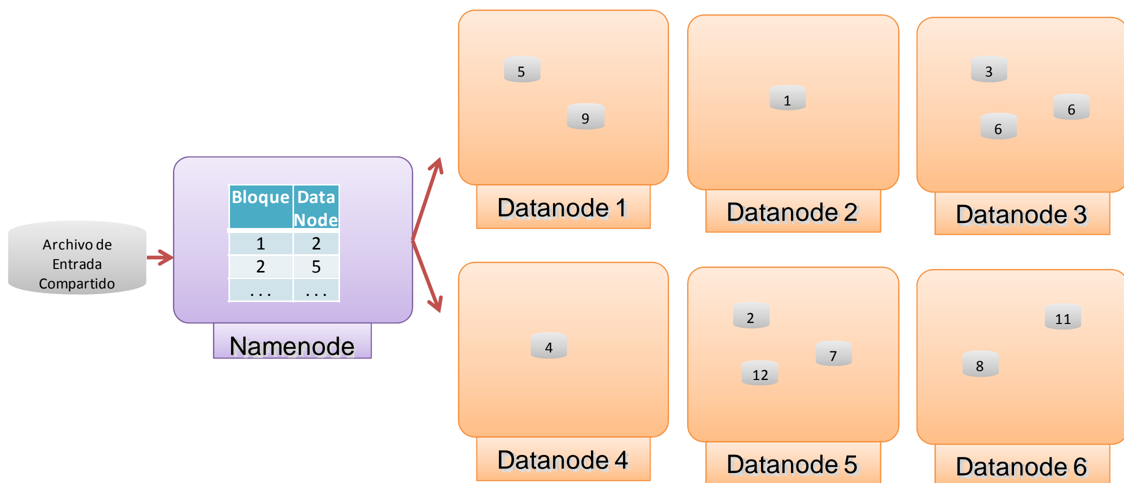

Figura3.1 - Distribución aleatoria de bloques de entrada en HDFS.

Sin embargo, la política de distribución de bloques implementada por defecto en Hadoop es aleatoria. La distribución de los bloques responde a solicitudes de nuevos bloques enviadas por cada uno de los nodos. La imprevisibilidad del orden de llegada de pedidos de nuevos bloques genera una distribución irregular de los mismos, dando lugar a nodos con un número de bloques diferente.

Cuando los trabajos que comparten el archivo de entrada son enviados al *cluster*, Hadoop hace la división de cada trabajo en tareas Map y Reduce y asigna cada tarea Map a su respectivo bloque de entrada. Como muestra la Figura 3.2.

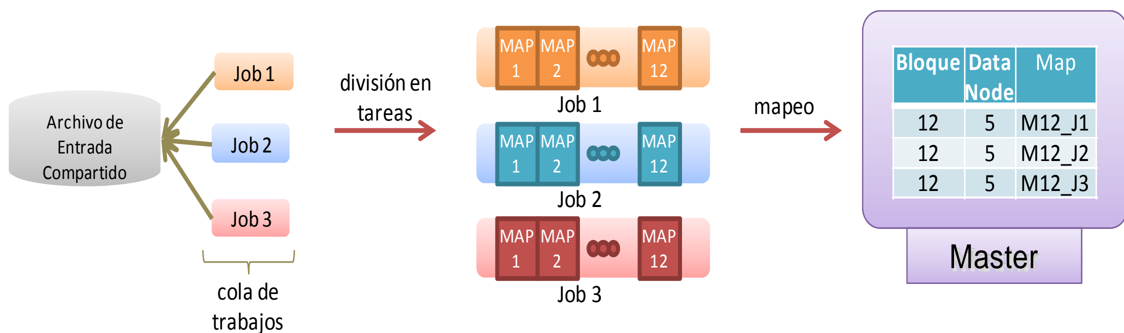

Figura 3.2 - Mapeo de la correspondencia entre bloques y tareas Map.

Durante la ejecución de los trabajos, Hadoop intenta garantizar la localidad de datos para las tareas Map. Esto quiere decir que Hadoop cuando planifica las tareas Map, intenta asignarlas en los mismos nodos donde están localizados los bloques de datos que les corresponde a procesar. Sin embargo hay un desbalanceo de carga en los *cluster* Hadoop producido por la política utilizada para distribuir los bloques entre los nodos. Puede ocurrir que, cuando un nodo termine de procesar todos sus bloques, todavía queden bloques para ser procesados en otros nodos. En estos casos, Hadoop hace una copia del bloque desde el nodo donde esté ubicado hasta el nodo que está libre, como muestra la Figura 3.3 en que el bloque 12 ubicado en el nodo 5 fue copiado hacia el nodo 2 para ser procesado por la tarea Map.

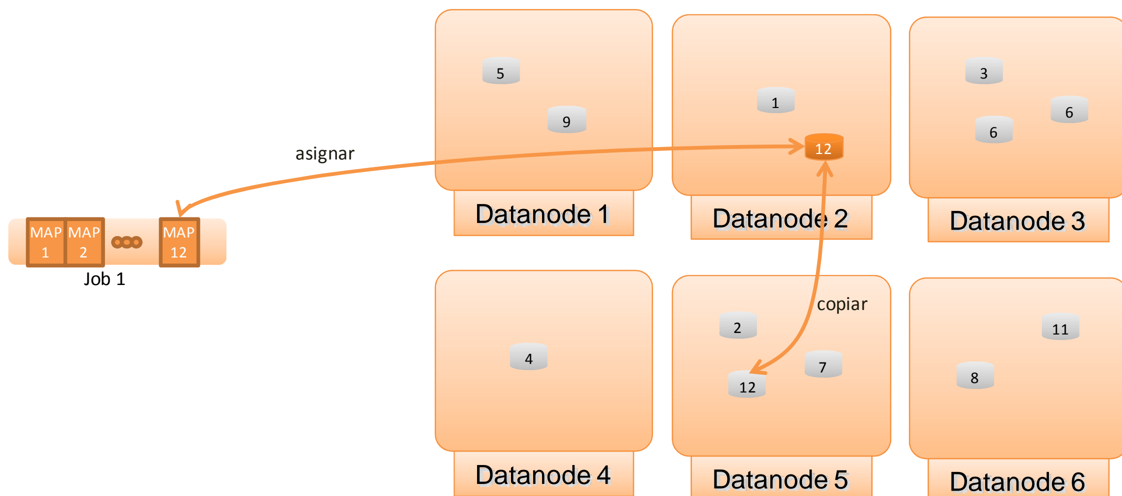

Figura 3.3 - Transferencia de bloques entre nodos cuando no se puede garantizar localidad de datos.

Cuanto más grande es el desbalanceo de carga en el *cluster,* más bloques necesitan ser copiados entre los nodos. Este problema se agrava cuando hay en la cola muchos trabajos que comparten el mismo conjunto de entrada. En estos casos puede ser imprescindible transferir el mismo bloque hacia distintos nodos, como muestra la Figura 3.4. Cuando se ejecutan estos trabajos en los entornos del tipo *cluster* Hadoop compartido, o en entornos en los que hay limitaciones de almacenamiento, el problema se agrava porque en estos casos no se puede disponer de réplicas para los bloques de entrada. La necesidad de transferir repetidas veces el mismo bloque de

datos a distintos nodos genera tráfico en la red, que además de no ser deseado es innecesario.

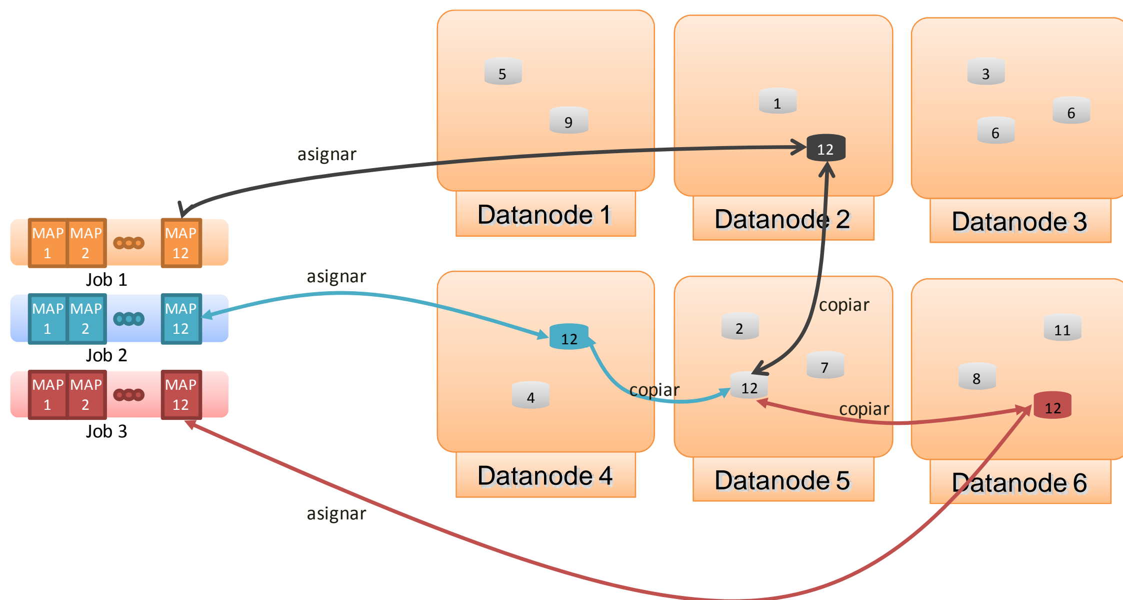

Figura 3.4 - Planificación de tareas con bloques de entrada compartidos sin localidad de datos.

3.2 Gestión de datos de entrada y salida en Hadoop

Los datos de entrada y salida son almacenadas en HDFS y son gestionados por el demonio *NameNode* en el nodo *master* y por los demonios *DataNode* en los nodos *workers*. *NameNode* mantiene la estructura de directorios de HDFS y las metasinformaciones de los archivos en el *namespace*. Después de que cada archivo es dividido en bloques y estos son distribuidos entre los nodos, *NameNode* también mantiene la localización de cada uno de estos bloques en el *cluster*.

La Figura 3.5 representa el proceso de lectura de un bloque de entrada en un *cluster* Hadoop. Cuando una tarea Hadoop necesita leer un bloque, en primer lugar solicita a *NameNode* la localización del bloque en el *cluster*, como muestra el paso 1 de la Figura 3.5. A continuación, conociendo la ubicación del bloque, el nodo donde se ejecuta la tarea define un *stream* de entrada para conectarse al archivo que necesita leer, paso 2 de la Figura 3.5. Si el bloque está localizado en el mismo nodo donde se

ejecuta la tarea, entonces la lectura ocurrirá desde el disco del propio nodo. Si el bloque está localizado en otro nodo, entonces los datos son enviados por *streaming* desde el nodo donde están ubicados hacia el nodo donde se ejecuta la tarea, a través de repetidas lecturas en el *stream* de entrada, paso 3 de la Figura 3.5. Cuando se alcanza el final del bloque entonces se cierra la conexión entre los nodos, paso 4 de la Figura 3.5 .

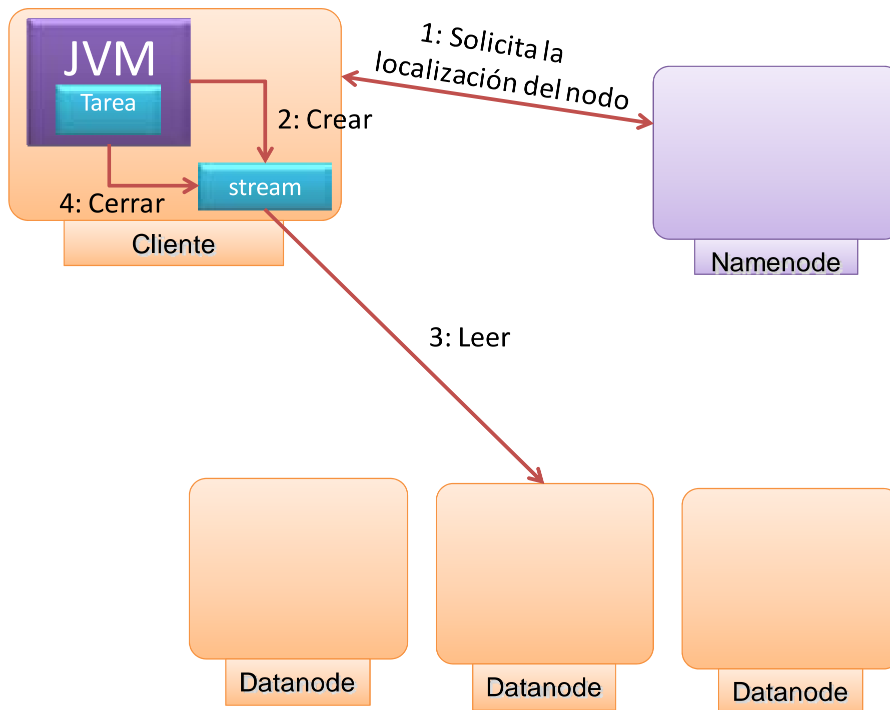

Figura 3.5 - Lectura de datos en HDFS.

La Figura 3.6 representa el proceso de creación de un bloque para escritura en un *cluster* Hadoop. Cuando una tarea Hadoop necesita crear un archivo para escribir su salida, en primer lugar solicita a *NameNode* añadir el archivo en el *namespace*, como muestra el paso 1 de la Figura 3.6. Entonces *NameNode* realiza una serie de comprobaciones para verificar si el archivo ya existe, o bien si el usuario tiene permiso para crear archivos. Si el resultado de estas verificaciones es negativo entonces se genera una excepción, y en caso contrario el *NameNode* envía la dirección de los nodos donde se pueden almacenar los bloques de datos. Cuando el nodo donde se ejecuta la tarea recibe la localización de los nodos donde se podrán almacenar los

datos, entonces crea un *stream* de salida para conectarse con el nodo donde pondrá el primer bloque de datos, paso 2 de la Figura 3.6. Los datos son mantenidos en una cola mientras son enviados por *streaming* hacia el nodo donde serán almacenados, paso 3 de la Figura 3.6. Cuando finaliza el proceso de escritura, la conexión entre los nodos es finalizada y el bloque es cerrado, paso 4 de la Figura 3.6.

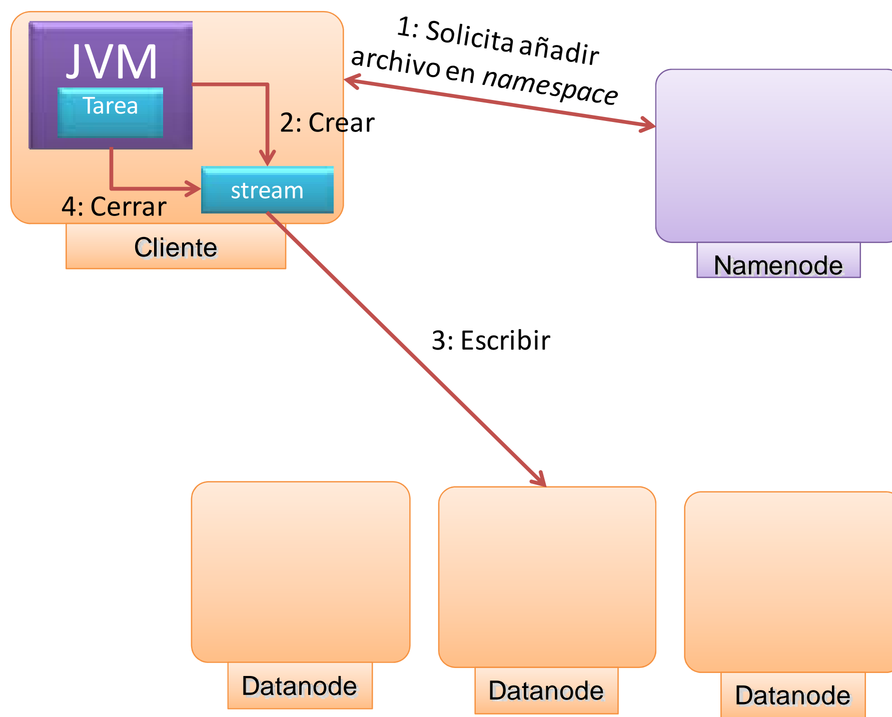

Figura 3.6 - Escritura de datos en HDFS.

Un aspecto importante de este diseño es que el nodo que necesita los datos accede directamente al nodo que los almacena, siguiendo las orientaciones de localización ofrecidas por el *NameNode*. Esta característica permite que HDFS escale el número de clientes concurrentes una vez que el tráfico de datos queda distribuido entre los *DataNodes*. Sin embargo, este modelo de acceso a datos prioriza el *throughput* en lugar de la latencia.

Aunque HDFS gestione los datos de entrada y salida, estos necesitan ser leídos o escritos en el disco duro de cada nodo. Aplicaciones que leen muchos datos de entrada o que generen muchos datos de salida utilizan de forma intensiva el disco; y en *clusters* compartidos el disco se convierte en un cuello de botella para el sistema.

3.3 *Gestión de datos intermedios en Hadoop*

Hadoop no utiliza HDFS para almacenar los datos intermedios generados entre las fases Map y Reduce. Los datos intermedios son guardados en el sistema de archivos local de cada nodo. Hadoop gestiona los datos intermedios a través de *buffers* creados en memoria principal y vuelca estos datos en el disco duro cuando los *buffers* están llenos. Los *buffers* son creados tanto en los nodos donde se ejecutan las tareas Map como en los nodos donde se ejecutan las tareas Reduce.

La salida de las tareas Map se almacenan primero en un *buffer* de memoria circular, como muestra la Figura 3.7. El tamaño de este *buffer* está definido por la variable *io.sort.mb* en un archivo de configuración. En una parte del *buffer* se almacenan los metadatos, y en la otra parte se almacenan las tuplas [clave, valor]. Para impedir que el *buffer* se llene, se define un determinado umbral por la variable *io.sort.spill.percent* a partir del cual Hadoop comienza a cargar el contenido del *buffer* al sistema de archivos local. Como la salida de cada Map es enviada para diferentes tareas Reduce, antes de volcar los datos en el disco duro, Hadoop primero divide el contenido del *buffer* en un número de particiones igual al número de tareas Reduce. Cada partición de la salida de Map se enviará a una tarea Reduce distinta. Una vez que los datos estén particionados, Hadoop los ordena en memoria y entonces los vuelca al disco duro.

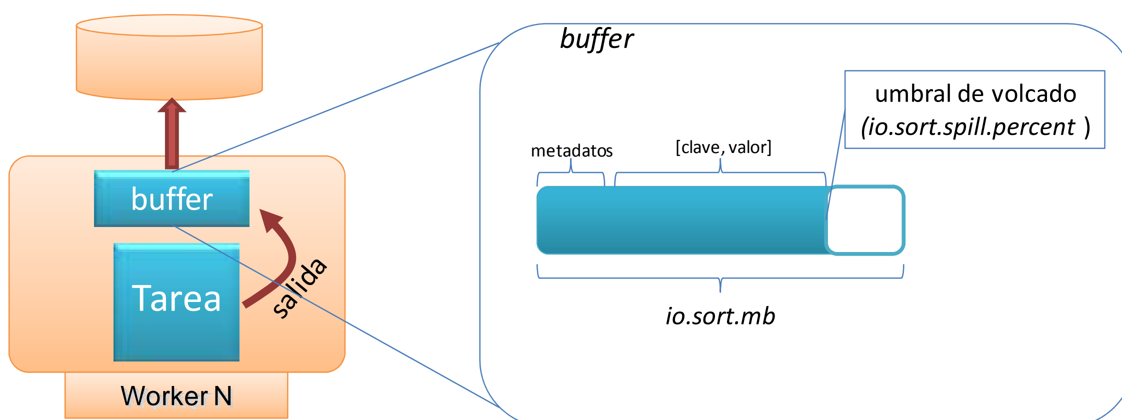

Figura 3.7 - Buffer temporal para recibir la salida de Map.

