Universidad Nacional Autónoma de México

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRUEBAS DE RENDIMIENTO EN UN SISTEMA DISTRIBUIDO MEDIANTE EL MODELO MAP-REDUCE

T E S I N A

Que para optar por el grado de:

ESPECIALISTA EN CÓMPUTO DE ALTO RENDIMIENTO

P R E S E N T A

SANTIAGO CRUZ PRADO

TUTOR

HÉCTOR BENÍTEZ PÉREZ

INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS Y EN SISTEMAS

CIUDAD UNIVERSITARIA, CIUDAD DE MÉXICO. ENERO, 2024.

Agradecimientos

Hacer mención al Dr. Héctor Benítez por la orientación y asesoría brindada para este proyecto. De igual manera al Dr. Ernesto Rubio por su tiempo para la revisión de este trabajo así como al profesor Adrián Durán de la Especialidad por el apoyo con el equipo de los servidores utilizados.

Índice General

Índice de Figuras

Índice de Tablas

Tablas de resultados

Introducción

Es evidente que nuestra era se ha convertido en una tendencia dependiente de la constante generación y procesamiento de información que es difícil asimilar algún área que no haya utilizado o tomado parte del cómputo. Al tiempo que desarrollamos y abrimos puertas a nuevas ramas de investigación, generamos nueva información, ya sea como una recopilación de distintas fuentes o como el resultado de analizarla con el fin de obtener alguna relación o patrón que nos brinde claridad en donde los datos primarios no muestren directamente. Es así que estos datos primarios recopilados en unidades de almacenamiento llega a crecer a grandes tamaños que en algunas situaciones puede ser interpretada como información desperdicio ya que no es utilizada mas allá de una consulta. El almacenamiento de datos despierta interés cuando se puede obtener provecho de este, aunque cuando sucede, se convierte en una necesidad de recopilar más información y con ello la tarea en estrategias para el almacenamiento. Aunque hoy en día se cuenta con distintas maneras de almacenamiento, el siguiente reto es poder maquinar (procesar) esta enorme cantidad de datos.

Este trabajo muestra la forma de poder implementar un framework dedicado para este propósito al procesar archivos de gran tamaño en donde las herramientas tradicionales se encuentran con un cuello de botella al transmitir los datos a las unidades de procesamiento. Por esta razón, el framework Hadoop, mediante técnicas del cómputo distribuido, logra tiempos de respuesta record para distintos propósitos. Aunado a la dinámica bajo la cual el framework se rige, Hadoop sirve como la base sobre la cual operan distintos sistemas, creando de esta forma un ecosistema basado sobre este framework.

Dados los distintos sistemas que soporta Hadoop, este escrito se enfoca hacia la utilización del modelo de programación MapReduce debido a que los experimentos a realizar brindan la información previamente almacenada que se pretende procesar en forma de lote.

Para este trabajo se siguió la siguiente estructura; el primer capítulo describe el panorama general para el cual este framework esta principalmente elaborado, así como algunas tecnologías alternativas para abordar el desafío en el procesamiento de grandes volúmenes de datos, enfocándose principalmente en la distribución de tareas. En el capítulo 2 se presenta el ecosistema Hadoop y describe su arquitectura, así como la forma en que coordina la ejecución de una aplicación enfocado en el modelo MapReduce. Para demonstrar la forma de ejecutar aplicaciones en este framework distribuido, el capitulo 3 presenta dos implementaciones utilizando el patrón de programación MapReduce analizando cada algoritmo así como los tiempos de ejecución. Llevando a cabo tales implementaciones, en el capítulo 4 se redactan los experimentos realizados con el equipo

proporcionado para configurar el cluster Hadoop y las líneas principales para preparar la aplicación, o *Job,* desde la clase principal, también conocida como *Driver*. Los resultados de la experimentación se describen en el capítulo 5. Se concluye en la última sección con los puntos más relevante extraídos del proceso completo de este trabajo. Se brindan los códigos utilizados en este trabajo en el apartado de Anexos, recordando que los tamaños de las matrices de entrada son modificados únicamente desde la clase controladora *Driver*.

Capítulo 1

Panorama. Procesamiento de datos masivos

Actualmente, nuestra sociedad continúa generando una cantidad masiva de información de distintas fuentes ya sea por humanos, por equipos de cómputo o por la misma naturaleza, producida de forma estructurada, semi-estructurada o no-estructurada. Debido a la enorme cantidad de datos, no es factible realizar el almacenamiento con herramientas habituales como sistemas de gestión de bases de datos relacional (RDBMS) debido a las limitantes en el procesamiento como consultas tipo SQL. (Ishwarappa & Anuradha, 2015).

Lo que globalmente se conoce como "Big Data" se ha convertido en un desafío para aquellos interesados en las áreas del cómputo que buscan explotar la recopilación de distintas fuentes.