La Figura 3.8 muestra como Hadoop crea el archivo de salida de la tarea Map. Cada vez que se alcanza el umbral del *buffer*, un nuevo archivo de volcado es creado en el disco. Cuando la tarea Map termina, todos los archivos que fueron volcados al disco son mezclados por Hadoop en un único archivo ordenado y particionado en el disco duro.

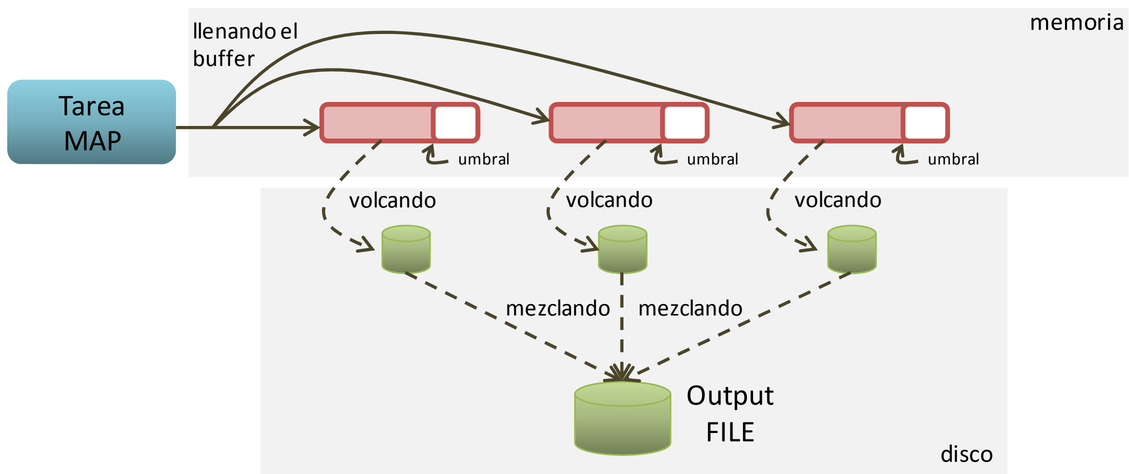

Figura 3.8 - Gestión de la salida Map - cargar los datos en el buffer y posteriormente cargarlos en disco.

A continuación, los datos de cada partición generada en la fase Map, deben ser enviados a los nodos donde serán procesados por las tareas Reduce. Hadoop también crea un *buffer* para recibir estos datos en cada nodo donde será ejecutada una tarea Reduce. El tamaño del *buffer* está definido en un archivo de configuración por la variable *mapred.job.shuffle.input.buffer.percent*.

Cada tarea Reduce recibe la salida de varias tareas Map simultáneamente en un único *buffer*. La Figura 3.9 muestra como el nodo Reduce recibe datos desde los nodos donde se ejecutaron las tareas Map. Cuando el nodo Reduce comienza a recibir la salida de cada tarea Map, Hadoop evalúa si los datos caben en el *buffer* que se creó para almacenarlos. En caso negativo, los datos son copiados directamente al sistema de archivos local. Y en caso afirmativo empiezan a ser copiados para el *buffer*. Para no permitir que el *buffer* se llene, también se define un umbral para el *buffer* (*mapred.inmem.merge.threshold*) en un archivo de configuración. Cuando se alcanza el

umbral definido, los datos que están en el *buffer* son mezclados y volcados en el sistema de archivos local. Cada volcado genera un archivo de datos temporal.

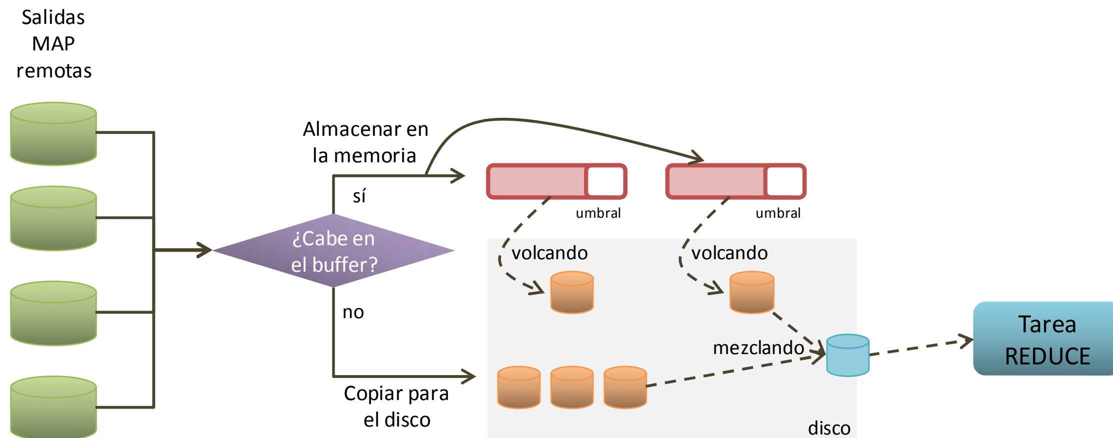

Figura 3.9 -Gestión de la entrada Reduce- recepción de datos directamente en el disco o en buffers intermedios y cargarlos en el disco.

Después de que los datos generados por todas las tareas Map fueron copiados para el nodo donde se ejecutará la tarea Reduce, Hadoop necesita mezclarlos antes de que la tarea Reduce empiece a procesarlos. Esta mezcla se produce en rondas hasta que los datos estén listos para ser procesados por la tarea Reduce. En cada ronda son mezclados una cantidad específica de archivos temporales. La cantidad de archivos mezclados en cada ronda está definida en la variable *io.sort.factor* en un archivo de configuración.

Aplicaciones que manejan grandes volúmenes de datos intermedios vuelcan muchos datos en el disco duro, tanto en la fase Map cuanto en la fase Reduce. Cuando esto ocurre, el disco se sobrecarga. El aumento de tamaño de los *buffers* no es una solución para aplicaciones que generan muchos datos intermedios porque como mínimo, se necesita almacenar una vez la salida de tarea Map, incluso cuando los *buffers* son lo suficientemente grandes para mantener todos los datos en la memoria.

3.4 SHARED INPUT POLICY

Shared Input Policy busca aprovechar el hecho, de que distintos trabajos compartan el mismo archivo de entrada para mejorar la localidad de datos de las

tareas Map. *Shared Input Policy* permite que las tareas Map que compartan un mismo bloque de entrada se puedan planificar en el mismo nodo de cómputo; incluso siendo tareas Map de diferentes trabajos. Esto asegura una localidad de datos más alta que la obtenida cuando se planifican los trabajos utilizando las políticas de planificación de trabajos tradicionales utilizadas en Hadoop.

Shared Input Policy trabaja en dos niveles: en un nivel macro (nivel de planificación de trabajos) y en un nivel micro (nivel de planificación de tareas).

En el nivel macro, *Shared Input Policy* coloca en un mismo grupo los trabajos que comparten los mismos archivos de entrada y los procesa en lotes, como muestra la Figura 3.10. El modelo de llegada de trabajos fue definido como un proceso estacionario, y cada lote tiene un número fijo de trabajos. El procesamiento por lotes es utilizado, por ejemplo, en aplicaciones de bioinformática de alineación genómica donde el mismo conjunto de datos de referencia es compartido por muchos trabajos diferentes. Los bioinformáticos construyen lotes de procesamiento formados por un archivo de referencia y por un número fijo de consultas con archivos proporcionados por el secuenciador genómico, y envían estos lotes al *cluster*.

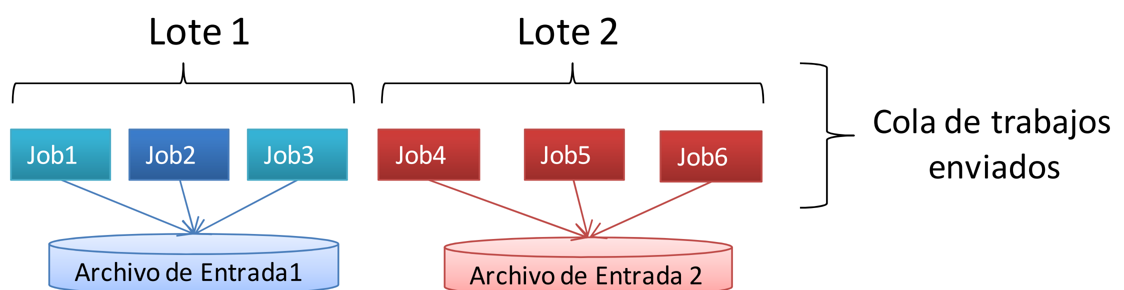

Figura 3.10 - Cola de trabajos con conjunto de entrada compartida.

En el nivel micro (nivel de planificación de tareas) las tareas de los diferentes trabajos procesados en el mismo lote, y que manejan los mismos bloques de datos de entrada, se agrupan para ser ejecutadas en el mismo nodo. Cuando un nodo solicita una nueva tarea, la política busca una tarea que maneje el mismo bloque de datos utilizado por la tarea que acaba de terminar. La búsqueda de las tareas que se asignarán a un nodo se produce en diferentes trabajos del mismo lote. La política

también se asegura de que una tarea podría ser asignada a un nodo que no tiene bloque de datos almacenado para ser procesado, si no hay otras tareas en el lote esperando para ser planificadas.

Shared Input Policy sigue un mecanismo simple como muestra la Figura 3.11. Cuando *TaskTracker* tiene un slot libre para comenzar una tarea, envía una solicitud al demonio maestro (*JobTracker*), informando sobre su estado de disponibilidad. Posteriormente, *JobTracker* busca entre las tareas que están esperando a ser ejecutadas (incluso las tareas de los diferentes trabajos desde que sean del mismo lote) una que maneje el mismo bloque de datos que ya se ha utilizado en este nodo. Si el *JobTracker* encuentra esta tarea será enviada al nodo, manteniendo la localidad de datos, como muestra la Figura 3.11.

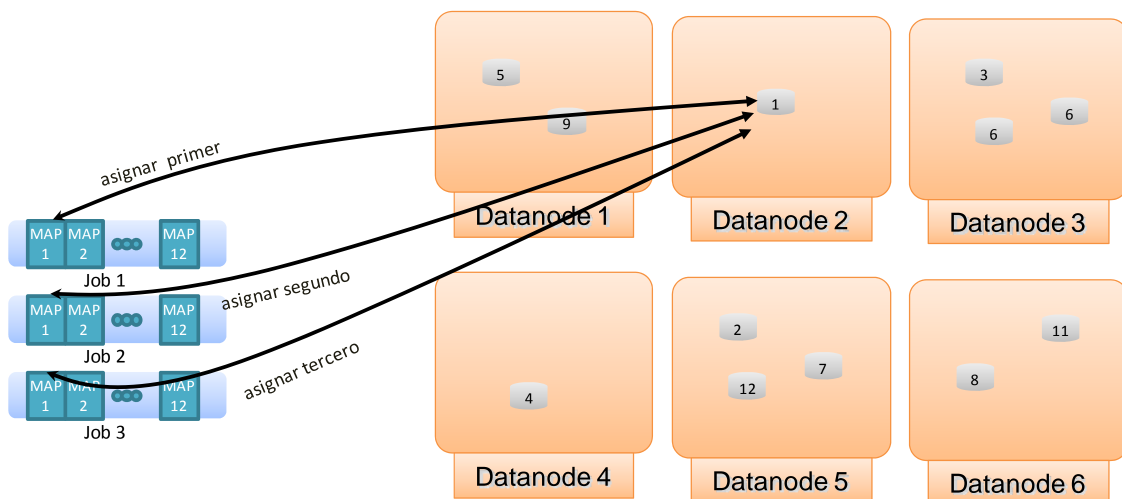

Figura 3.11 - Asignación de tareas Map utilizando Shared Input Policy con localidad de datos.

En el caso de que el planificador no encuentre una tarea pendiente en la cola de tareas de todos los trabajos del lote que se esté ejecutando, entonces el *JobTracker* busca una tarea que procese un bloque almacenado en el mismo nodo que solicitó una nueva tarea. Si el *JobTracker* no encuentra ninguna tarea, entonces se enviará la primera tarea pendiente a realizar, como muestra la Figura 3.12, suponiendo que la próxima tarea a ser ejecutada es la tarea Map 12.

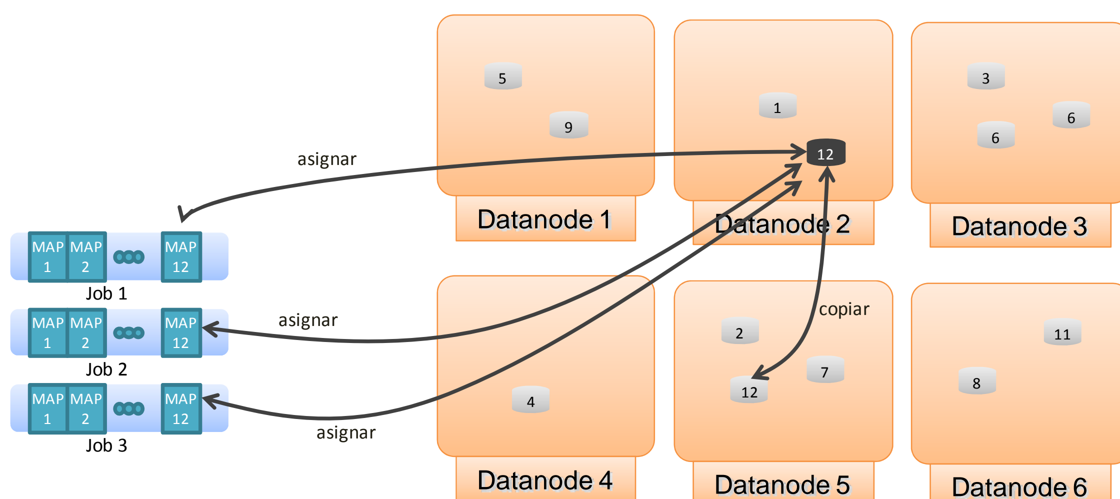

Figura 3.12 - Asignación de tareas Map utilizando Shared Input Policy sin localidad de datos.

3.5 RAMDISK para gestión de datos intermedios

El acceso al disco es un cuello de botella para aplicaciones intensivas de datos Hadoop, especialmente para aquellas aplicaciones que generan muchos datos intermedios. El modelo de datos implementado en el *framewok* Hadoop requiere que los datos intermedios se almacenen en el sistema de archivos local. Hadoop debe volcar en el disco una gran cantidad de datos cuando el volumen de datos intermedio es muy alto. Tanto en los nodos donde se generan los datos intermedios (nodos Map), como en los nodos en los que se procesan los datos intermedios generados por la fase Map (nodos Reduce). Por lo tanto, las sobrecargas de acceso a disco provocadas por su uso intensivo, causan elevados tiempos de espera por los datos de entrada/salida.

El objetivo de nuestra propuesta es mejorar los resultados obtenidos a partir de *Shared Input Policy* reduciendo el costo de acceso al sistema de entrada y salida de datos intermedios, generados por las aplicaciones Hadoop. Para lograr este objetivo nos propusimos introducir una RAMDISK para el almacenamiento temporal de los datos. La RAMDISK fue creada como una capa entre los *buffers* intermedios generados por Hadoop y el disco duro. RAMDISK es una región de la memoria RAM que simula una unidad de disco real. Básicamente consiste en reservar cierta cantidad de memoria y permitir que el sistema operativo la utilice como una partición más de disco. Como la

velocidad de acceso a la memoria RAM es más alta que la velocidad para acceder al disco duro en algunos órdenes de magnitud, el uso de la RAMDISK permite que Hadoop pueda volcar los datos de una forma más rápida.

En nuestra propuesta, utilizamos la RAMDISK para volcar datos intermedios tanto en los nodos donde se ejecutan las tareas Map como en los nodos donde se ejecutan las tareas Reduce. Los *buffers* creados por Hadoop siguen existiendo. Sin embargo, en nuestra propuesta Hadoop vuelca los datos intermedios desde los *buffers* para la RAMDISK cuando estos *buffers* alcanzan sus límites de umbral definidos por defecto. La gestión del entorno se simplifica porque las predicciones no son necesarias ni tampoco cambios en los parámetros de configuración de los buffers. Además, las mismas funciones como las de lectura, escritura, apertura y cierre de archivos que se utilizan en un disco duro tradicional se pueden utilizar con RAMDISK, sin necesidad de efectuar cambios en el código de la aplicación. Por otro lado, la RAMDISK tiene un tamaño limitado y no hay persistencia de sus datos. Por lo tanto, estas características requieren la definición de una política de sustitución de datos y la gestión de sus contenidos para el caso de fallos.

Construimos el almacenamiento temporal en la RAM usando *tmpfs* (temporary file storage), un sistema de archivos de Linux para crear particiones temporales del tipo *RAM Driver*. Se especifica un límite de tamaño en *tmpfs* como en un sistema de archivos tradicional del disco que también está limitado por su capacidad. Si el límite definido para la RAMDISK es alcanzado, se puede configurar *tmpfs* para enviar un mensaje de error de "disco lleno", como en un sistema de archivos tradicional. Sin embarge el límite de tamaño de la RAMDISK se puede cambiar de forma dinámica cuando hay más (o menos) necesidades de almacenamiento. Se puede gestionar el tamaño y la cantidad de espacio utilizada por la partición *tmpfs* de manera sencilla a través del comando Linux **df**. Otra ventaja adicional es que *tmpfs* puede utilizar SWAP cuando el sistema se ejecuta fuera del espacio físico de la RAM. Cuando esto ocurre, los archivos que están en la partición *tmpfs* son escritos a disco en particiones SWAP y serán leídos desde el disco cuando sean accedidos posteriormente. La principal desventaja de *tmpfs* es que el sistema de RAMDISK está montado en un sistema volátil de memoria. Así, como no hay persistencia de datos, es necesarios definir una política de resguardo para los datos almacenados temporalmente en la partición.

Aunque el objetivo principal de la RAMDISK en nuestra propuesta fue mejorar el tiempo de acceso y el rendimiento de los datos intermedios, también separamos un espacio de la RAMDISK para almacenar los archivos de entrada y de salida de los

trabajos ejecutados en el *cluster* Hadoop. En entornos Hadoop compartidos el acceso al disco es un cuello de botella, y el uso de la RAMDISK para almacenar temporalmente los archivos de entrada y salida, disminuye la sobrecarga en el sistema de entrada/salida.

Los demás espacios de la RAMDISK están dedicados a almacenar datos intermedios volcados por los *buffers* creados por Hadoop. Como la RAMDISK tiene un tamaño limitado, tenemos que administrar su contenido. Por eso, hemos creado dos umbrales: uno *soft* y otro *hard*, como se puede apreciar en la Figura 3.13. Si se alcanza el límite *soft*, las nuevas tareas asignadas al nodo pasan a volcar los datos directamente al disco sin usar la RAMDISK; como ocurre en la configuración por defecto de Hadoop. Sin embargo, si se alcanza el límite *hard*, los datos que están colocados en RAMDISK son volcados en el disco duro bajo una política de sustitución de datos. La Figura 3.13 describe el funcionamiento de la RAMDISK en el sistema. El tamaño de la RAMDISK y los límites son parámetros configurables de la política y sus valores dependen de los recursos disponibles en cada nodo y de los requerimientos de las aplicaciones que se ejecutan.

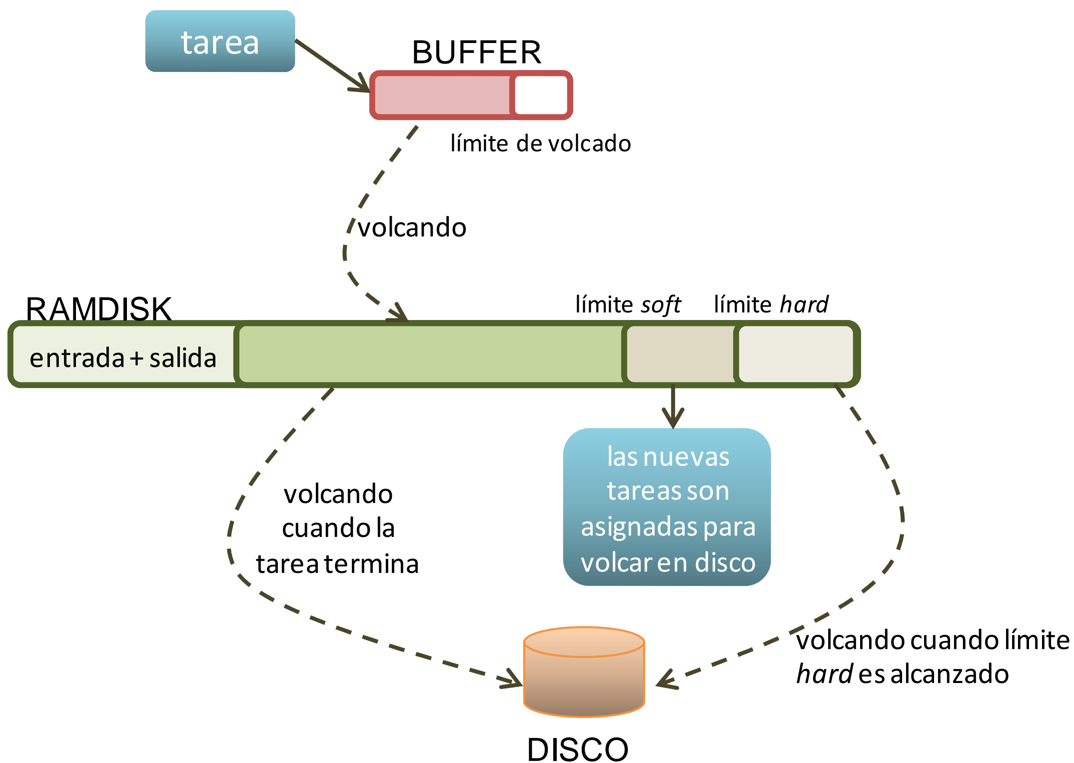

Figura 3.13 - gestión de datos intermedios utilizando RAMDISK.

3.6 Implementación

En este apartado presentamos los puntos principales de la implementación de *Shared Input Policy* a través de seudocódigo.

Al inicio del procesamiento de cada lote, transferimos los bloques de entrada, que se procesarán durante la fase Map desde el disco local hacia la RAMDISK. Entonces, nuestra política de planificación de tareas – *Shared Input Policy* - comienza a realizar dinámicamente la planificación de las tareas Map y Reduce.

Los principales enfoques de la política propuesta son compensar el desbalanceo de carga entre los nodos del *cluster* Hadoop, provocado por la distribución aleatoria de los bloques de entrada de datos; y disminuir la sobrecarga en los discos de cada nodo cuando manejan aplicaciones intensivas en datos. La política intenta planificar las tareas Map de manera que aumente la localidad de datos, y por lo tanto, disminuya el volumen de datos transferido por la red. Y, por otra parte, utilizar una RAMDISK para almacenamiento temporal de datos.

Shared Input Policy sigue el proceso que se muestra en la Figura 3.14. Cuando un demonio de gestión de tareas del nodo de cómputo (*TaskTracker*) tiene un slot libre para comenzar una tarea, envía una solicitud al demonio maestro (*JobTracker)*, informando sobre su nuevo estado. Posteriormente, *JobTracker* busca entre las tareas que están esperando a ser ejecutadas (incluso las tareas de los diferentes trabajos desde que sean del mismo lote) una que maneje el mismo bloque de datos que ya se ha utilizado en este nodo. Si el *JobTracker* encuentra esta tarea, ella será enviada al nodo, manteniendo la localidad de datos, líneas 4-9. De lo contrario, *JobTracker* busca una tarea que procesa un bloque almacenado en el mismo nodo que solicitó una nueva tarea, líneas 10-13. Si el *JobTracker* no encuentra ninguna tarea, entonces se enviará la primera tarea pendiente a realizar, líneas 15-18.

Después de seleccionar una tarea por el *JobTracker*, Hadoop la asigna para un nodo. Cuando una tarea se asigna a un nodo, nuestra política evalúa el espacio disponible en la RAMDISK. Como la RAMDISK tiene un tamaño limitado existe la posibilidad de que lo datos intermedios excedan este tamaño. Por lo tanto *Shared*

Input Policy establece previamente un límite *soft* y un límite *hard* para la RAMDISK. Si aún no se ha alcanzado el límite *soft,* la tarea es asignada para volcar los datos intermedios en la RAMDISK. La tarea es asignada para volcar los datos en el disco duro si ya se ha alcanzado el límite *soft*. Como muestran las líneas 22-26 del algoritmo de asignación de tareas que se muestra en la Figura 3.14

Shared Input Policy también define un límite *hard* para aquellas tareas que vuelcan sus datos en la RAMDISK. Las tareas que están volcando sus datos en la RAMDISK pasan a volcar sus datos en el disco duro cuando se supera este límite *hard*. Primero transferimos la última tarea Reduce que se ha asignado a volcar sus datos intermedios a RAMDISK. Si todavía estamos por encima del límite *hard*, seguimos transfiriendo tareas Reduce de la última a la primera tarea asignada, una por una. Si después de transferir todas las tareas Reduce, el límite *hard* de la RAMDISK aún está siendo superado entonces repetimos el procedimiento para las tareas Map. Como podemos observar en las líneas 34-40 del algoritmo de asignación de tareas y el volcado de datos mostrado en la Figura 3.14.

Definimos como la política de sustitución de datos para *Shared Input Policy* que los archivos temporales y de salida se guardan en la RAMDISK hasta la finalización de cada trabajo. Cualquier archivo de entrada compartido entre los trabajos de un mismo lote se mantiene en la RAMDSISK hasta el final de procesamiento del lote.

```
00  //selecting task that will be assigned
01  for each event scheduling of taskTracker:
02      while jobs_in_the_queue do
03              //blockID  has the last block ID  processed by the taskTracker
04               if (blockID != NULL)
05                      if (obtainNewLocalTask(taskTracker, blockID) != NULL)
06                          select(task);
07                          break;
08                       end_if
09                 end_if
10                 if (obtainNewLocalTask(taskTracker) != NULL)
11                       select(task);
12                       break;
13                 end_if
14                  else
15                         if(obtainNewNoLocalTask(taskTracker) != NULL)
16                             select(task);
17                             break;
18                       end_if
19               end_else
20
21            //task has the task ID selected by Job Scheduler and need to alocated
22           if soft limit is not reached
23                   allocate task to spill data to RAMDISK
24           else
25                   allocate task to spill data to hard disk
26           end if
27     end while
28  end for
29 //defining where data will be spilled
30  while task generate data
31      if task was allocated on the hard disk
32             spill data to disk
33      else
34             while hard limit reached
35                    if there is Reduce task allocated to RAMDISK
36                            transfer latest Reduce task for hard disk
37                    else
38                            transfer latest Map task for hard disk
39                   end if
40              end while
41              if task was transferred to hard disk
42                     spill data to disk
43              else
44                     spill data to RAMDISK
45             end if
46       end if
47   end while
```

Figura 3.14 - Pseudo-código de Shared Input Policy.

La persistencia de los datos intermedios sigue los mismos principios estándar de Hadoop de cuando los datos se descargan directamente en el disco local. Si el nodo que procesa la fase Reduce cae, los datos se envían de nuevo desde el nodo en el que han sido generados por la tarea Map. Si el nodo que almacena la salida de la tarea Map cae antes de que los datos sean enviados por completo a los nodos donde serán procesados a continuación por la fase Reduce, entonces la tarea Map debe ser procesada una vez más en otro nodo.

Capítulo 4 -

EXPERIMENTACÍON REALIZADA Y RESULTADOS

En este capítulo presentamos los experimentos realizados y los resultados obtenidos por *Shared Input Policy* cuando son comparados con otras políticas de planificación de trabajos utilizadas en *clusters* del tipo Hadoop. Presentamos el entorno de experimentación utilizado y hacemos una descripción de las aplicaciones utilizadas en los experimentos. Los resultados muestran como *Shared Input Policy* incrementa la localidad de datos para las tareas Map, y como mejora el tiempo de *makespan* cuando planifica lotes de trabajos que comparten el mismo conjunto de ficheros de entrada. Comparamos *Shared Input Policy* con FIFO que es la política de planificación por defecto de Hadoop y con otras dos políticas de planificación de trabajos para entornos compartidos: *Fair Scheduler* y *Capacity Scheduler*. Monitorizamos los recursos del *cluster* y evaluamos como el uso de la RAMDISK influye en el tiempo de ejecución de las aplicaciones, y comparamos los resultados obtenidos con la alternativa de mantener los datos intermedios en memoria aumentando el tamaño de los *buffers*.

4.1 Entorno de experimentación

Para llevar a cabo los experimentos utilizamos un *cluster* Hadoop que consiste en máquinas de doble núcleo interconectados por una red Ethernet Gigabit. Este *cluster* está compuesto por 16 máquinas homogéneas. Cada máquina está configurada con dos procesadores de doble núcleo Intel Xeon CPU 5160, 3,0 GHz de frecuencia, memoria DRAM con 12 GB de capacidad y 160 GB de disco duro. El *cluster* tiene un nodo de entrada (*front end*) para enviar trabajos, configurado con dos procesadores CPU Intel Xeon 5160, de 2,66 GHz, 8 GB de memoria DRAM y 1,8 TB de disco duro.

Utilizamos el nodo de entrada para asignar un trabajo que monta el *cluster* Hadoop y su sistema de archivo distribuido en los 16 nodos. Utilizamos la distribución Apache de Hadoop en su versión 1.2.1

Desde la perspectiva de Hadoop, uno de los 16 nodos del *cluster* realiza al mismo tiempo las tareas de *master* y las tareas de *worker*. En este nodo se ejecutan los demonios *jobtracker* y *NameNode* que tienen las tareas de gestión del *master* además de los demonios *tasktraker* y *DataNode* con las tareas correspondientes a uno nodo *worker*.

Los trabajos Hadoop son lanzados en el *cluster* a través del nodo de entrada donde cada usuario Hadoop se conecta. La Figura 4.1 presenta esquemáticamente el *cluster* utilizado para la experimentación. Cada nodo se ha configurado con 4 slots para las tareas Map y 2 slots para las tareas Reduce. Los valores de los parámetros de configuración utilizados por Hadoop se definieron a partir de Starfish [57] o a través de experimentación y están descritos antes de cada experimentación.

Para la evaluación de *Shared Input Policy*, montamos en cada nodo del *cluster* una RAMDISK de 6 GB, haciendo un total de 96 GB para el almacenamiento. Una tercera parte de este espacio de almacenamiento fue reservada para los archivos de entrada/salida y el restante para los datos intermedios. Las tareas asignadas a cada nodo vuelcan sus datos intermedios desde los buffer creados por Hadoop hacia la RAMDISK. Como el tamaño de la RAMDISK es limitado, establecimos un límite *soft* de 70% a partir del cual las nuevas tareas asignadas a los nodos volcaban los datos directamente en el disco duro, tal como se establece en la configuración por defecto de Hadoop. Y, definimos como límite *hard* el valor de 90% a partir del cual los datos que están en la RAMDISK son volcados en el disco duro. El tamaño de la RAMDISK, y los valores del límite *soft* y del límite *hard* son configurables. Para este apartado del documento son presentados los valores que obtuvieron mejores prestaciones durante la experimentación realizada.

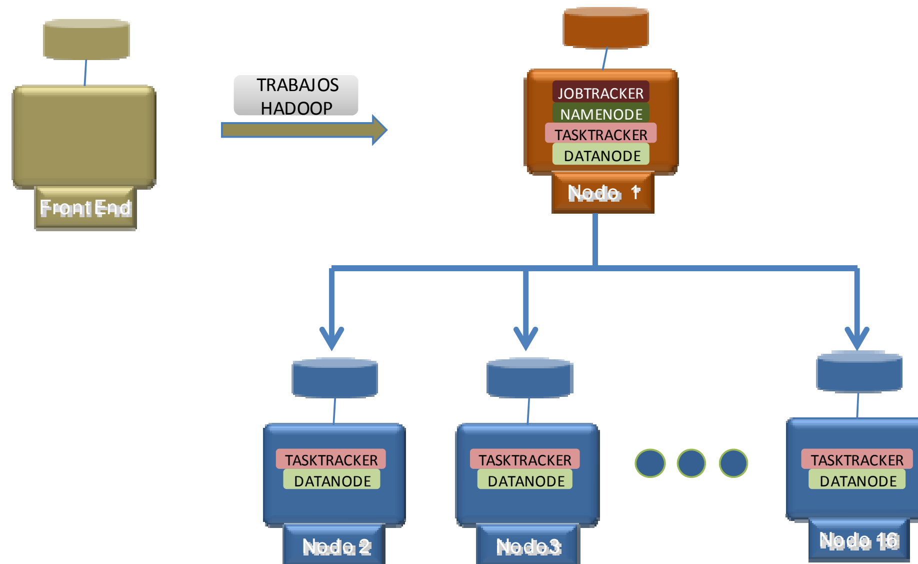

Figura 4.1 - Esquema del cluster utilizado en las experimentaciones.

Hadoop se utilizó en la distribución Apache 1.2.1, Java en la versión jdk1.6.0_18 y el sistema operativo usado ha sido el Linux con la versión del *kernel* 2.6.16.46-0.12.

4.2 Aplicaciones

Para evaluar nuestra política de planificación de trabajos utilizamos un conjunto de aplicaciones intensivas en datos: Mapreduce Mapping and Assembly with Quality (MrMAQ) [20] que es una aplicación MapReduce de Bioinformática diseñada para alinear un conjunto de secuencias de ADN del tipo *short read* a un genoma de referencia; y un conjunto de aplicaciones disponibles en el *benchmark* Hibench [59].

MrMAQ es una aplicación de alineamiento genético del tipo *short alignment* desarrollada por nuestro grupo de investigación. MrMAQ busca coincidencias entre un conjunto de entrada de secuencias tipo *short read* y un genoma de referencia utilizando semillas y extendiendo el algoritmo para ambos conjuntos de datos. La fase Map es encargada de generar las semillas para las entradas. Las tareas de la fase Reduce reciben estas semillas; buscan coincidencias entre las secuencias y el genoma de referencia; y posteriormente calculan la calidad de estas coincidencias.

HiBench Benchmark Suite es un *benchmark* desarrollado por Intel y está constituido por un conjunto de aplicaciones de diferentes características. Evaluamos

nuestra política de planificación de trabajos utilizando las aplicaciones intensivas en datos: Sort, WordCount y K-means.

4.2.1 Aplicación MrMAQ

MrMAQ está basada en el algoritmo Mapping and Assembly with Quality (MAQ) [60] para alinear las secuencias de ADN del tipo *short read* a un genoma de referencia. Para ello, MAQ crea una tabla hash para almacenar e indexar el conjunto de entrada del tipo *short read*. El algoritmo de alineamiento MAQ se basa en la observación de que un alineamiento completo de un numero n de pares de bases (bp) con el máximo de ***k*** diferencias debe contener al menos un alineamiento exacto de $\frac{n}{k+1}$ bases consecutivas [61]. Es decir, para una lectura con 30 pares de bases (bp) a ser alineada con una secuencia de referencia que tenga una sola discrepancia, debe haber por lo menos 15 bases consecutivas que coinciden exactamente a partir de la posición de la discrepancia encontrada.

MAQ construye seis tablas hash diferentes para indexar los primeros 28 pb de la lectura y para asegurar que los alineamientos con un máximo de dos discrepancia se ven afectados. Estas seis tablas corresponden a seis semillas no contiguas como 11110000, 00001111, 11000011, 00111100, 11001100 y 00110011, si se lee 8 caracteres de longitud.

Después de que las tablas hash de lectura se construyen, la referencia es escaneada en orden natural y en orden inverso. Cada secuencia de referencia escaneada es comparada a los valores que se encuentran en las tablas hash. Si la comparación resulta en un éxito (*hit*), MAQ calcula una puntuación de calidad para esta coincidencia basada en la probabilidad de que la alineación no será válida. Para calcular el nivel de calidad de la coincidencia, MAQ se utiliza de la Ecuación 4.1:

$$Q = \min\left\{\begin{array}{c} q_2 - q_1 - 4{,}343\log n_2, \\ 4 + (3 - k')(q - 14) - 4{,}343\log p_1(3 - k'), \\ 28 \end{array}\right\}$$

Ecuación 4.1 - Utilizada para calcular la calidad de la alineación en MAQ.

Donde q_1 es la suma de los valores de calidad de las discrepancias para el mejor *hit*, q_2 es la suma correspondiente a lo segundo mejor *hit*, n_2 es el número de *hits* con el mismo número de discrepancias que el mejor *hit*, k' es el número mínimo de discrepancias en la semilla con 28bp, y q es el valor promedio de la calidad de la base en la semilla con 28bp. Otros ejemplos de algoritmos de indexación de semillas espaciadas similares como MAQ son SOAP [62] y RMAP [63].

MrMAQ es una aplicación para *clusters* Hadoop que está diseñada bajo el modelo de programación MapReduce, y que está implementada en JAVA. Los principios básicos de la aplicación son los mismos que encontramos en el algoritmo MAQ. Es decir, encontrar coincidencias de una lista de secuencias de ADN del tipo *short read* con un genoma de referencia usando una semilla y un algoritmo extendido.

El diseño de MrMAQ se basa en aplicaciones anteriores de bioinformática como MapReduce Cloudburst [16], Crossbow [64] y MrsRF [65]. En esta implementación particular, las tareas Map leen los archivos del genoma de referencia y de los *short reads* para generar las semillas con 28 bases (bp) de ambos. Las tareas Reduce reciben pares de [clave-valor] con las coincidencias entre las secuencias de referencia y los *short read*, conocidas como éxitos (*hits*). Las tareas Reduce extienden el alineamiento para todos los hits que se encuentran, calculan sus cualidades de alineamiento y devuelven una lista de todos los *short reads*, sus posiciones de alineamiento en el genoma de referencia y sus cualidades de alineamiento.

La fase Map procesa los archivos de consulta con las secuencias de ADN del tipo *short read* y los archivos con los genomas de referencia. Las tareas Map generan su salida como las tablas hash de la aplicación MAQ original. Es decir, crean una nueva clave a partir de cada secuencia del tipo *short read* y almacenan el resultado de la aplicación de cada uno de las seis semillas. A continuación, las tareas Map leen la secuencia del genoma de referencia generando también una nueva clave para cada subsecuencia de la referencia. El resultado de la fase Map es una lista de tuplas [clave, valor] con las semillas que vienen de ambos tipos de entrada, como expresado en la Figura 4.2.