1.1. Big Data.

Datos masivos o Big Data no significa los datos por completo dado un enfoque o temática, ya que sería una tarea interminable recolectar todos los datos para un enfoque en particular. Sin embargo, cantidades masivas de datos proporciona una muestra mucho más larga que datos en menor escala. Conforme más grande sea la muestra más robusto será(n) el(los) resultado(s). Entre los distintos desafíos para el procesamiento de datos masivos se encuentran (Shi, 2022):

- La Transformación de Datos Semi-Estructurados y No-Estructurados en un Formato Estructurado. Se refiere al cambio de estado en que los datos fueron tomados de las fuentes a fin de ser acoplados para un análisis en particular, es decir, orientados al objetivo.
- Explorar la complejidad, incertidumbre y Modelado Sistemático de Big data. Se refiere a la variedad en los tipos de datos que pueden tener un objetivo común pero las fuentes y el análisis es manejado sobre un ambiente distinto, buscando sistematizar las distintas situaciones a abordar.
- Explorar la relación de Heterogeneidad de Datos, Heterogeneidad del Conocimiento y Heterogeneidad de Decisión. Hace referencia a las distintas etapas y niveles en la estructuración de los datos en diferentes áreas y jerarquías. Debido a la variedad de fuentes y un objetivo general, diversas áreas y niveles del conocimiento se unen para un propósito en específico. Por esto, existe la heterogeneidad en decisión debido a los distintos niveles y formas de abordar un objetivo final, en donde las decisiones podrían estar altamente definidas (estructuradas), o no por completo (No estructuradas o Semi-estructuradas) dando cierto grado de incertidumbre en la toma de decisión.

Big Data puede ser descrita a partir de 5 características, llamadas las 5 V's (Naeem, et al., 2019) :

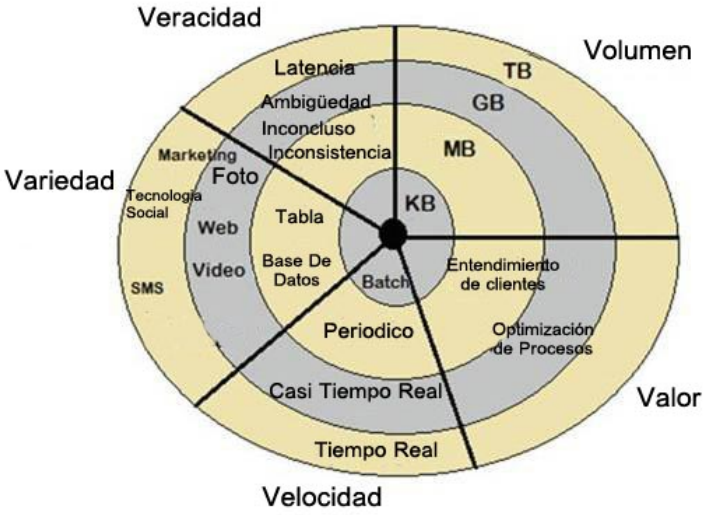


Figura 1. Características de Big Data (Naeem, et al., 2019)

- **Volumen**. Se refiere a la masiva cantidad de datos generados en cualquier momento de distintas fuentes; sensores, teléfonos móviles interfaces de redes sociales, etc. Aunque es un desafío procesar toda esta gran cantidad al mismo tiempo, es posible utilizar herramientas de almacenamiento de Big Data.
- **Valor**. En este aspecto, el valor de los datos significa qué tan útil es la vasta información con la que se cuenta, por lo que sin procesarla es información desperdicio.
- **Veracidad.** Relacionado a la calidad, precisión y valor de la información para cierto propósito, resolviendo los problemas de inconsistencias, anormalidades, duplicación entre otros.
- **Velocidad**. Aquí, se enfoca en la tasa en que los datos se mantienen en crecimiento.
- **Variedad.** Hace referencia a los diferentes tipos de datos y la heterogeneidad al buscar un fin en particular. Se toma en cuenta que los datos son extraídos en distintos formatos, por ejemplo archivos pdf, csv, png, etc.

1.2. Data Mining

Lo que algunos conocen como "minería de datos" (o globalmente "Data Mining"), requiere de estrategias para realizar análisis de una forma óptima que de otra forma, poder predecir u obtener resultados con anticipación se vuelve inútil para ciertos casos. Y cuando hablamos de optimizar, el Cómputo de Alto Rendimiento (High Performance Computing) es la estrategia primaria a considerar.

Básicamente, la minería de datos es el proceso de descubrir patrones interesantes, modelos, u otro tipo de conocimiento en un conjunto de datos masivos. Fue en la década de los 90's que se comenzó a

utilizar el término Minería de Datos (Data Mining), que podría ser intercambiable con el término "Descubrimiento de conocimiento en Bases de Datos" (KDD acrónimo en inglés). Por lo que actualmente la minería de datos se ha convertido en la tecnología para el análisis de datos en intersección de la intervención humana, el aprendizaje de máquina, el modelado matemático y las bases de datos (Han, Pei, & Tong, 2022).