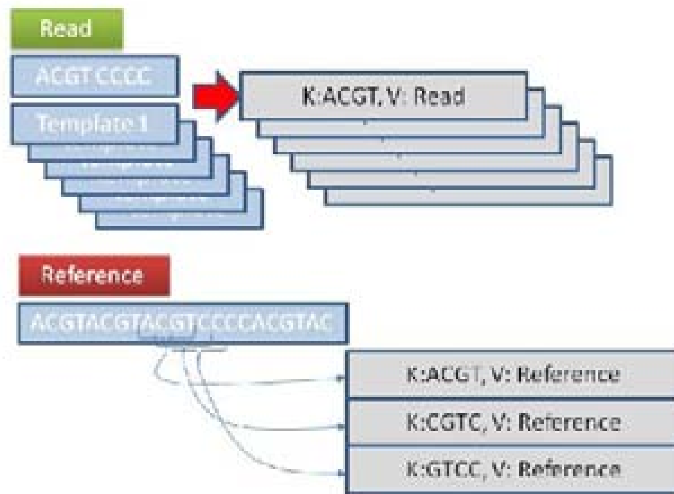

Figura 4.2 - Fase Map de la aplicación MrMAQ.

La fase Reduce hace la extensión del alineamiento. Para cada clave que la tarea Reduce recibe, ella también recibe una lista de todas las secuencias del tipo *short read* y de las referencias que tienen la misma semilla. Para cada una de estas coincidencias, la tarea Reduce extenderá el alineamiento para el resto de la secuencia del tipo *short read* y calculará la suma de las cualidades de bases coincidentes. Entonces se dará salida a los alineamientos de mejor calidad encontrados para cada posición de referencia. La Figura 4.3 muestra el procedimiento de extender el algoritmo de búsqueda de coincidencias.

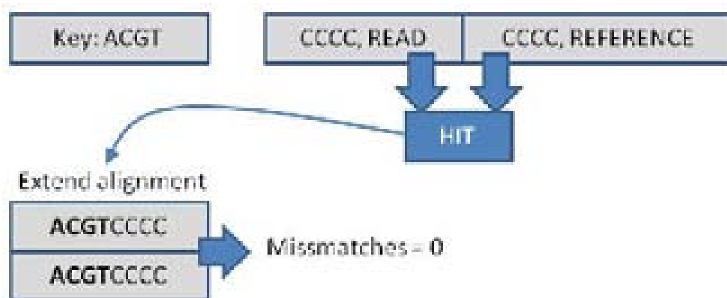

Figura 4.3 - Fase reduce de la aplicación MrMAQ.

4.2.1.1 Caracterización de MrMAQ

Hicimos la caracterización de la aplicación MrMAQ en nuestro entorno de experimentación. Utilizamos archivos de consulta con 5 KB y variamos el tamaño del archivo de referencia: entre 18 MB, 60 MB y 1000MB para caracterizar el consumo de recursos. En cada experimento se hizo coincidir el tamaño del bloque de entrada con el tamaño de los datos de entrada, de modo que cada aplicación tenía un solo bloque y, por lo tanto, una sola tarea Map. También definimos cada trabajo con una sola tarea Reduce. De esta manera no hay concurrencia entre las tareas ejecutadas. Utilizamos un único nodo del *cluster* para ejecutar la aplicación y monitorizamos el uso de los recursos con la herramienta SAR.

En este apartado presentamos los resultados para tres de los experimentos, que en función del volumen de datos generados entre las fases Map y Reduce, manejan de manera distinta los buffers intermedios. Cuando el archivo de referencia es de 18 MB todos los datos intermedios generados caben en los buffers creados por Hadoop. Para el archivo de referencia de 60 MB el buffer para la salida de la tarea Map no es suficientemente grande para almacenar todo su contenido, aunque el buffer creado para almacenar la entrada de Reduce lo sea. Y para el archivo de 1000 MB los buffers no son capaces de almacenar todo el volumen de datos intermedios generados.

Los parámetros de configuración para los buffers utilizados durante la experimentación fueron los valores establecidos por defecto en Hadoop. Sus valores son presentados en la tabla 4.1.

Parámetro	Valor
HADOOP_HEAPSIZE (tamaño máximo de heap)	1000 MB
io.sort.mb (tamaño del buffer de salida de Map)	100 MB
mapred.job.shuffle.input.buffer.percent (tamaño del buffer de entrada de Reduce)	70 % (del valor de HADOOP_HEAPSIZE)
io.sort.spill.percent (límite del buffer de salida de Map para iniciar volcado a disco)	80 % (de io.sort.mb)
mapred.job.shuffle.merge.percent (límite del buffer de entrada de Reduce para iniciar volcado a disco)	66 % (del tamaño del buffer)
mapred.inmem.merge.threshold (límite del número de entradas Maps distintas para el buffer de entrada de Reduce iniciar volcado a disco)	1000

Tabla 4.1 – Parámetros de los buffers utilizados durante la caracterización de MrMAQ

El objetivo del primer experimento fue evaluar el uso del disco por la aplicación MrMAQ. La Figura 4.4 muestra el porcentaje de utilización del disco cuando el volumen del fichero de referencia es de 18 MB. Como el volumen del archivo es pequeño, la utilización del sistema de entrada y salida es baja y está concentrada en el principio de la fase Map, cuando se leen los datos de entrada y al final de las fases Map y Shuffle, cuando se vuelcan a disco el contenido de los buffers intermedios.

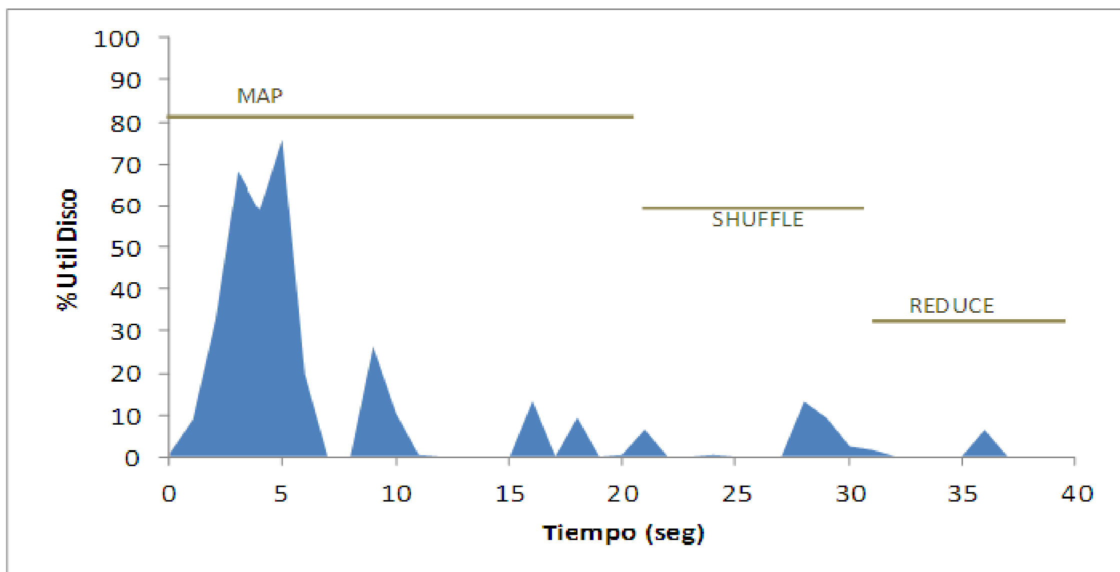

Figura 4.4 - Uso de disco por MrMAQ para una entrada de 18 MB.

La Figura 4.5 muestra la cantidad de datos escritos (en color naranja) y leídos (en color azul) desde el disco durante cada una de las fases. Podemos observar que sólo se produce lectura durante la fase Map, cuando son leídos los datos desde los archivos de consulta y referencia. Esto ocurre porque todos los datos intermedios generados durante el procesamiento de Map y Reduce pueden ser mantenidos en los buffers creados por Hadoop. Las escrituras ocurren al final de la fase Map y al final de fase Shuffle (fase que distribuye la salida de Map para los nodos donde se procesarán la fase Reduce y que hace la mezcla de los archivos recibidos) porque la política de persistencia de Hadoop así lo determina. También al final de la fase Reduce se genera la escritura de los datos de salida.

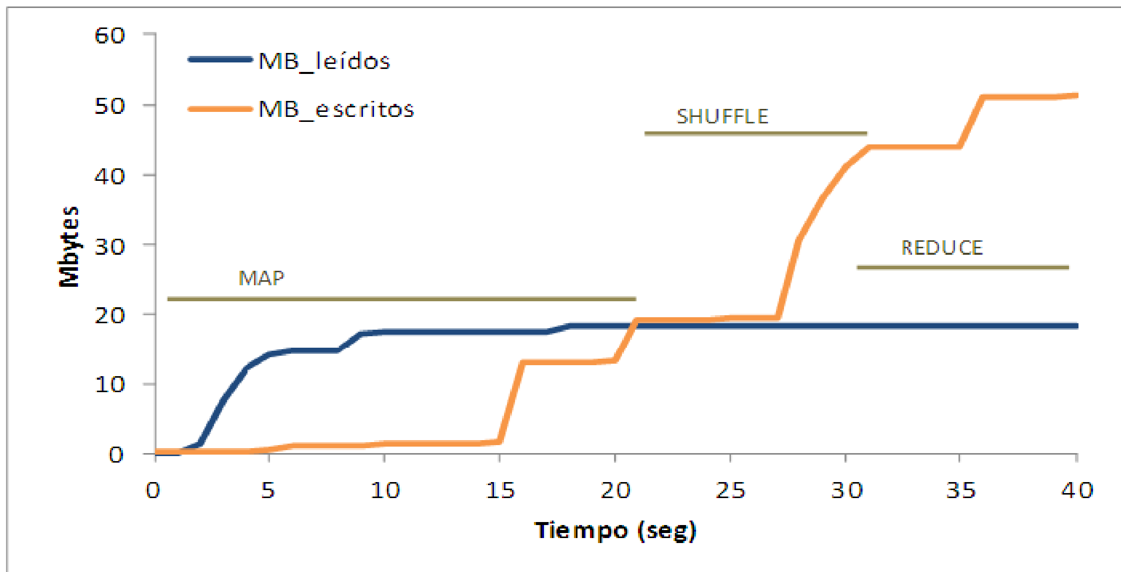

Figura 4.5 - Lecturas/Escrituras realizadas por MrMAQ para una entrada de 18 MB.

La Figura 4.6 muestra el porcentaje de utilización del disco durante el procesamiento de un archivo de entrada de 60 MB. Para este volumen de entrada ocurre un incremento en la utilización del disco durante la fase Map, y especialmente al final de su procesamiento. También se puede notar un uso más intenso del disco al final de la fase Shuffle cuando los datos recibidos de la fase Map son volcados en el disco.

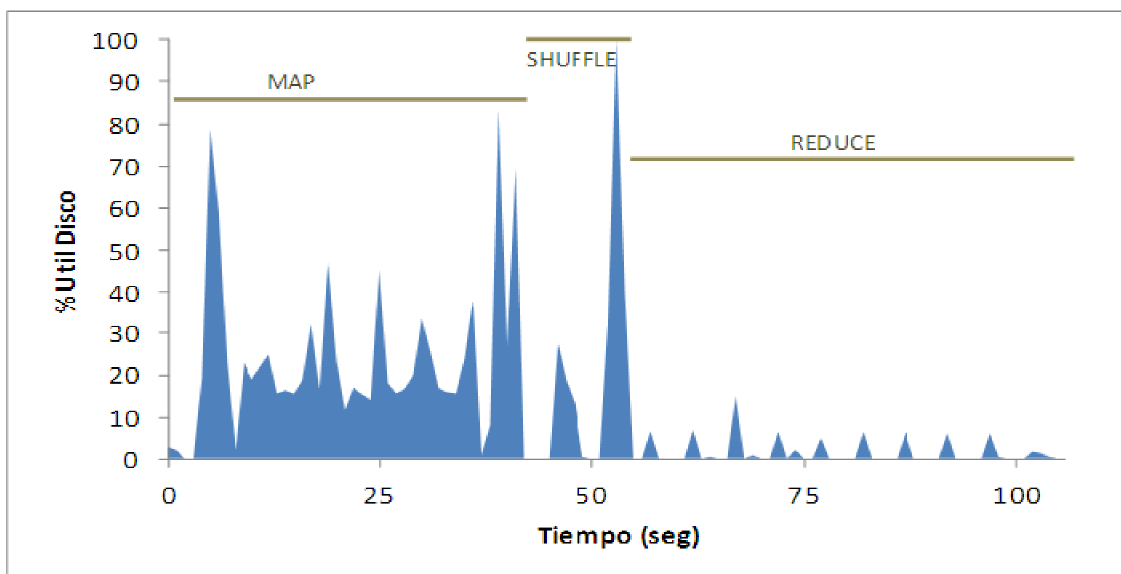

Figura 4.6 - Uso de disco por MrMAQ para una entrada de 60 MB.

Podemos observar en la Figura 4.7 que el volumen de datos escritos durante la fase Map es tres veces superior al volumen de datos de entrada de la aplicación. El *buffer* de salida de la tarea Map no es suficientemente grande para mantener este volumen de datos intermedios generados. Por lo tanto, se hace necesario volcar estos datos en disco, toda vez que el límite de almacenamiento del *buffer* es alcanzado. Sin embargo, como el *buffer* de entrada de la tarea Reduce es capaz de almacenar todo el volumen de datos intermedios, el uso más intenso del disco ocurre apenas al final de la fase Shuffle, cuando los datos recibidos son volcados al disco, como define la política de persistencia de datos de Hadoop..

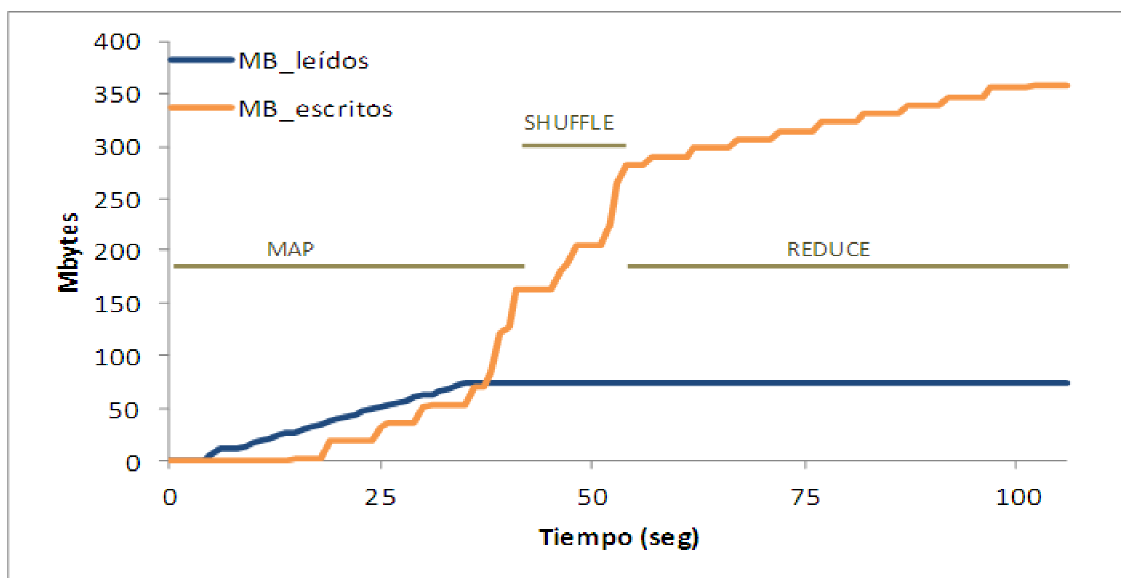

Figura 4.7 - Lecturas/Escrituras realizadas por MrMAQ para una entrada de 60 MB.

Cuando el tamaño del archivo de referencia crece a 1000MB, ambos *buffers* ya no son capaces de almacenar todo el volumen de datos intermedios generado. Podemos observar en la Figura 4.8 que el uso del disco sufre una sobrecarga (uso superior al 100%) en el final de la fase Map cuando hace el *merge* de los archivos volcados en disco, y también durante la fase Shuffle

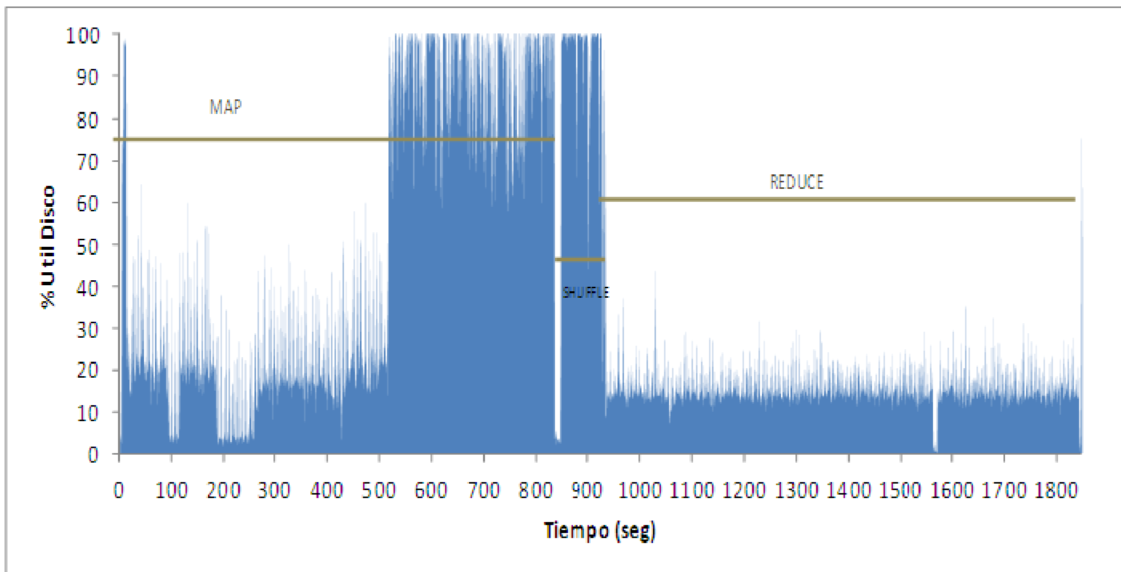

Figura 4.8 - Uso de disco por MrMAQ para una entrada de 1000 MB

En la Figura 4.9 podemos observar intensa actividad de lectura y escritura durante todo el procesamiento de la aplicación MrMAQ. El volumen de datos intermedios generados por la aplicación es cerca de siete veces superior al volumen de datos de entrada. Los *buffers* intermedios creados por Hadoop no soportan esta gran cantidad de datos recibidos y necesitan volcarlos en disco. Cada vez que los límites de los *buffers* son alcanzados, un nuevo archivo es creado con los datos volcados. Y el contenido de estos archivos necesita volver a ser leído desde el disco durante el procesamiento de la aplicación.

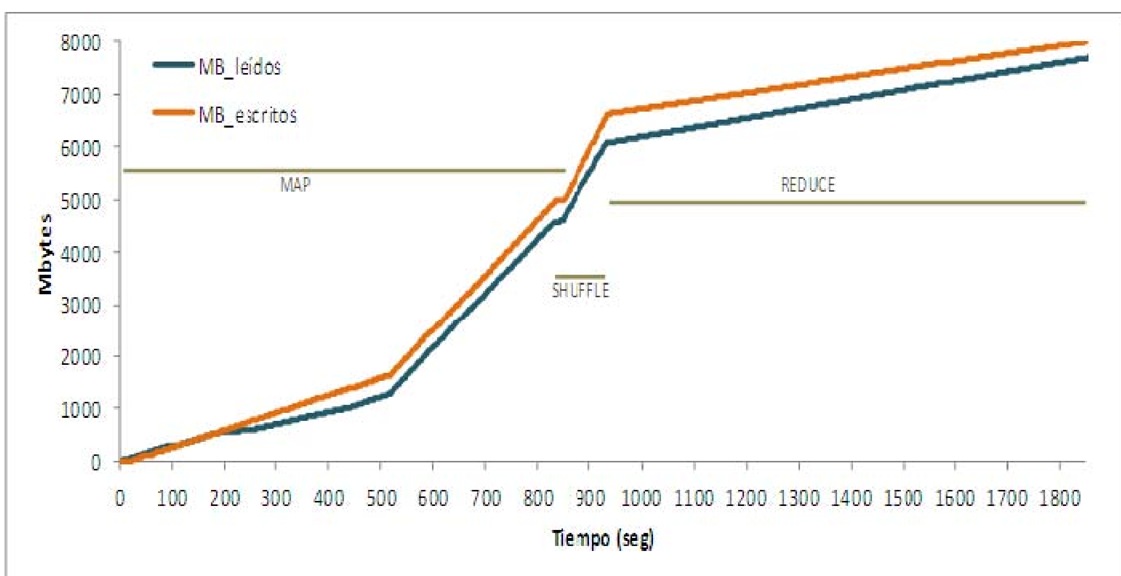

Figura 4.9 - Lecturas/Escrituras realizadas por MrMAQ para una entrada de 1000 MB.

También evaluamos en nuestros experimentos el uso de la CPU durante la ejecución de una instancia de MrMAQ. La Figura 4.10 muestra los resultados cuando el tamaño del archivo de referencia es de 18 MB. La gráfica en azul representa el porcentaje de uso del usuario (*%user*). El color rojo representa el porcentaje de uso del sistema (*%sys*) y el verde indica el porcentaje del tiempo de espera de CPU por los datos de entrada y salida (*%iowait*). Podemos observar que hay un tiempo de espera generado durante la lectura de los datos de entrada de la tarea Map. Sin embargo, como el volumen de datos procesados es relativamente bajo, no hay sobreutilización de la CPU durante la ejecución de la aplicación.

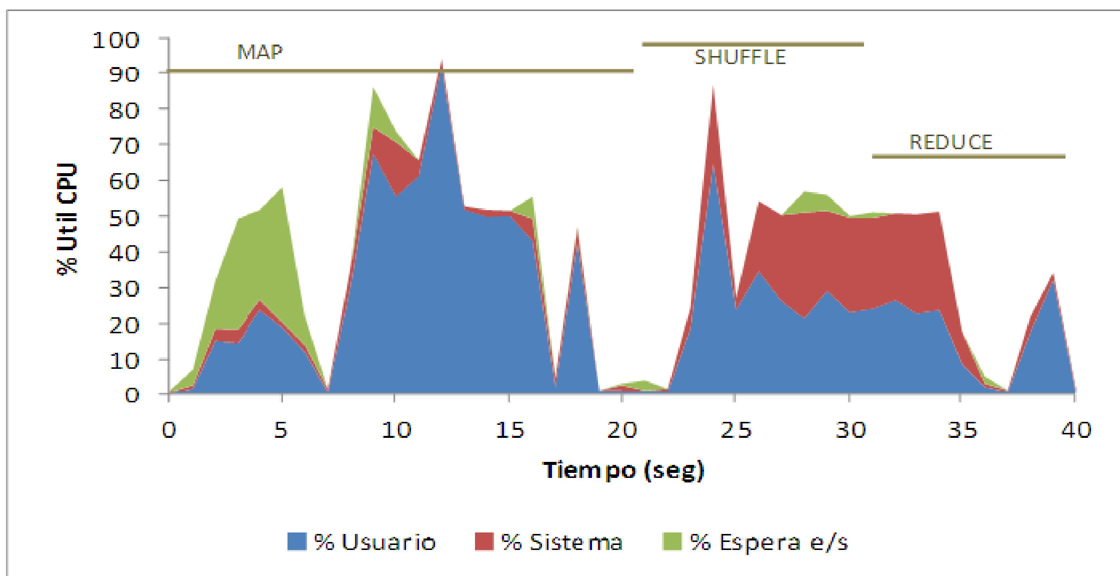

Figura 4.10 - Uso de CPU por MrMAQ para una entrada de 18 MB.

La Figura 4.11 presenta el uso de la CPU cuando el archivo de referencia de entrada es de 60 MB. Podemos observar en los valores de las gráficas que durante la fase Map hay períodos de picos de uso de la CPU. Estos picos se producen en función de las alternancias entre accesos al disco para leer los datos de entrada y el procesamiento de los mismos. La fase Shuffle presenta al principio una etapa de bajo consumo de CPU, cuando envía los datos de la salida de Map para los nodos donde se ejecutarán las tareas Reduce, y un momento donde hay un mayor consumo de CPU, cuando se mezclan y ordenan los datos recibidos de cada tarea Map; y se los prepara para que sean procesados por la tarea Reduce. La fase Reduce presenta un uso más uniforme de la CPU.

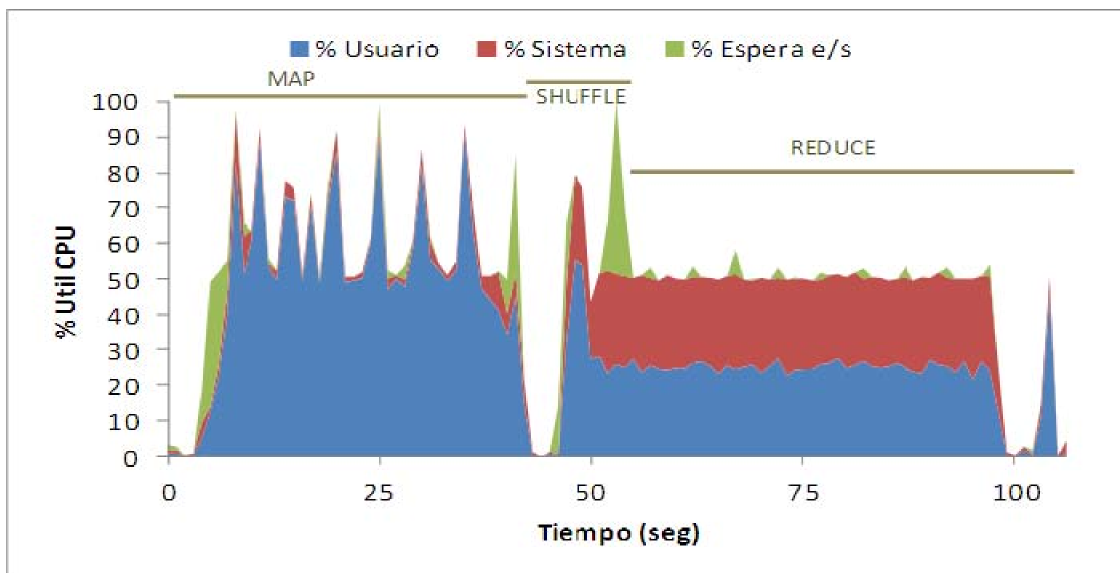

Figura 4.11 - Uso de CPU por MrMAQ para una entrada de 60 MB.

La Figura 4.12 presenta los resultados del experimento cuando el tamaño del archivo de referencia es de 1000 MB. Para este volumen de datos de entrada vimos anteriormente que Hadoop realiza volcados sucesivos de los datos intermedios para el disco. Se puede observar en la Figura 4.12 que la sobrecarga sofreída por el disco en razón de su uso intensivo durante el final de la fase Map y durante la fase Shuffle provocan tiempos elevados de espera por los datos.

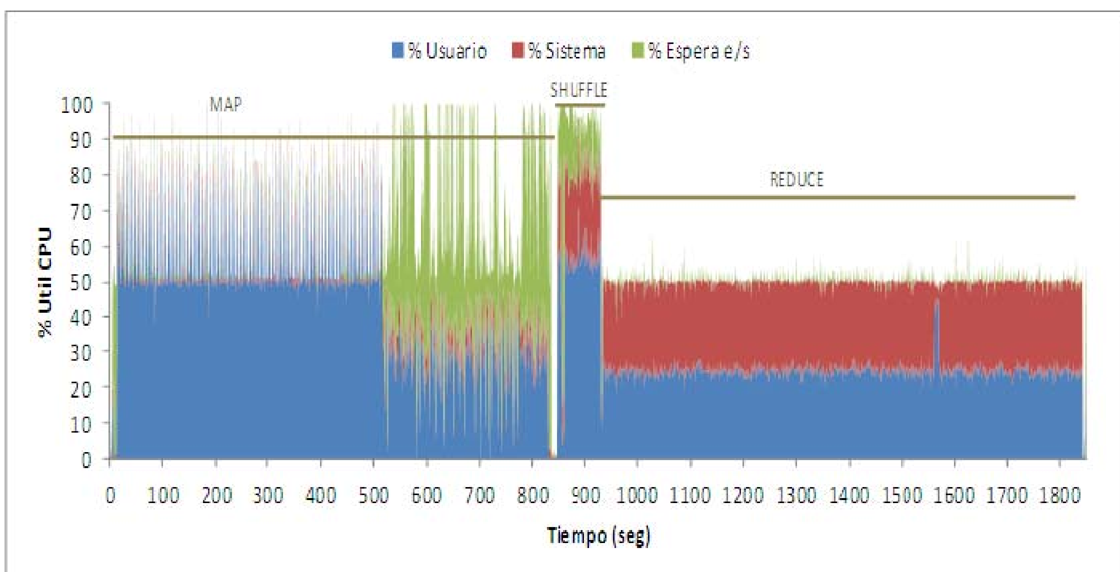

Figura 4.12 - Uso de CPU por MrMAQ para una entrada de 1000 MB.

También podemos observar en las gráficas anteriores que el crecimiento del tamaño de los datos de entrada aumenta la duración de cada fase de la aplicación, sin embargo no causa un incremento en el uso medio de la CPU en cada una de estas fases. En la Figura 4.13 se muestra el uso promedio de la CPU (tiempo de usuario + tiempo de sistema) para una entrada de 1000 MB. En color negro se presenta el uso promedio de todo el periodo de ejecución de la aplicación. Y en color rojo se presenta el uso promedio de la CPU en cada fase. Podemos observar que la CPU tiene un uso promedio de aproximadamente 50% durante la ejecución de la aplicación, lo que nos sugiere que las esperas por datos de entrada y salida llevan a una subutilización de la CPU.

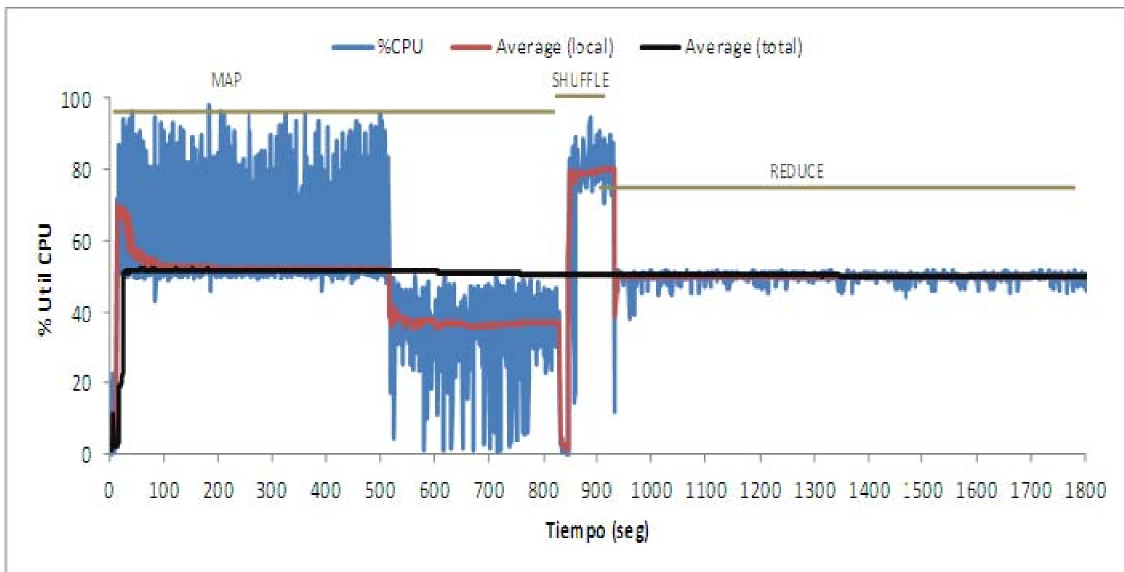

Figura 4.13 - Uso en promedio de CPU por MrMAQ para una entrada de 1000 MB.

También realizamos experimentos con el objetivo de evaluar el nivel de localidad de datos obtenido por la política de planificación de trabajos de Hadoop. La distribución de los bloques de entrada entre los nodos del *cluster* implementada por HDFS es aleatoria. Eso genera un desbalanceo de carga entro los nodos del cluster. Por defecto, la política de planificación de trabajos en Hadoop busca garantizar localidad para las tareas Map. Lo que significa que Hadoop intenta asignar las tareas Map a los mismos nodos donde estén ubicados los bloques de entrada correspondientes a cada tarea. Sin embargo, como hay una distribución desbalanceada de bloques entre los

nodos, Hadoop tiene que copiar algunos de los bloques, en tiempo de ejecución, para garantizar el balanceo de carga. Sin embargo esto genera tráfico innecesario en la red.

Para realizar este experimento ejecutamos la aplicación MrMAQ procesando repetidas veces un archivo de consulta con 5 KB y variando el tamaño del archivo de referencia: 4 a 64 GB.

Los parámetros de configuración utilizados durante la experimentación fueron los valores presentados en la tabla 4.2. Los parámetros de configuración de los buffers intermedios fueron mantenidos en sus valores por defecto.

Parámetro	**Valor**
dfs.block.size (tamaño de cada bloque de entrada)	64 MB
dfs.replication (número de réplicas de cada bloque)	1
mapred.tasktracker.map.tasks.maximun (número máximo de tareas Map asignadas por nodo)	4
mapred.tasktracker.reduce.tasks.maximun (número máximo de tareas Reduce asignadas por nodo)	2

Tabla 4.2 – Parámetros de configuración utilizados durante la evaluación de la localidad de datos para las tareas Map en MrMAQ.

La Figura 4.14 muestra los resultados obtenidos por la experimentación. En color naranja se presenta el número total de bloques creados por HDFS en función del tamaño del archivo de entrada. En color lila se presentan el número de bloques que fueron procesados en el mismo nodo donde estaban almacenados, por lo tanto en estos casos las tareas Map tuvieran garantizadas la localidad de datos. Los demás bloques fueron copiados desde el nodo donde estaban almacenados hacia los nodos donde fueron procesados. Podemos observar en los resultados que la localidad de datos fue obtenida por entre 50% y 80% de las tareas Map.

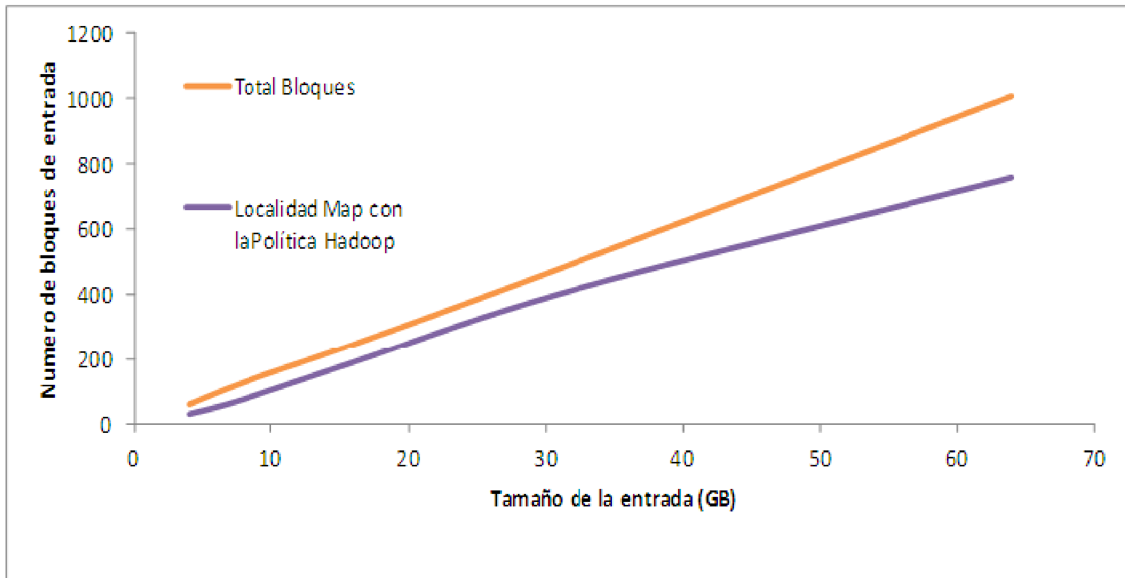

Figura 4.14 - Localidad de datos para la fase Map de la aplicación MrMAQ.

4.2.2 Intel Benchmark

HiBench Suite es un *benchemark* Hadoop desarrollado por Intel que consiste en un conjunto de aplicaciones aplicaciones de código abierto. Entre las aplicaciones disponibles en el paquete, hemos utilizado los micros *benchmarks* Sort y WordCount. También hemos utilizado el *benchmark* de aprendizaje automático: Mahout K-means Clustering.

La implementación MapReduce del algoritmo de ordenación Sort toma como entrada un archivo de texto generado por el programa generador automático de textos de Hadoop, nombrado RandomTextWriter. Sus funciones Map y Reduce son funciones de tipo identidad, esto significa que generan salidas iguales a las entradas. La aplicación Sort, en su versión Hadoop, es una aplicación intensiva de datos tanto en la fase Map como en la fase Reduce.

La aplicación WordCount implementada en el *benchmark* HiBench también utiliza como archivo de entrada un texto generado por RandomTextWriter Hadoop. La función Map genera el par de valores [palabra, 1] para cada palabra que se encuentra en el archivo de entrada. Aún en la fase Map una función del tipo *combiner* hace la suma parcial de los valores de cada palabra coincidente. El objetivo de la función

combiner es disminuir al máximo posible el volumen de datos intermedios generados por la aplicación. Por último, en la fase Reduce se suman los valores de todas las palabras coincidentes generadas por las diferentes instancias de Map.

La aplicación Mahout K-means Clustering implementa el algoritmo K-means utilizado para extraer conocimiento y levar a cabo la minería de datos. La aplicación toma como entrada un conjunto de muestras representadas por los vectores d-dimensional que se generan en base a la distribución Guassian y con distribución uniforme. La aplicación calcula el centroide de cada grupo y luego asigna a cada muestra un clúster. En su versión implementada para el *benchmark* HiBench también está clasificada como una aplicación intensiva de datos y que hace uso intensivo de la CPU.

4.3 Shared Input Policy

La implementación de *Shared Input Policy* ocurrió en dos etapas. En la primera de ellas, establecemos como objetivo aumentar la localidad de datos para las tareas Map, que se observa disminuida por cuenta del desbalanceo de carga, provocado por la política de distribución de bloques de entrada implementada por HDFS. En la segunda etapa definimos como objetivo mejorar la gestión de datos utilizando una RAMDISK.

Las próximas secciones presentan la experimentación en la misma orden en que la política fue implementada. Primero los resultados parciales obtenidos por el aumenta de la localidad de las tareas Map. A continuación, los resultados obtenidos por la utilización de la RAMDISK.

4.3.1 Localidad de las tareas Map

Diseñamos experimentos para evaluar el aumento de la localidad de datos para las tareas Map y su efecto en el tiempo necesario para procesar lotes de trabajo. La métrica elegida para evaluar los efectos de la política *Shared Input Policy* fue el tiempo de *makespan*. Consideramos tiempo de *makespan* como el tiempo medido desde el

envío del primero trabajo al *cluster* hasta la finalización del último trabajo lanzado. Así tenemos como tiempo medido de *makespan* el tiempo medido para ejecutar todos los trabajos en un lote de procesamiento.