En este contexto, el proceso en el descubrimiento de algún nuevo conocimiento es generalmente (Han, Pei, & Tong, 2022) :

1. Preparación de los datos.
    - Limpieza de los datos, removiendo elementos indeseables e inconsistencias.
    - Integración de datos de múltiples fuentes.
    - Transformación de los datos y su consolidación apropiadamente para ser minada.
    - Selección de los datos en donde la información es relevante de las fuentes
2. Minería en los datos. Proceso en donde métodos inteligentes son aplicados para extraer patrones o construir modelos.
3. Evaluación del modelo, identificando patrones o modelos basándose en medidas de interés.
4. Presentación del nuevo conocimiento, usando técnicas de representación para mostrar al usuario.

Esta minería puede ser aplicada a cualquier tipo de datos mientras la información sea significativa para la aplicación que se pretenda. Sin embargo, distintos tipos de datos podrían necesitar diferentes metodologías sean simples o más sofisticadas. Así, los datos pueden utilizarse desde un extremo en forma estructurada, por ejemplo, estructuras tipo tabla como datos de matrices, cubos de datos o bases de datos relacionales, hasta no estructurada, como datos de texto y multimedia (audio, video, etcétera) aplicados a campos como el entendimiento del lenguaje natural, visión computacional y reconocimiento de patrones.

Algunos de los enfoques comúnmente utilizados al realizar minería de datos se encuentra; la clasificación, usando árbol de decisiones, red neuronal con peso en las conexiones, clasificación bayesiana o el k-vecino más cercano; regresión lineal, para la predicción de resultados posteriores, hasta análisis por clustering, agrupando datos sin revisar etiquetas sobre estos (Han, Pei, & Tong, 2022).

1.3. Cómputo Distribuido

1.3.1. Tecnologías de cómputo para análisis de Big Data.

Debido a las actuales tendencias que demandan manejar enormes cantidades de información constantemente en crecimiento, en orden de terabytes, petabytes y mayores cantidades, no puede ser logrado satisfactoriamente por una sola unidad de cómputo, con lo que se exigen mayores requerimientos en el poder de procesamiento y en tiempo considerable de ejecución. El procesamiento de Big Data implica entonces de una determinada configuración en el sistema de almacenamiento, división o repartición de tareas, distribución de la carga de cómputo y seguridad

cuando algún equipo de procesamiento falla. En términos simples, el cómputo distribuido es el uso de múltiples equipos compartiendo tareas de cómputo, siendo una de las principales soluciones actualmente (Wang, Xu, Zhang, & Zhong, 2022).

Debido a esta distribución en las tareas de cómputo, es posible paralelizar un trabajo de procesamiento, dígase principal o global, de forma que tales **tareas se realicen independientemente** generando resultados que al final sean **globalmente coherentes**.

En este sentido, el ámbito de cómputo paralelo y cómputo distribuido tiende a pensarse indistintamente, pero no es así. Aunque es complicado contar con una clara distinción entre estos dos conceptos, algunos consideran el cómputo paralelo como objetivo primordial el incrementar el rendimiento mientras el cómputo distribuido se enfoca en la tolerancia a fallos. Alternativamente, se considera a los sistemas paralelos en ambientes sincronizados de única instrucción-múltiples datos (SIMD) o múltiples instrucciones-múltiples datos (MIMD), mientras los sistemas distribuidos se inclina hacia procesos cooperantes con direcciones de memoria privada (Ghosh, 2014).

El Computo Distribuido es una extensa y creciente área de estudio, referente a diferentes tipos de configuraciones en donde distintos procesadores trabajan en paralelo usualmente en unión hacia un problema en particular. Uno de los modelos en el cómputo distribuido de gran importancia es el modelo de paso de mensajes, en donde una red en comunicación puede ser representada por un grafo de n-vértices en donde cada uno es un procesador independiente, comunicándose uno a otro mediante enlaces o bordes en el grafo.

Cerca del año 2004 tres propuestas respecto al sistema de archivos distribuido, modelo de computo paralelo y un sistema de almacenamiento de datos no-relacional fueron por primera vez expuestos como una alternativa al procesamiento distribuido de Big Data. Fueron los ingenieros de los motores de búsqueda con mayor influencia, Doug Cutting y Mike Cafarella que desarrollaron la propuesta Hadoop, y que posteriormente, a fin de mejorar esta propuesta, emergen diversos frameworks que fueron aplicados en ambientes distribuidos, como nos muestra la Figura 2.

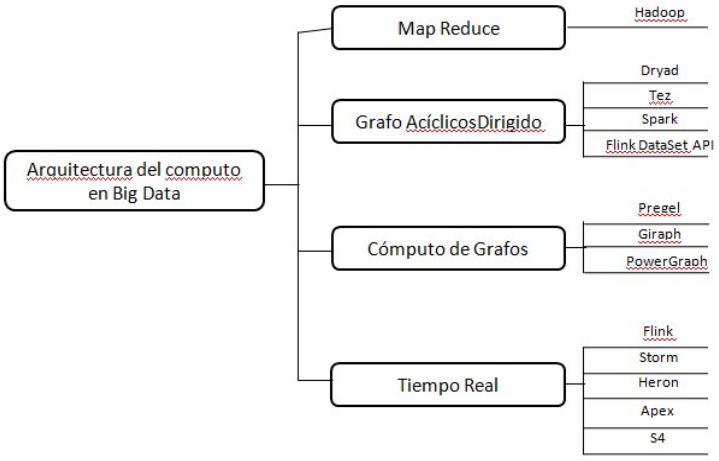


Figura 2. Frameworks para el Cómputo Distribuido para el manejo de Big Data. (Wang, Xu, Zhang, & Zhong, 2022)