Implementamos dos conjuntos de experimentos. En el primero de ellos planificamos lotes de trabajos conteniendo aplicaciones del tipo MrMAQ. Un solo archivo de referencia era compartido entre todos los trabajos del lote, sin embargo cada trabajo tenía su proprio archivo de consulta. Variamos el tamaño del archivo de referencia en cada experimento para evaluar el efecto de la política en distintos volúmenes de datos. Utilizamos entradas de 1 GB, 2GB y 4GB. Los archivos de consulta variaran entre 2 KB y 400 KB dependiendo del trabajo en ejecución.

Variamos el tamaño de cada lote desde 1 trabajo hasta 64 trabajos y comparamos los resultados obtenidos entre *Shared Input Policy* y dos políticas de planificación de trabajos distintas: FIFO, la política por defecto de HADOOP y *Fair Scheduler* que es una política de planificación de trabajos que procura asegurar, en promedio, igual disponibilidad de recursos entre los trabajos a largo plazo.

Para las políticas *Shared Input Policy* y FIFO configuramos una única cola de trabajos y para la política *Fair Scheduler* configuramos un pool para cada usuario. Los recursos del *cluster* fueron compartidos igualmente entre cada pool a lo largo del tiempo. Los trabajos lanzados en cada pool también fueron planificados usando *Fair Scheduler*.

Las Figuras 4.15, Figura 4.16 y Figura 4.17 presentan los tiempos de *makespan* obtenidos mediante la experimentación. Los resultados muestran que *Shared Input Policy* mejora el tiempo de *makespan* promedio en 7,8% cuando comparado con la política de planificación de trabajos FIFO y uno promedio de 8,3 % en comparación con *Fair Scheduler*.

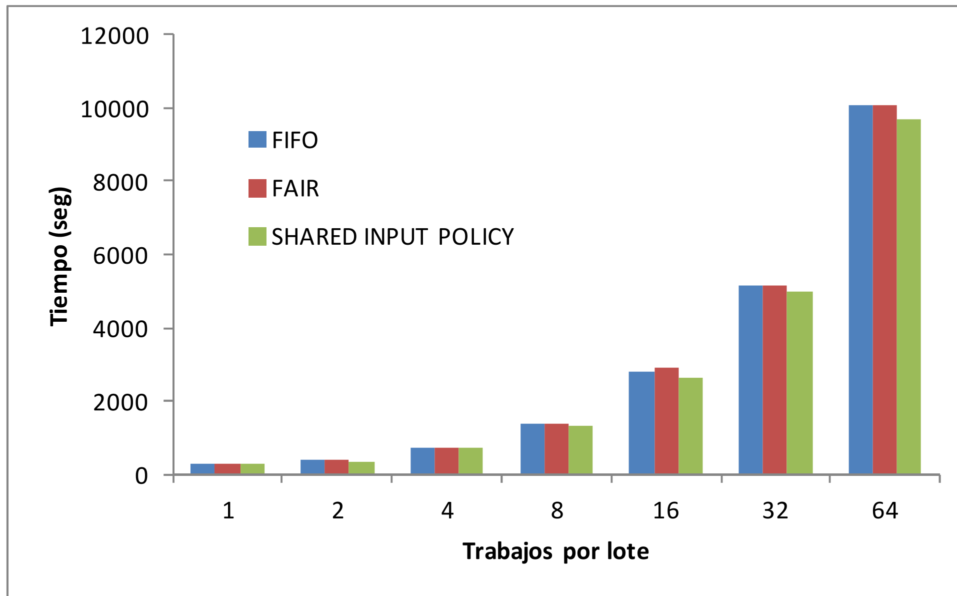

Figura 4.15 - Tiempo de Makespan por lote de aplicaciones MrMAQ con entrada compartida de 1 GB.

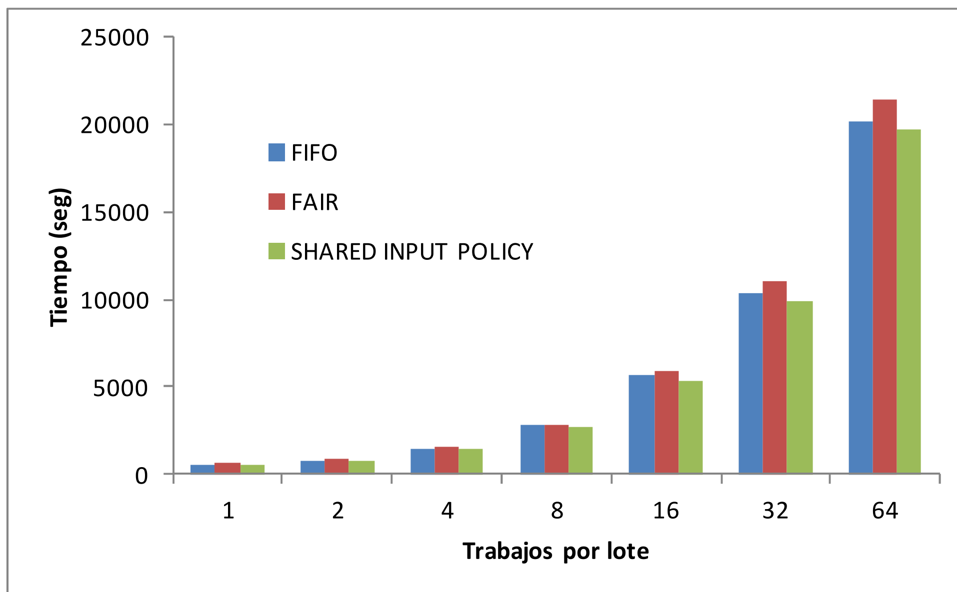

Figura 4.16 - Tiempo de Makespan por lote de aplicaciones MrMAQ con entrada compartida de 2 GB.

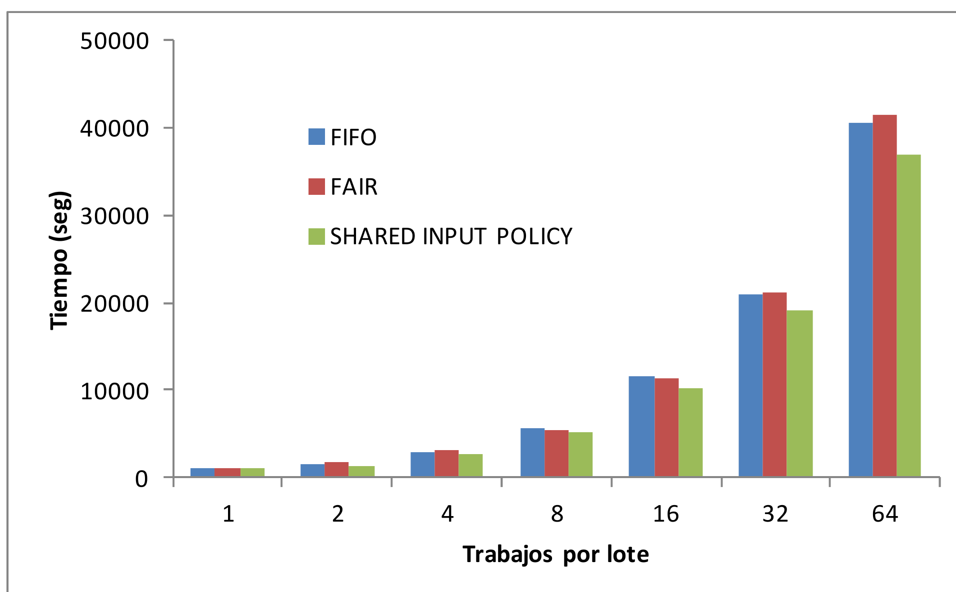

Figura 4.17 - Tiempo de Makespan por lote de aplicaciones MrMAQ con entrada compartida de 4 GB.

En el segundo conjunto de experimentos planificamos la ejecución de cuatro lotes de trabajos cada uno de ellos compartiendo un archivo de referencia distinto. Este diseño de experimentación nos permitió comparar *Shared Input Policy* con la política de planificación de trabajos *Capacity* procura garantizar una cantidad mínima de recursos entre usuarios o aplicaciones de un *cluster*.

Para utilizar *Capacity Scheduler* dividimos los recursos del *cluster* en cuatro grupos. Cada grupo tenía la misma cantidad de recursos y su propia cola para lanzamiento de trabajos. Cada cola recibió los trabajos de un lote distinto. Variamos el número de trabajos en cada lote desde 1 hasta 16 trabajos (haciendo un total de hasta 64 trabajos para planificación).

En las ejecuciones donde planificamos los trabajos con FIFO y *Shared Input Policy* utilizamos una única cola para el lanzamiento de los cuatro lotes. Los lotes fueron procesados en secuencia y no de manera concurrente. La métrica de prestaciones utilizada también fue el tiempo de *makespan*. Medimos el tiempo desde el lanzamiento del primero trabajo del primero lote hasta la finalización del último trabajo del último lote de procesamiento.

Las Figuras 4.18, Figura 4.19 y Figura 4.20 presentan los tiempos de *makespan* obtenidos mediante la experimentación. Los resultados muestran que *Shared Input Policy* mejora el tiempo de *makespan* promedio en 7,8% cuando es comparado con la

política de planificación de trabajos FIFO y en promedio de 3,2 % en comparación con *Capacity Scheduler*.

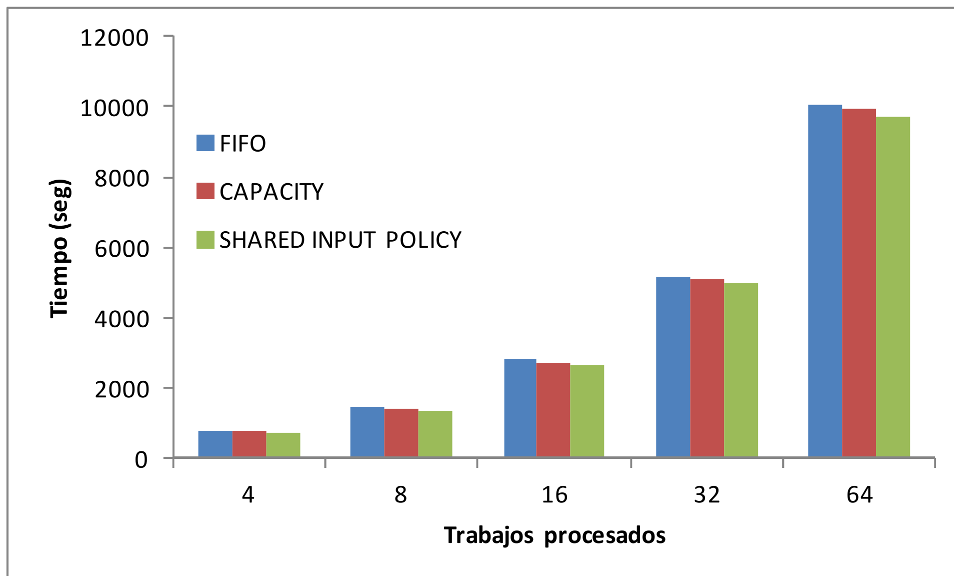

Figura 4.18 - Tiempo de Makespan para cuatro lotes de aplicaciones MrMAQ con entrada compartida de 1 GB.

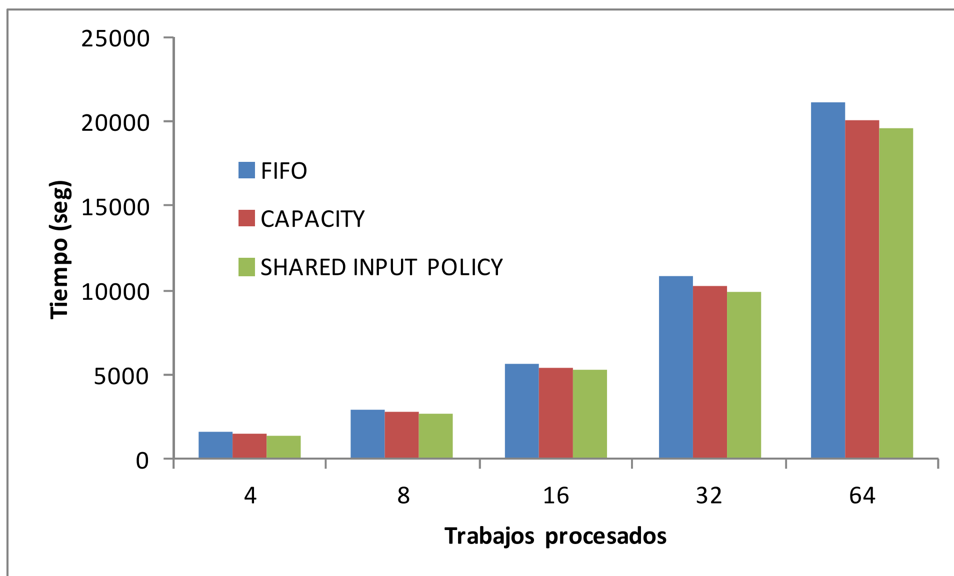

Figura 4.19 - Tiempo de Makespan para cuatro lotes de aplicaciones MrMAQ con entrada compartida de 2 GB.

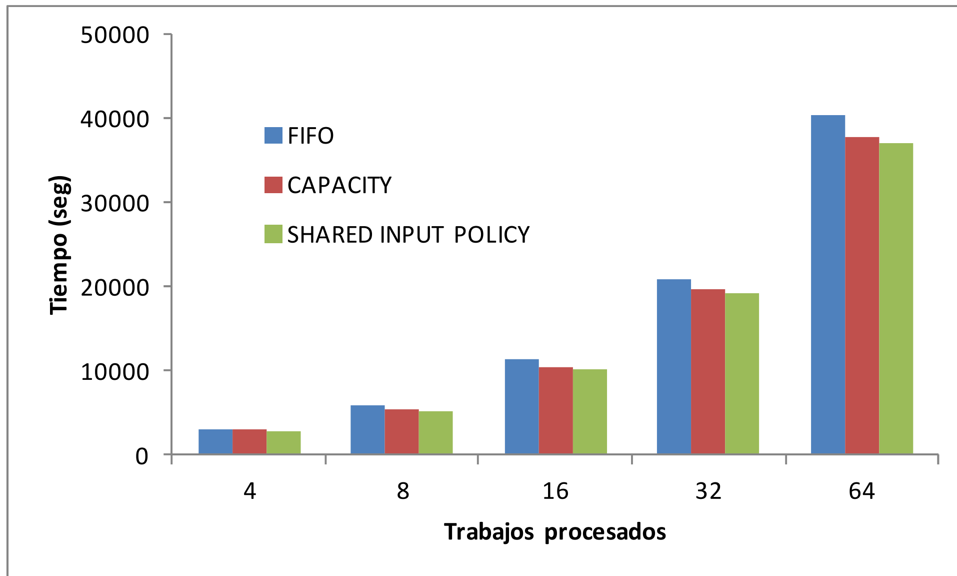

Figura 4.20 - Tiempo de Makespan para cuatro lotes de aplicaciones MrMAQ con entrada compartida de 4 GB.

También medimos el incremento obtenido en el aumento de la localidad para las tareas Map. La Figura 4.21 compara la localidad obtenida por *Shared Input Policy* y por Hadoop con su política de planificación por defecto. En naranja se presenta el total de bloques de entrada para conjuntos de dados de 4 a 64 GB. El tamaño de cada bloque es de 64 MB y el número de copias de cada bloque es igual a uno. En lila se presenta el número de bloques que han sido procesados en el mismo nodo donde han sido almacenadas por HDFS cuando se utiliza FIFO. Y en azul, la localidad de las tareas Map obtenida cuando se utiliza *Shared Input Policy* para planificar los trabajos.

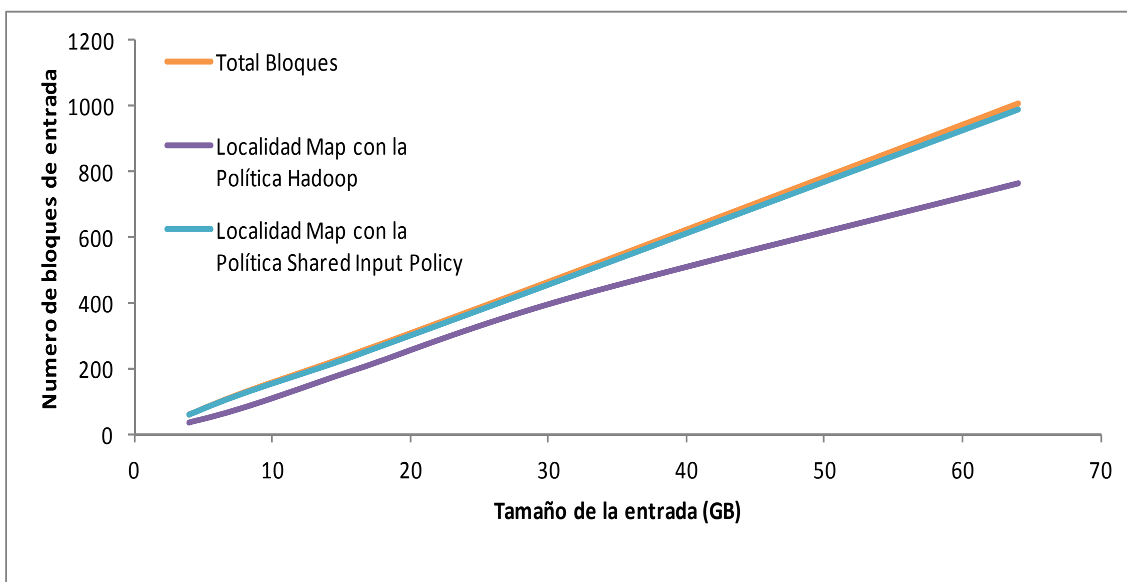

Figura 4.21 - Localidad de datos para la fase Map con Sahred Input Policy.

Shared Input Policy es capaz de incrementar la localidad das tareas Map a valores cercanos a los 100% mejorando el tiempo de ejecución de las aplicaciones como se podrá observar en el próximo apartado.

4.3.2 Monitorización de los Recursos

Evaluamos cómo la RAMDISK influye en la utilización de los recursos del *cluster* y las consecuencias de su uso para las prestaciones obtenidas por nuestra política de planificación de trabajos. A tal efecto monitorizamos los recursos del *cluster* en dos escenarios distintos. En el primero de ellos utilizamos los principios estándar de Hadoop donde los datos intermedios generados por la aplicación fueron volcados en el disco duro. Y en el segundo escenario, montamos la RAMDISK para el almacenamiento temporal de los datos intermedios. Hicimos la monitorización de uso del disco con la herramienta *iostat* y el uso de la CPU con la herramienta *vmstat*, ambas herramientas disponibles en Linux.

La Figura 4.22 muestra el porcentaje de uso del disco cuando en el *cluster* Hadoop ejecutamos la aplicación Sort del paquete HiBench Benchmark Suite teniendo un archivo de entrada de 8 GB. La línea azul representa los valores obtenidos cuando utilizamos Hadoop en su configuración original, eso quiere decir que los datos intermedios generados en la fase Map se vuelcan directamente en el disco duro cuando los buffers de memoria generados para recibir la salida de las tareas Map están llenos. También en los nodos donde se reciben los datos intermedios para la fase Reduce, los datos se vuelcan en el disco duro cuando los buffers están llenos. Podemos observar que el disco se utiliza intensamente a lo largo de la ejecución de la aplicación debido a la enorme cantidad de datos que se vuelcan. Incluso algunas veces el disco sufre sobrecarga (el uso mayor de 100 %), debido a su uso intensivo. Podemos observar en la Figura 4.23 que el uso intensivo del disco genera tiempos de espera de la CPU (en color verde) para los datos de entrada y salida.

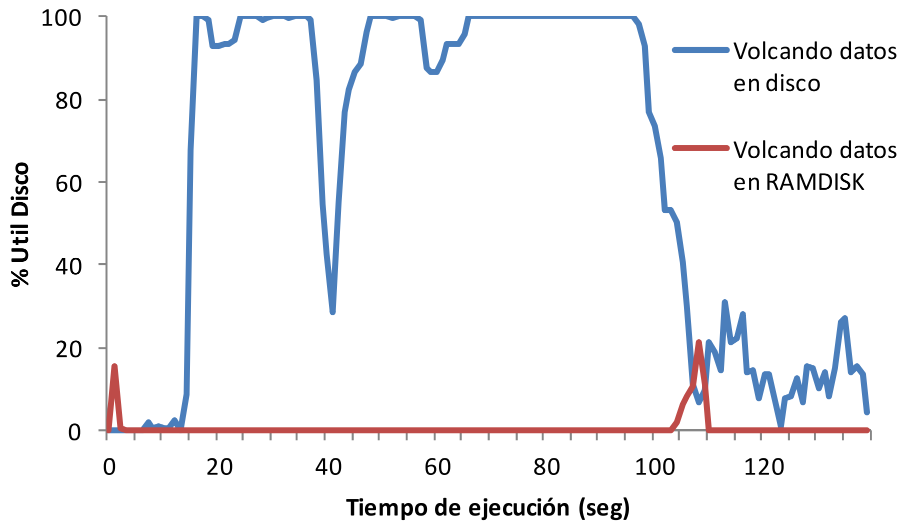

Figura 4.22 - Uso de disco volcando datos en disco y en RAMDISK (aplicación Sort con entrada de 8 GB).

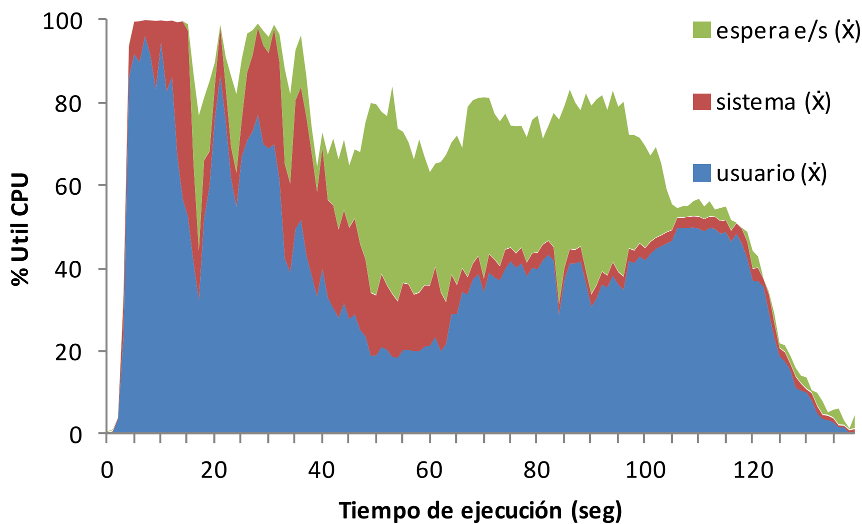

Figura 4.23 - Uso de CPU volcando datos en disco (aplicación Sort con entrada de 8 GB).

Posteriormente introducimos la RAMDISK como una capa intermedia entre los buffers temporales creados por Hadoop y el disco duro. Cuando los buffers excedían sus límites de almacenamiento Hadoop volcaba su contenido directamente en la RAMDISK. Establecimos el límite *soft* de 70 % del tamaño del buffer a partir del cual las nuevas tareas asignadas al nodo volcaban el contenido de sus buffers directamente en el disco duro. También definimos como valor límite *hard* 90 % del tamaño del buffer a partir del cual las tareas que volcaban los datos en la RAMDISK pasaban a volcar los

datos intermedios directamente en el disco, tal como el establecido por la configuración por defecto de Hadoop. La línea roja de la Figura 4.22 refleja la monitorización de uso del disco. Podemos observar que hubo una fuerte disminución en el uso del dispositivo. Como resultado, se puede ver en la Figura 4.24 que el tiempo de espera de la CPU por los datos de entrada/salida desaparecen, y el tiempo de ejecución de la aplicación sufre una reducción significativa.

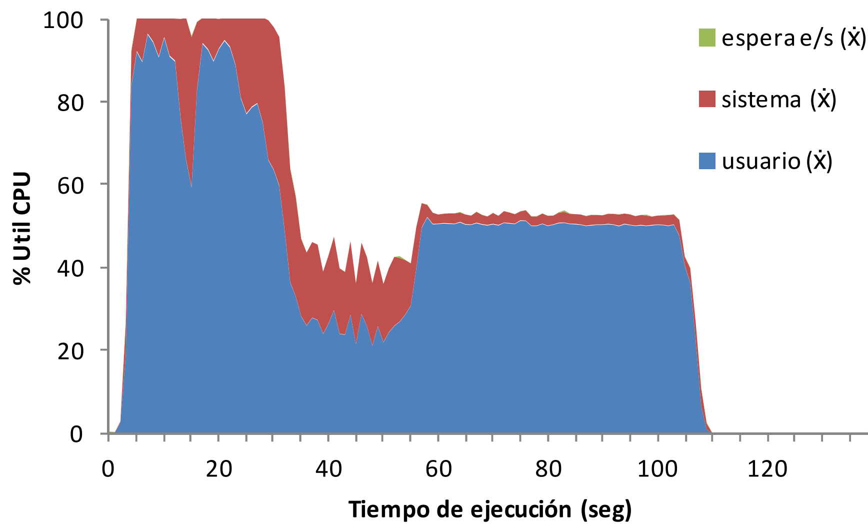

Figura 4.24 - Uso de CPU volcando datos en RAMDISK
(aplicación Sort con entrada de 8 GB).

4.3.3 Disco Local y RAMDISK

Comparamos el rendimiento entre Hadoop configurado para volcar los datos intermedios en el disco local y Hadoop configurado para volcar los datos intermedios en la RAMDISK. En esta situación, ejecutamos la implementación Sort del paquete HiBench Benchmark Suite en tres escenarios distintos.

En el escenario A, las variables que definen los buffers creados y utilizados por Hadoop para almacenamiento de los datos intermedios fueron mantenidos con los tamaños de la configuración por defecto; además del contenido de los buffers ser volcado en disco duro cuando se llenaban.

En el escenario B, los datos en los buffers siguieron siendo volcados hacia el disco. Sin embargo los buffers fueron optimizados por la aplicación de sintonización

Starfish para mantener la máxima cantidad posible de dados en memoria y así disminuir el volumen de dados volcados hacia el disco.

Cuando hay suficiente espacio de almacenamiento para guardar los datos intermedios en la memoria, podemos ajustar los parámetros de configuración de Hadoop para reducir el número de datos volcados en el disco. Para la fase Reduce es posible mantener todos los datos en la memoria sin volcarlos en el disco (siempre que haya espacio suficiente en memoria). Sin embargo, la implementación Hadoop utilizada, requiere que la salida generada por la fase Map sea volcada en el disco duro al menos una vez antes de iniciar la fase Shuffe, aunque haya espacio en memoria para mantener los datos.

Así, en el escenario B ajustamos los valores de las variables que definen los buffers utilizados por Hadoop para el almacenamiento de datos intermedios para volcar los datos hacia el disco una única vez, al final de la fase MAP. A tal efecto, aumentamos el tamaño de la memoria disponible de la máquina virtual de Java en cada tarea; lo que permitió aumentar el tamaño del búfer para la salida de la fase Map (*io.sort.mb*) y permitió también aumentar el tamaño del buffer para los datos de entrada de la fase Reduce (*mapred.job.shuffle.input.buffer.percent*). La variable *mapred.job.reduce.input.buffer.percent* define el porcentaje del buffer de entrada de la tarea Reduce que se puede utilizar para mantener los datos intermedios en la memoria después de terminar la fase Shuffle. Por defecto, esta variable se establece con el 0,0%, lo que requiere que Hadoop vuelca en el disco duro los datos recibidos hacia la fase Reduce y que están todavía en el buffer antes de iniciar efectivamente la fase Reduce. Cambiamos el valor de esta variable para permitir que los datos de entrada de Reduce se pudieran mantener en los buffers sin volcarlos hacia el disco duro.

En el escenario C incluimos la RAMDISK para el almacenamiento temporal de los datos intermediarios generados por la aplicación. Los buffers generados por Hadoop fueron definidos utilizando los mismos valores del escenario A. Así que los datos intermedios eran almacenados en los buffers generados por Hadoop y estos cuando estaban llenos volcaban su contenido en la RAMDISK; tal como estaba definido por nuestra política de planificación de trabajos.

La Tabla 4.3 muestra los valores de los parámetros que fueron cambiados y los tiempos de ejecución obtenidos por la aplicación Sort teniendo un archivo de entrada de 8 GB con 128 tareas Map y 32 tareas Reduce.

Parámetro	Escenario A Disco Local Defecto	Escenario B Disco Local Todo en memoria	Escenario C RAMDISK
HADOOP_HEAPSIZE (MB)	1000	2046	1000
Memoria de la JVM (MB)	200	600	200
io.sort.mb (MB)	100	300	100
mapred.job.shuffle.input.buffer.percent(%)	0,70	0,80	0,70
mapred.job.reduce.input.buffer.percent (%)	0,0	0,50	0,0
Número de volcados	960	128	960
Registros volcados	153.096.866	48.838.169	153.096.866
Tiempo de ejecución (seg)	146	116	109

Tabla 4.3 - Parámetros de configuración de Hadoop y tiempos de ejecución para la aplicación Sort.

Podemos observar que volcar datos en la RAMDISK (escenario C) mejora el rendimiento de la aplicación en 1.34 veces cuando comparado con los parámetros por defecto de Hadoop. Incluso cuando podemos mantener los datos intermedios en los buffers sin hacer volcados (1.06 veces cuando comparado con el escenario B) porque Hadoop realiza como mínimo un volcado de datos en disco al final de cada tarea Map (aunque haya espacio suficiente para mantener los datos en la memoria)

El uso de la RAMDISK también tiene la ventaja de no requerir ajustes de los parámetros de configuración de Hadoop. La gestión del entorno se ve facilitada por el mantenimiento de la configuración estándar de Haoop.

4.3.4 Shared Input Policy con RAMDISK

Diseñamos experimentos para evaluar el efecto sobre el tiempo de *makespan* causado por la integración de *Shared Input Policy* con el uso de una RAMDISK, y almacenar tanto los archivos de entrada como los archivos intermedios. Consideramos tiempo de *makespan* como el tiempo medido desde el envío del primero trabajo al *cluster* hasta la finalización del último trabajo lanzado. Los trabajos fueron enviados al *cluster* en lotes de tamaño fijo y todos los trabajos en el lote compartieron el mismo conjunto de datos de entrada

Creamos dos tipos de experimentos. En el primer tipo, planificamos los trabajos enviados al *cluster* utilizando nuestra política de planificación de trabajos, sin embargo *Shared Input Policy* utilizó el disco duro para volcar los datos intermedios generados por las aplicaciones. En el segundo tipo de experimentación, expandimos *Shared Input Policy* agregando la RAMDISK para recibir de Hadoop los datos intermedios generados por la aplicación. En este segundo experimento también utilizamos la RAMDISK para almacenar el conjunto de datos de entrada compartido entre los trabajos del lote, además de los archivos de salida generados por cada trabajo.

Cada lote lanzado al *cluster* tenía desde un único trabajo hasta 10 trabajos compartiendo el mismo conjunto de datos de entrada. En ambos experimentos los datos de entrada fueron divididos en bloques de 64 MB. Hadoop generó automáticamente el número de tareas Map para coincidir con el número de bloques del archivo de entradas, a tal efecto que cada tarea Map procesa uno de los bloques de entrada. Sin embargo, el número de tareas Reduce se estableció en 32 para todos los trabajos, en función del valor optimo sugerido por Starfish. Para permitir evaluarnos cómo el desbalanceo de carga provocado por la distribución de los bloques afecta la localidad de datos no usamos replicación de bloques. Los resultados se compararon con el tiempo de *makespan* obtenido por la política FIFO.

La Figura 4.25 presenta los tiempos de *makespan* obtenidos en los experimentos cuando ejecutamos la aplicación de Bioinformática MrMAQ implementada por nuestro grupo de investigación. MrMAQ es una aplicación de uso intensivo de datos y maneja una gran cantidad de datos intermedios. Todos los trabajos presentados en el mismo lote compartían un único archivo de referencia de 8 GB. Cada trabajo tenía su propio archivo de consulta con tamaños que variaban desde los 5 KB hasta los 400 KB. Los resultados muestran que la política de planificación de trabajos propuesta – *Shared Input Policy* – mejora el tiempo de *makespan* en promedio de 11,3%, en comparación

con FIFO. Y al añadir RAMDIK a la política propuesta, mejoró el tiempo de *makespan* promedio en el 48,4%.

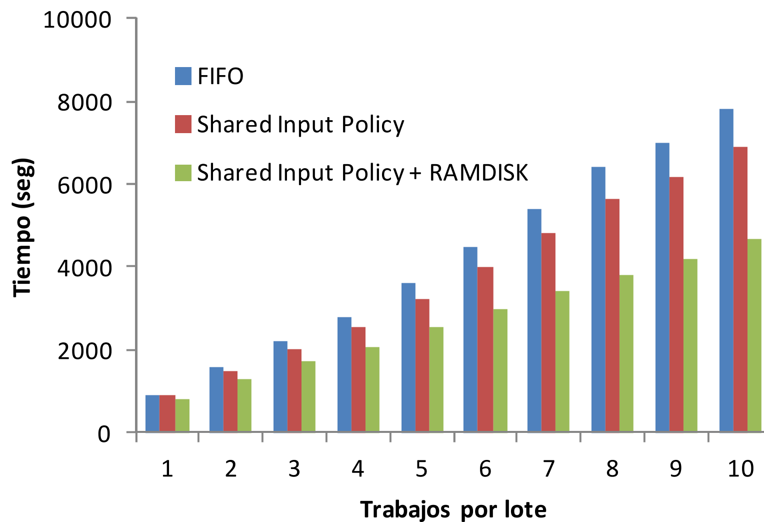

Figura 4.25 - Tiempo de Makespan para la aplicación MrMAQ con entrada compartida de 8 GB.

La Figura 4.26 presenta los tiempos de *makespan* obtenidos en los experimentos con la aplicación Sort perteneciente al *benchmark* HiBench ofrecido por Intel. Para la aplicación Sort se utilizó un conjunto de datos de entrada de 8 GB compartido entre todos los trabajos de los lotes. La implementación Hadoop de Sort es intensiva en datos y maneja una gran cantidad de datos de entrada y de datos intermedios. El tiempo de *makespan* mejoró un 15,9% para la aplicación Sort cuando usamos la política de planificación de trabajos *Shared Input Policy*. Y la versión expandida de *Shared Input Policy* incorporando la RAMDISK el tiempo de *makespan* promedio presentó mejoras en torno del 60,0%.

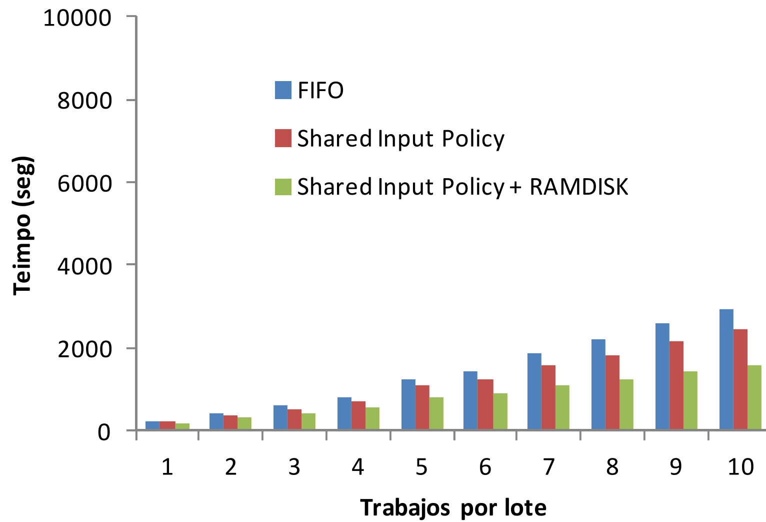

Figura 4.26 - Tiempo de Makespan para la aplicación Sort con entrada compartida de 8 GB.

La Figura 4.27 presenta los tiempos de *makespan* obtenidos en los experimentos con la aplicación Mahout K-means *clustering* también perteneciente al *benchmark* HiBench ofrecido por Intel. Para la aplicación K-means se utilizó un conjunto de datos de entrada de 16 GB. El conjunto de datos de entrada se compartió entre todos los trabajos de los lotes. La implementación Hadoop de Mahout K-means *clustering* es una aplicación que hace uso intensivo de CPU además de manejar una gran cantidad de datos de entrada. El tiempo de *makespan* mejoró un 14,5 % para la aplicación K-means cuando usamos la política de planificación de trabajos *Shared Input Policy*. Y en la versión expandida de *Shared Input Policy* incorporando la RAMDISK el tiempo de *makespan* promedio presento mejoras en torno de 46,7 %.

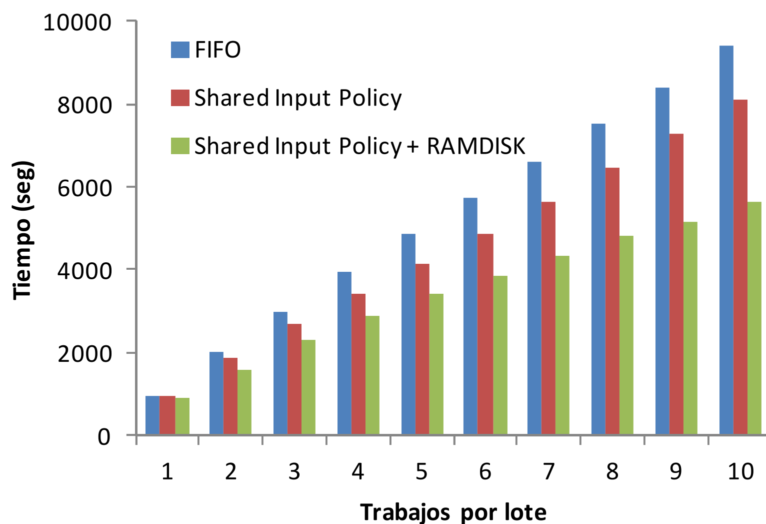

Figura 4.27 - Tiempo de Makespan para la aplicación K-means con entrada compartida de 16 GB.

La Figura 4.28 muestra los tiempos de *makespan* obtenidos en los experimentos con la aplicación WordCount también perteneciente al *benchmark* HiBench ofrecido por Intel. Todos los trabajos del lote compartieron el mismo conjunto de datos de entrada con un tamaño de 30 GB. Las mejoras que se presentan en tiempo de *makespan* promedio fueron de 2,9 % cuando se utilizó *Shared Input Policy* en su versión volcando los datos intermedios en el disco duro y de 2,7 %, cuando nuestra política de planificación de trabajos fue utilizada en su versión que volcaba los datos intermedios en la RAMDISK. La aplicación WordCount se destacó de las demás aplicaciones probadas en nuestro conjunto de experimentación por su implementación Hadoop ser una aplicación intensiva en el uso de la CPU; además de manejar una gran cantidad de datos solamente de entrada. Sin embargo, WordCount tiene una optimización implementada para la salida generada por las tareas Map: una función *combiner*, que es una función que se aplica a la salida de las tareas Map y hace la combinación de los registros que tengan la misma clave. El objetivo de la función *Combiner* es reducir la cantidad de datos volcados en el disco y por lo tanto transferidos para la fase Reduce. La aplicación WordCount genera solamente 12 MB de datos intermedios para una entrada de 30 GB. El bajo volumen de datos intermedios manejados por la aplicación WordCount no mejora significativamente las prestaciones mediante el uso de RAMDISK.