El sistema Hadoop está compuesto de dos partes; Hadoop Distributed File System (HDFS) y la parte del Cómputo, inicialmente MapReduce la cual migró la forma de gestionar los recursos y la programación de tareas cambiando a su actual nombre como YARN, abriendo con este cambio la posibilidad a frameworks alternativos que operen en capas superiores sobre Hadoop, como se describirá más adelante. Los pasos esenciales de procesamiento son los siguientes:

1. El archivo de entrada es dividido y a cada división le corresponderá una tarea de nombre *Map*. Esto es, a cada unidad de procesamiento le corresponde una división o *pedazo* del archivo original, y el procesamiento respectivo.
2. Se realiza el procesamiento de los datos por división (tareas *Map*), considerando que el framework interpretará los datos asignados en una estructura clave-valor.
3. Al terminar el procesamiento, cada tarea *Map* devuelve el resultado siguiendo la misma estructura de datos, clave-valor, la cual, gracias al mismo framework, es agrupada y ordenada por clave. El resultado intermedio se envía a la etapa *Reduce* de forma balanceada para equilibrar la carga sobre el total de estas tareas.
4. Cada tarea *Reduce* procesará una parte del total de claves generadas con sus valores respectivos. La cantidad de tareas *Reduce* se especifica desde la configuración del cluster Hadoop mediante una propiedad llamada *mapreduce.job.reduces* y puede cambiar dependiendo del enfoque y criterio del programador o del propósito.
5. Al finalizar la etapa *Reduce*, el resultado es devuelto en un formato HDFS.

De esta forma puede ser implementado el framework de programación MapReduce para procesar datos en el orden superior a terabytes. Así, podemos hablar de ejecución por *lote* o *batch processing*, es decir, el dataset completo de información es procesado una sola vez y no consecutivamente. Opuesto al modelo de ejecución en batch se cuenta con la ejecución en flujo, o *stream processing*, usualmente bien ajustado para el procesamiento en datasets de menor tamaño. La Tabla 1muestra una comparativa de ambos modelos de ejecución:

Tabla 1. Procesamiento por lote vs Procesamiento en flujo (Wang, Xu, Zhang, & Zhong, 2022)

Características	Procesamiento por Lote	Procesamiento en flujo
Alcance de los datos	Consultas y/o procesamiento en todo o mayoría de información del dataset	Consultas y/o procesamiento en los datos dentro de un determinado tiempo o únicamente los registros más recientes
Tamaño de los datos	Grandes Lotes de Datos	Registros Individuales o pequeños lotes compuestos de pocos registros
Rendimiento	Tiempos de latencia en minutos a horas	Requiere tiempo de latencia en el orden de segundos ó milisegundos
Análisis	Análisis Complejo	Funciones de respuesta simple, y metricas cambiantes

Es importante mencionar que algunos frameworks permiten una mezcla de las características entre

Batch Processing y Stream Processing como Flink y Spark dentro del ambiente Apache:

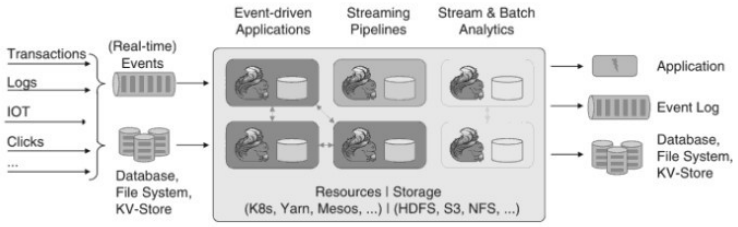


Figura 3. Arquitectura Básica de Apache Flink (Apache Flink®, 2023)

En la siguiente sección se detalla la forma de operar de este ambiente en sus elementos esenciales y su operación para coordinar la distribución de las tareas paralelas con el propósito de tener una base solida, para el caso de este trabajo, en la programación bajo el framework de MapReduce. Por supuesto, entender tal coordinación de las tareas es de gran importancia si se pretende desarrollar alguna aplicación bajo algunos de los ambientes que operan bajo hadoop; Spark, Flink, Pig, Sqoop, Flume, entre otros.

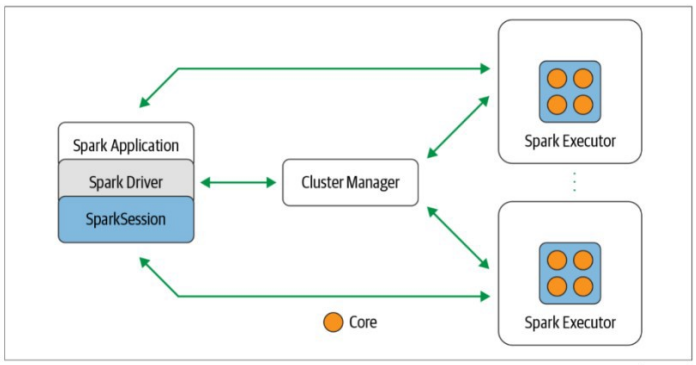


Figura 4. Arquitectura Básica de Spark (Jules S. Damji, 2020)

Capítulo 2

Hadoop - Ambiente de sistema distribuido.

Es importante recordar el objetivo que ofrece el framework de MapReduce en Hadoop para lograr un mayor rendimiento en el procesamiento. A menudo para conseguir un mayor rendimiento se busca paralelizar o dividir nuestro trabajo en distintas partes o tareas de forma que estas puedan ser distribuidas hacia ciertas unidades que operen con cierto grado de independencia de las demás, que en algunos casos no es posible la autonomía completa y se deba compartir entre estas unidades alguna abstracción como puede ser información (localidades de memoria) o recursos, por lo que estrategias de concurrencia toman parte y con ello incertidumbre de una arbitraria solución tomada.

Debido a lo anterior, con el fin de evitar la incertidumbre de compartir recursos y distribuir el procesamiento, el patrón que encontramos es en la arquitectura comúnmente conocida como *Manager - Workers*, donde la entidad principal, el Maestro, distribuye el procesamiento hacia diferentes unidades de forma que no se requiera comunicación entre ellas y devuelvan el resultado a la entidad inicial. Algunas soluciones como el uso de interfaces que permitan la comunicación entre distintos procesos como MPI o PVM con flexibilidad sobre distintos lenguajes de programación, implican una curva de aprendizaje e implementación que si bien podría ser comprendida, en ciertos casos debe tomar en cuenta una enorme cantidad de información para ser repartida, por lo que, además del procesamiento, primordialmente los datos deben ser distribuidos. La solución que nos permite ambas particiones es el proyecto Hadoop, el cual permite evitar la inherente incertidumbre en impredecibles estrategias de concurrencia y por supuesto siguiendo el patrón *Manager - Workers*.

2.1. Hadoop

Hadoop es considerado un ecosistema complejo debido a la variedad de frameworks (sqoop, spark, etcétera) que operan sobre su estructura base, HDFS y YARN, para el procesamiento de una cantidad masiva de información.

2.1.1. Ecosistema Hadoop

El ecosistema Hadoop está compuesto de diversos subproyectos pensados para operar sobre un clúster. Esto debido a que Hadoop es un sistema de código abierto y por lo tanto su creciente popularidad en muchas contribuciones y mejoras por distintas organizaciones.

Las siguientes son algunas de las implementaciones que trabajan en capas superiores a Hadoop para distintos propósitos. La Figura 5 comparte un mapa más completo de la diversidad de frameworks y componentes de Hadoop:

- **MapReduce**: Se trata del estilo de programación nativo de Hadoop considerando una programación distribuida. Basado en la propuesta solución de (Dean & Ghemawat, 2004) en donde se consideran dos etapas de programación pilares: *Map y Reduce*.
- **Hive**: Se trata de un sistema de infraestructura para Hadoop que crea una envoltura tipo SQL sobre capa de procesamiento MapReduce llamada HiveQL.
- **Spark**. Proporciona una poderosa alternativa a MapReduce como framework para procesamiento de datos en paralelo que se ejecuta alrededor de 100 veces más rápido que Hadoop MapReduce en memoria. Comúnmente usado para procesamiento en *stream* de tiempo real.
- **Flume**. Se trata de una herramienta para recolectar, agregar y mover eficientemente grandes cantidades de datos, mayormente usado del tipo *logs*, de distintas fuentes hacia un almacenamiento centralizado.
- **Sqoop**: Es una herramienta de código abierto que permite a los usuarios extraer datos generalmente de almacenamiento de datos estructurado, como sistemas de gestión de bases de datos relacionales(RDBMS), hacia Hadoop para su procesamiento y al termino devolverlos al formato de su fuente original. Esto mediante la ayuda de conectores.

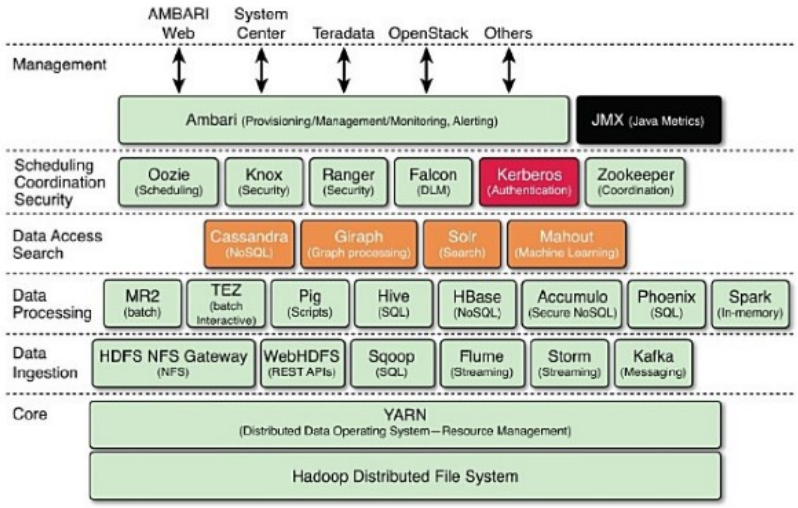


Figura 5. Frameworks y componentes que operan sobre Hadoop (Trujillo, Kim, Jones, Garcia, & Murray, 2015)