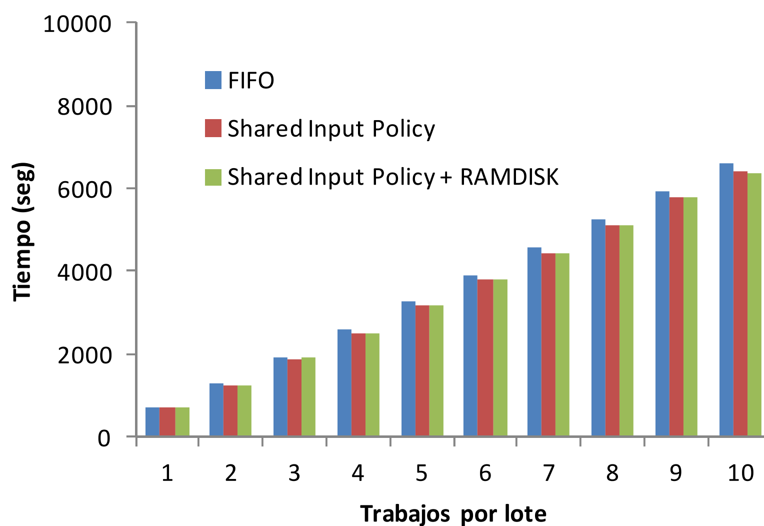

Figura 4.28 - Tiempo de Makespan para la aplicación WordCount con entrada compartida de 30 GB.

Capítulo 5 - CONCLUSIONES

En esta Tesis implementamos una nueva política de planificación de trabajos para *clusters* Hadoop compartidos denominada *Shared Input Policy*, que integra en una única propuesta de solución el problema de planificación de trabajos que comparten un mismo conjunto de entradas, y el problema de gestión de grandes volúmenes de datos intermedios entre las fases Map y Reduce.

Para diseñar la propuesta, evaluamos las prestaciones de un *cluster* Hadoop compartido. El desbalanceo de carga provocado por la política aleatoria de distribución de bloques de entradas entre los nodos del *cluster*, dificulta al planificador de Hadoop garantizar la localidad de datos durante la asignación de tareas Map hacia los nodos. Este hecho conduce a un tráfico innecesario de datos en la red del *cluster* y accesos repetidos a disco, porque los bloques que no tienen garantizada la localidad de datos son copiados desde el nodo donde han sido ubicados hasta el nodo donde serán procesados. En el procesamiento de trabajos con entradas compartidas, este problema se agrava porque el mismo bloque puede ser copiado varias veces.

Por otra parte, los trabajos que generan grandes volúmenes de datos intermedios llevan a una sobrecarga del sistema de archivos local de cada nodo. En el diseño de Hadoop los datos intermedios no son almacenados en el sistema de archivos distribuidos de Hadoop, sino que en el propio sistema de archivos local. Tanto en los nodos en que se les genera (nodos Map), como en los nodos en que se les procesa (nodos Reduce). En los *clusters* compartidos el sistema de entrada/salida se torna un cuello de botella para los trabajos intensivos en datos intermedios.

Propusimos una nueva política de planificación de trabajos que pretende abordar los dos problemas detectados en nuestros análisis, y que puede ser incorporada sin cambios en el diseño original de Hadoop.

En la política propuesta, la gestión de la planificación de trabajos que comparten el mismo conjunto de datos de entrada ocurre en dos niveles: en un nivel micro (nivel de planificación de tareas), que agrupa las tareas de los diferentes trabajos procesados en un mismo lote, y que comparten los mismos bloques de datos, para que se ejecuten en el mismo nodo donde se asignó el bloque. La mejoras en el tiempo de *makespan* para procesar los lotes de trabajo fueron publicados en la Jornada de Paralelismo de España:

- JP2011 – Jornada de Paralelismo 2011
  Autores: Aprigio Bezerra, Tharso de Souza, Porfídio Hernández, Antonio Espinosa, Juan Carlos Moure
  Título: Planificación de trabajos MapReduce en clusters Hadoop no-dedicados.
  Local: La Laguna (Tenerife), España.
  Año: 2011.

Y en el congreso internacional:

- Conference CMMSE-2012 – 12th International Conference on Computational and Mathematical Methods in Science and Engineering.
  Autores: Aprigio Bezerra, Porfídio Hernández, Antonio Espinosa, Juan Carlos Moure
  Título: Job Scheduling in Hadoop Non-dedicated Shared Clusters.
  Publicación: Proceedings. ISBN 978-84-615-5392-1. Páginas 184-195, Volumen: I.
  Local: Murcia, Spain.
  Año: 2012.

Y en un nivel macro (nivel de planificación de trabajos), que agrupa a los trabajos que comparten los conjuntos de datos de entrada para procesarlos en lote. Los resultados fueron publicados en el congreso internacional:

- Conference PBio 2013 (EUROMPI 2013) - International Workshop on Parallelism in Bioinformatics 2013.
  Autores: Aprigio Bezerra, Porfídio Hernández, Antonio Espinosa, Juan Carlos Moure
  Título: Job Scheduling for Optimizing Data Locality in Hadoop Clusters.
  Publicación: Proceedings. ISBN 978-1-4503-1903-4. Páginas 271-276, Volumen I. Editorial: ACM.
  Local: Madrid, Spain.
  Año: 2013.

Para la gestión de los datos intermedios propusimos utilizar una RAMDISK como una capa de almacenamiento temporal para datos intermedios generados por las aplicaciones Hadoop. Cada nodo del clúster proporciona un segmento de su DRAM para la construcción de una RAMDISK agregada. Los datos intermedios generados entre las fases MAP y Reduce pasaron a ser cargados directamente a la RAMDISK cuando había espacio suficiente para hacerlo, o cargados en el disco cuando no lo había.

La RAMDISK se utiliza para almacenar datos tanto de entrada como datos intermedios generados entre las fases Map y Reduce. Definimos un límite *soft* configurable a partir del cual las nuevas tareas asignadas al nodo comienzan a cargar los datos directamente en el disco duro. Y establecemos un límite *hard* también configurable a partir del cual las tareas que se ejecutan en el nodo y que cargan sus datos a RAMDISK pasan a cargarlos en el disco duro. Como política de tolerancia a fallos mantenemos los datos intermedios en los nodos donde han sido generados hasta la conclusión de cada trabajo.

Shared Input Policy supera significativamente la política de planificación de trabajos por defecto de Hadoop, cuando ocurre la planificación de aplicaciones intensivas en datos que comparten conjuntos de datos de entrada y que manejan grandes volúmenes de datos intermedios. *Shared Input Policy* disminuye el tiempo de *makespan* de trabajos procesados en lotes para estos tipos de aplicaciones. *Shared Input Policy* también se mostró eficiente cuando comparada con otras políticas de planificación de trabajos utilizados en *clusters* Hadoop compartidos, como las políticas *Capacity* y *Fair Scheduler*. Los resultados han sido descritos en el artículo:

- Conference PBio 2014 (Cluster 2014) - International Workshop on Parallelism in Bioinformatics 2014.
  Autores: Aprigio Bezerra, Porfídio Hernández, Antonio Espinosa, Juan Carlos Moure
  Título: Job Scheduling in Hadoop with Shared Input Policy and RAMDISK.
  Publicación: a ser publicado (22 – 26 de Septiembre). Editorial: IEEE
  Local: Madrid, Spain.
  Año: 2014.

5.1 Líneas Abiertas

Como continuad de este trabajo, se ha visto la necesidad de evaluar las prestaciones de *Shared Input Policy* para un conjunto de aplicaciones más grande y con requerimientos más diversificados. Comparar Shared Input Policy con otras políticas de planificación de trabajos Hadoop. Igualmente se hace necesario evaluar el comportamiento de la RAMDISK cuando gestione volúmenes de datos más grandes.

Definir e implementar una política de persistencia de datos para Shared Input Policy que prevenga la volatilidad de la RAMDISK.

Implementar una política dinámica de admisión de lotes.

Abordar el problema del alto costo para acceder a archivos pequeños en Hadoop. Evaluar las técnicas e implementar una política para acceder a estos archivos sin comprometer el tiempo de ejecución de las aplicaciones, y sin generar extensas tablas para el mapeo de las asociaciones de bloque con tareas Map en el nodo *master*.

Evaluar la aplicabilidad de *Shared Input Policy* a la nueva generación de planificadores Hadoop 2.0.

Referencias

[1] R. T. Kouzes, G. A. Anderson, S. T. Elbert, I. Gorton, and D. K. Gracio, “The changing paradigm of data-intensive computing,” *Computer (Long. Beach. Calif).*, vol. 42, pp. 26–34, 2009.

[2] J. Dean and S. Ghemawat, “MapReduce : Simplified Data Processing on Large Clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 1–13, 2008.

[3] J. Shiers, “The Worldwide LHC Computing Grid (worldwide LCG),” *Comput. Phys. Commun.*, vol. 177, pp. 219–223, 2007.

[4] M. Cannataro, D. Talia, and P. K. Srimani, “Parallel data intensive computing in scientific and commercial applications,” *Parallel Comput.*, vol. 28, no. 5, pp. 673–704, May 2002.

[5] L. Ralf, “Google ’ s MapReduce Programming Model — Revisited,” *Sci. Comput. Program.*, vol. 70, pp. 1–30, 2008.

[6] M. de Kruijf and K. Sankaralingam, “MapReduce for the Cell Broadband Engine Architecture,” *IBM Journal of Research and Development*, vol. 53. pp. 10:1–10:12, 2009.

[7] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, “Mars: Accelerating MapReduce with graphics processors,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, pp. 608–620, 2011.

[8] M. Elteir, H. Lin, W. C. Feng, and T. Scogland, “StreamMR: An optimized MapReduce framework for AMD GPUs,” in *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, 2011, pp. 364–371.

[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for multi-core and multiprocessor systems,” in *Proceedings - International Symposium on High-Performance Computer Architecture*, 2007, pp. 13–24.

[10] R. Chen, H. Chen, and B. Zang, “Tiled-MapReduce : Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling,” *Optimization*, pp. 523–534, 2010.

[11] J. Talbot, R. M. Yoo, and C. Kozyrakis, “Phoenix ++ : Modular MapReduce for Shared-Memory Systems,” *Phoenix Usa*, p. 9, 2011.

[12] Q. Zou, X.-B. Li, W.-R. Jiang, Z.-Y. Lin, G.-L. Li, and K. Chen, “Survey of MapReduce frame operation in bioinformatics.,” *Brief. Bioinform.*, vol. 15, pp. 637–647, 2013.

[13] A. Matsunaga, M. Tsugawa, and J. Fortes, “CloudBLAST: combining MapReduce and virtualization on distributed resources for bioinformatics applications,” in *Proceedings - 4th IEEE International Conference on eScience, eScience 2008*, 2008, pp. 222–229.

[14] Z. Meng, J. Li, Y. Zhou, Q. Liu, Y. Liu, and W. Cao, "BCloudBLAST: An efficient mapreduce program for bioinformatics applications," in *Proceedings - 2011 4th International Conference on Biomedical Engineering and Informatics, BMEI 2011*, 2011, vol. 4, pp. 2072–2076.

[15] Y. S. Aulchenko, D.-J. de Koning, and C. Haley, "Genomewide rapid association using mixed model and regression: a fast and simple method for genomewide pedigree-based quantitative trait loci association analysis.," *Genetics*, vol. 177, pp. 577–585, 2007.

[16] M. C. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce.," *Bioinformatics*, vol. 25, pp. 1363–1369, 2009.

[17] L. Pireddu, S. Leo, and G. Zanetti, "Seal: A distributed short read mapping and duplicate removal tool," *Bioinformatics*, vol. 27, pp. 2159–2160, 2011.

[18] T. Nguyen, W. Shi, and D. Ruden, "CloudAligner: A fast and full-featured MapReduce based tool for sequence mapping.," *BMC Res. Notes*, vol. 4, p. 171, 2011.

[19] V. Prasad and G. Loshma, "HPC-MAQ: a parallel short-read reference assembler," *CCSEA 2011, 2011*, p. 84, 2011.

[20] A. Espinosa, P. Hernandez, J. C. Moure, J. Protasio, and A. Ripoll, "Analysis and improvement of map-reduce data distribution in read mapping applications," *The Journal of Supercomputing*, vol. 62. pp. 1305–1317, 2012.

[21] B. Langmead, K. D. Hansen, and J. T. Leek, "Cloud-scale RNA-sequencing differential expression analysis with Myrna.," *Genome Biol.*, vol. 11, p. R83, 2010.

[22] C. L. Hung, Y. L. Lin, C. E. Hsieh, and G. J. Hua, "Efficient protein structure alignment algorithms under the MapReduce framework," in *CloudCom 2012 - Proceedings: 2012 4th IEEE International Conference on Cloud Computing Technology and Science*, 2012, pp. 753–758.

[23] B. T. Rao and L. S. S. Reddy, "Survey on Improved Scheduling in Hadoop MapReduce in Cloud Environments," *Int. J. Comput. Appl.*, vol. 34, pp. 29–33, 2012.

[24] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," *Proc. 5th Eur. Conf. Comput. Syst.*, pp. 265–278, 2010.

[25] S. Seo, I. Jang, K. Woo, I. Kim, J. S. Kim, and S. Maeng, "HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment," in *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2009.

[26] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in MapReduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).*, 2012, pp. 419–426.

[27] M. Zaharia, "Hadoop fair scheduler design document," pp. 1–11, 2009.

[28] T. Apache and S. Foundation, “Capacity Scheduler Guide,” pp. 1–5.

[29] S. Ma, X.-H. Sun, and I. Raicu, “I/O Throttling and Coordination for MapReduce,” 2012.

[30] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat, “Themis: An I/O-efficient MapReduce,” in *Third ACM Symposium on Cloud Computing*, 2012, pp. 13:1–13:14.

[31] D. Moise, T.-T.-L. Trieu, L. Bougé, and G. Antoniu, “Optimizing intermediate data management in MapReduce computations,” *Proc. First Int. Work. Cloud Comput. Platforms - CloudCP ’11*, pp. 1–7, 2011.

[32] D. Zhao and I. Raicu, “HyCache: A User-Level Caching Middleware for Distributed File Systems,” in *27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013, pp. 1997–2006.

[33] “FUSE Project.” [Online]. Available: http://fuse.sourceforge.net.

[34] S.-G. Kim, H. Han, H.-S. Jung, H.-S. Eom, and H. Y. Yeom, “Improving MapReduce Performance by Exploiting Input Redundancy.,” *J. Inf. Sci. Eng.*, vol. 27, no. 3, pp. 789–804, 2011.

[35] Y. Luo, S. Luo, J. Guan, and S. Zhou, “A RAMCloud Storage System based on HDFS: Architecture, implementation and evaluation,” *J. Syst. Softw.*, vol. 86, pp. 744–750, 2013.

[36] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, “The case for RAMClouds: scalable high-performance storage entirely in DRAM,” *ACM SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 92–105, 2010.

[37] M. Hanzich, “Un Sistema de Planificación Temporal y Espacial para Clusters no Dedicados,” Tesis Doctoral, Universidad Autónoma of Barcelona, 2006.

[38] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, “Adaptive parallelism and Piranha,” *Computer (Long. Beach. Calif).*, vol. 28, pp. 40–49, 1995.

[39] R. Hagmann, “Process server: Sharing processing power in a workstation environment.,” in *Sixth International Conference on Distributed Computing Systems*, 1986, pp. 260–267.

[40] A. Barak and O. La’adan, “The MOSIX multicomputer operating system for high performance cluster computing,” *Future Generation Computer Systems*, vol. 13. pp. 361–372, 1998.

[41] M. J. Litzkow, M. Livny, and M. W. Mutka, “Condor-a hunter of idle workstations,” *[1988] Proceedings. 8th Int. Conf. Distrib.*, 1988.

[42] P. Krueger and D. Babbar, “Stealth: a liberal approach to distributed scheduling for network of workstations,” 1993.

[43] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, “SPRITE NETWORK OPERATING SYSTEM.,” *Computer (Long. Beach. Calif).*, vol. 21, pp. 23–36, 1988.

[44] M. Hanzich, P. Hernández, E. Luque, F. Giné, and F. Solsona, “3DBackflling: A space sharing approach for non-dedicated clusters,” in *International Conference on Parallel and Distributed Computing Systems*, 2005, pp. 131–138.

[45] Y. Mao, R. Morris, and M. F. Kaashoek, “Optimizing MapReduce for Multicore Architectures,” *Comput. Sci. Artif. Intell. Lab. Massachusetts Inst. Technol. Tech. Rep.*, 2010.

[46] C. J. Rossbach, Y. Yu, J. Currey, J. Martin, and D. Fetterly, “Dandelion: A compiler and runtime for heterogeneous systems,” in *SOSP13 - 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 49–68.

[47] “Hadoop on Demand.” [Online]. Available: http://hadoop.apache.org/docs/r0.18.3/hod.html.

[48] “Open Grid Scheduler.” [Online]. Available: http://gridscheduler.sourceforge.net/.

[49] “The Condor Project.” [Online]. Available: http://toolkit.globus.org/grid_software/computation/condor.php.

[50] “Torque - Adaptive Computing.” [Online]. Available: http://www.adaptivecomputing.com/products/open-source/torque/.

[51] T. White, *Hadoop: The Definitive Guide*, Second Edi. O’Reilly Media, 2010.

[52] C. LAM, *Hadoop in Action*, First Edi. Manning Publications, 2011.

[53] T. Sandholm and K. Lai, “Dynamic proportional share scheduling in hadoop,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, vol. 6253 LNCS, pp. 110–131.

[54] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, “Improving MapReduce Performance in Heterogeneous Environments.,” *OSDI*, pp. 29–42, 2008.

[55] “Amazon Elastic Compute Cloud.” [Online]. Available: http://aws.amazon.com/ec2.

[56] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, “SAMR: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment,” in *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*, 2010, pp. 2736–2743.

[57] H. Herodotou, H. Lim, G. Luo, N. Borisov, and L. Dong, “Starfish : A Self-tuning System for Big Data Analytics,” *CIDR*, vol. 11, pp. 261–272, 2011.

[58] “Starfish Project.” [Online]. Available: http://www.cs.duke.edu/starfish/.

[59] S. H. S. Huang, J. H. J. Huang, J. D. J. Dai, T. X. T. Xie, and B. H. B. Huang, “The HiBench benchmark suite: Characterization of the MapReduce-based data analysis,” *Data Eng. Work. (ICDEW), 2010 IEEE 26th Int. Conf.*, 2010.

[60] H. Li, J. Ruan, and R. Durbin, “Mapping short DNA sequencing reads and calling variants using mapping quality scores.,” *Genome Res.*, vol. 18, pp. 1851–1858, 2008.

[61] R. A. Baeza-Yates and C. H. Perleberg, “Fast and practical approximate string matching,” *Inf. Process. Lett.*, vol. 59, pp. 21–27, 1996.

[62] R. Li, Y. Li, K. Kristiansen, and J. Wang, “SOAP: short oligonucleotide alignment program.,” *Bioinformatics*, vol. 24, pp. 713–714, 2008.

[63] A. D. Smith, Z. Xuan, and M. Q. Zhang, “Using quality scores and longer reads improves accuracy of Solexa read mapping.,” *BMC Bioinformatics*, vol. 9, p. 128, 2008.

[64] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg, “Searching for SNPs with cloud computing.,” *Genome Biol.*, vol. 10, p. R134, 2009.

[65] S. J. Matthews and T. L. Williams, “MrsRF: an efficient MapReduce algorithm for analyzing large collections of evolutionary trees.,” *BMC Bioinformatics*, vol. 11 Suppl 1, p. S15, 2010.