2.2. Arquitectura.

Como se mencionó en el capítulo introductorio, los componentes básicos de este framework son HDFS (Hadoop Distributed File System) y actualmente YARN (Yet another Resource Negotiator), previamente manejado por una arquitectura llamada MapReduce.

2.2.1. HDFS

Se trata del sistema de archivos núcleo de Hadoop. Este sistema de archivos está diseñado para almacenar datos de gran tamaño ejecutándose en un clúster de hardware de propósito general. Tales archivos son idealmente destinados en HDFS en el orden de cientos de megabytes, gigabytes, terabytes e incluso clúster almacenando peta bytes de información. El hardware que compone un clúster con la implementación de Hadoop está pensado en términos de propósito general debido a las fallas que cualquier unidad de procesamiento (de ahora en adelante nodo) pueda presentar al operar un clúster y pueda ser reemplazado con facilidad sin fabricación especial.

2.2.1.1. Bloques en HDFS

De forma similar a los bloques utilizados en los sistemas de archivos tradicionales, HDFS almacena la información de la misma forma en bloques, con la distinción que estos últimos están pensados con tamaños mayores, por defecto en 128 MB. Esto es, cada archivo de información que es ingresado en HDFS es dividido en fracciones, es decir en bloques que son almacenados de forma independiente en cada nodo. Una característica de este sistema es que a pesar de que el tamaño del archivo sea menor a 128MB, el bloque no es rellenado con información basura, por lo tanto el archivo se mantiene con el tamaño original.

Bajo esta implementación tipo bloque, se asegura la consistencia de la información y tolerancia a fallos mediante replicación. Esto es, cada bloque es replicado en nodos distintos separados del primero con el propósito de mantener consistencia en el archivo general desde la perspectiva del usuario en caso que el nodo que contiene el bloque original deje de reportarse como activo en el clúster por cualquier situación de falla. Si esto sucede, por medio de las replicas se vuelve a generar los bloques repetidos necesarios a fin de ajustarse al número de replicación establecido en otros nodos. Esta configuración de replicación puede ser modificada de la preestablecida (3 replicas), con la consecuencia de menor tolerancia al reducir este número o, por el contrario, una menor capacidad de almacenamiento al incrementarla.

2.2.1.2. Elementos de HDFS

La coordinación del sistema HDFS, se lleva a cabo mediante dos procesos fundamentales; namenode y datanode, a forma de seguir el patrón Maestro (namenode) – Trabajador (datanode) en Hadoop.

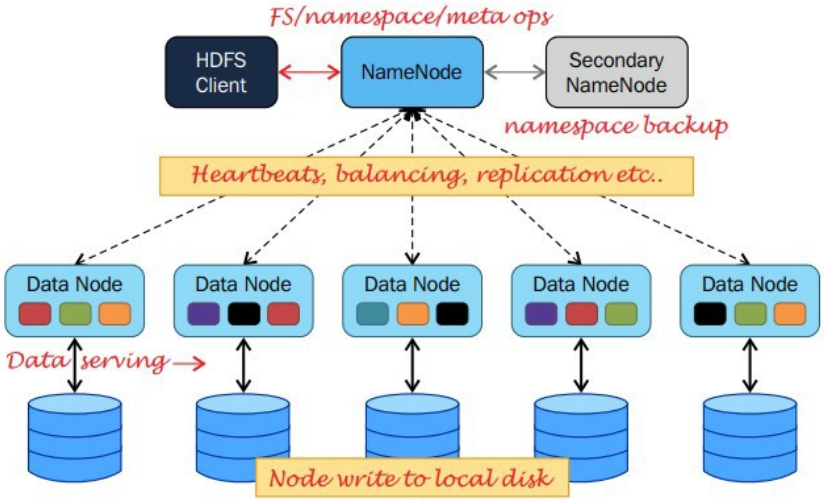


Figura 6. Arquitectura HDFS (Achari, 2015)

- **Namenode**: Es el proceso maestro que coordina las operaciones relacionadas al almacenamiento, como la escritura y lectura en HDFS. Debido a que es responsable de estas operaciones, es crítica su funcionalidad. Contrario al proceso *datanode*, es un único punto de falla de todo el clúster Hadoop. Debido a este punto crítico, Hadoop puede ser configurado con funcionalidad de *Alta disponibilidad*, en cuyo caso, un *namenode* en stand-by puede tomar su lugar en caso de fallo. Este proceso mantiene los metadatos de todos los bloques así como en qué nodo están ubicados dentro del clúster. Tales metadatos que gestiona el *namenode* es almacenada de dos formas: *Namespace Image* y *Edit Log*, en la memoria de acceso aleatorio, la cual, después de determinado periodo, es agregada y almacenada en la memoria principal y transferida al *Secondary Namenode*.
- **Secondary/Checkpoint Namenode**: Es un proceso que almacena los metadatos del *namenode* en caso que este falle y al ser reemplazado, el *secondary namenode* se encarga de compartir tales metadatos al nuevo *namenode*, por lo cual solo mantiene un registro del estado del *namenode*, mas no es un reemplazo de este.
- **Datanode**: Es el proceso que se encarga de coordinar el almacenamiento del bloque que le es asignado por el *namenode* así como la replicación de este bloque hacia otros nodos. Para mantener coordinación con el *namenode*, el *datanode* le envía mensajes, llamados *heartbeats*, en intervalos periódicos. Si los *heartbeats* fallan en llegar al *namenode*, este último asumirá tal *datanode* como inactivo.

2.2.2. Arquitectura YARN

Este componente es el gestor de recursos del cluster Hadoop. Esta versión fue introducida en Hadoop 2 con el propósito de mejorar la implementación previa - MapReduce - siendo lo suficientemente general para soportar otros paradigmas del cómputo distribuido. Como se introdujo, el ecosistema Hadoop está compuesto de diversos frameworks que operan gracias a la capa de

YARN, esto es, un usuario puede escribir código pensado para utilizarse en un ambiente distribuido, en un alto nivel mediante API's las cuales por sí mismas están compuestas en YARN, ocultando los detalles de gestión de recursos al usuario. La Figura 7 muestra tales frameworks y su lugar dentro del ambiente.

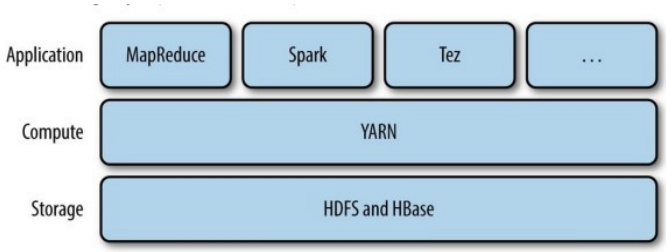


Figura 7. Aplicaciones de YARN (White, 2015)

Incluso algunos frameworks no interactúan directamente con la capa de YARN, sino sobre alguna capa posterior como MapReduce o Spark como es el caso de Pig, Hive, etc (White, 2015).

2.2.2.1. Componentes YARN

Los elementos principales de los que YARN está compuesto en un patrón similar a HDFS en *Master-Workers* son: *ResourceManager*, únicamente uno por clúster, y *NodeManager*, ejecutándose usualmente en el resto de nodos para monitorear sus respectivos *contenedores*. Para entender mejor la operación de YARN, obsérvese la Figura 8 a continuación.

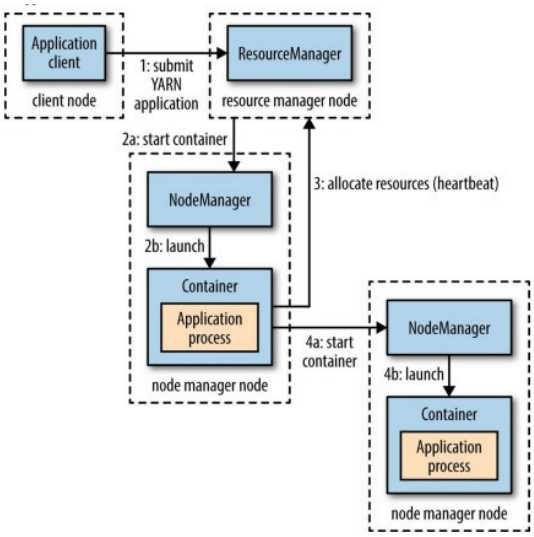


Figura 8. Proceso de ejecución en YARN (White, 2015)

- **ResourceManager**: En términos de procesamiento, se encarga de gestionar los recursos en todo el clúster. Recibe la solicitud para ejecutar un trabajo de procesamiento del cliente y

procede a solicitar la creación de un proceso llamado *Application Master* al *nodemanager* para ejecutar tal trabajo en recursos reservados llamados *contenedores*.

- **Nodemanager**. Se encarga de crear y monitorear el contenedor para ejecutar el *Application Master*. De la misma forma, reporta el progreso del trabajo o tarea asignada particularmente a este nodo.
- **Application Master**. Este subproceso alojado en el nodo *Worker*, coordina el progreso de tareas que creadas para cada nodo respecto al bloque que tienen alojado. Si el *Application Master* determina que para llevar a cabo el trabajo global (*job*) es necesario crear otro *contenedor* lo reporta a fin de ordenar la creación de tal en otro nodo.

Al igual que la versión previa a YARN, MapReduce también coordina a través de dos procesos principales *Job Tracker* como el proceso Master, y *Tasktracker* como el proceso Worker. En la Tabla 2 se puede observar las responsabilidades equivalentes entre los dos sistemas.

Tabla 2. Equivalencia de Responsabilidades MapReduce vs YARN (White, 2015)

MapReduce	YARN
Job Tracker	ResourceManager, Application Master, Timeline Server
Tasktracker	NodeManager
Slot	Container

Además de la ventaja que tiene YARN como soporte para distintos frameworks operando sobre esta capa, MapReduce cuenta con una escalabilidad de 4,000 nodos y 40,000 tareas, esto debido a que el *Job Tracker* administra tanto las tareas como a los Jobs* que son solicitados. En contraste a YARN que soporta hasta 10,000 nodos y 100,000 tareas.

2.2.2.2. Ejecución de Jobs en YARN

Un clúster Hadoop, por lo general no es utilizado para un solo trabajo en específico y al termino de este trabajo utilizarlo sólo cuando exista uno nuevo, esperando inactivo. Por el contrario, existen diversas solicitudes cliente que de forma continua son subidas al clúster y la coordinación en la ejecución es un tema primordial a fin de no sobresaturar o mantener tales solicitudes en espera por periodos indefinidos.

Para abordar esta problemática, YARN proporciona tres modos para planificar la ejecución de cada solicitud que se tocarán en este texto de forma general: *FIFO*, *Capacity Scheduler* y *Fair Scheduler*.

* Job se refiere al proceso global que se ingresa en el clúster de Hadoop general, al contrario de Tareas que se refiere a los sub-procesos que componen al Job Global

- **FIFO** es apropiado para clústers que no se comparten por multitud de usuarios y por lo tanto no necesita configuración en especial. En este caso, todos los recursos disponibles en el clúster se enfocan en la tarea actual en ejecución, por lo que si una segunda solicitud es subida al clúster, deberá esperar el término de la primera.

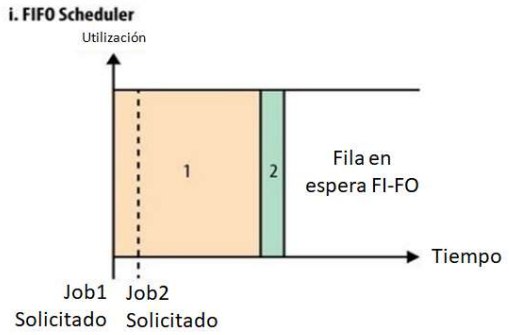


Figura 9. Diagrama Tiempo vs Recursos para FIFO (White, 2015)

- **Capacity Scheduler** es apropiado donde se tiene un escenario de reserva de recursos de acuerdo a distintas organizaciones o grupos de usuarios. Esto es, para cierto grupo se podría tener reservado un límite de recursos a utilizar dentro del clúster, mientras otro grupo podría tener asignado otro porcentaje de recursos, quizá menor que el primero, y sucesivamente de esta forma para grupos adicionales. A cada reserva de recursos asignada por organización, se agenda trabajos (*Jobs* enteros) solicitados al clúster, es decir, dentro de los límites en recursos se aceptan solicitudes que se irán encolando hasta el término de la solicitud previa.

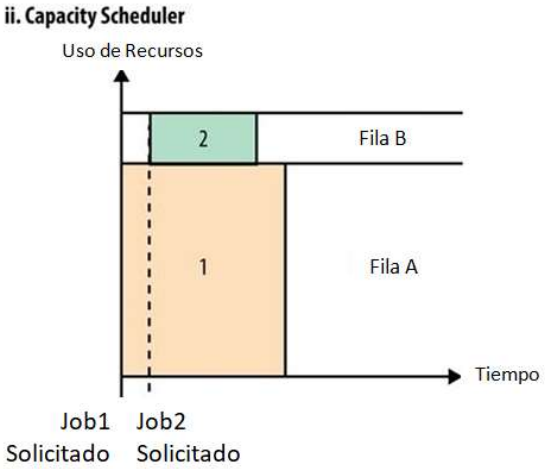


Figura 10. Diagrama Tiempo vs Recursos para Capacity Scheduler (White, 2015)

- **Fair Scheduler** organiza la ejecución de los *Jobs* en el clúster utilizando todos los recursos disponibles para un *Job* que es solicitado con el clúster totalmente disponible. Cuando un segundo *Job* solicita recursos en el clúster, se le asigna la mitad de recursos de forma que no

deba esperar el término del *Job* previo, logrando que ambos sean ejecutados al mismo tiempo pero con mitad de recursos. En caso que un tercer *Job* sea solicitado por el mismo usuario que solicitó el primer *Job*, en tal caso, los recursos asignados para ese usuario son divididos por mitad a fin de ejecutar al mismo tiempo los tres *Jobs* simultáneamente. Por lo que ahora cada *Job* solicitado por el primer usuario tendrá asignado un cuarto de recursos para su ejecución inmediata, mientras el segundo *Job* subido por el segundo usuario al clúster permanece con la mitad de recursos globales del clúster.

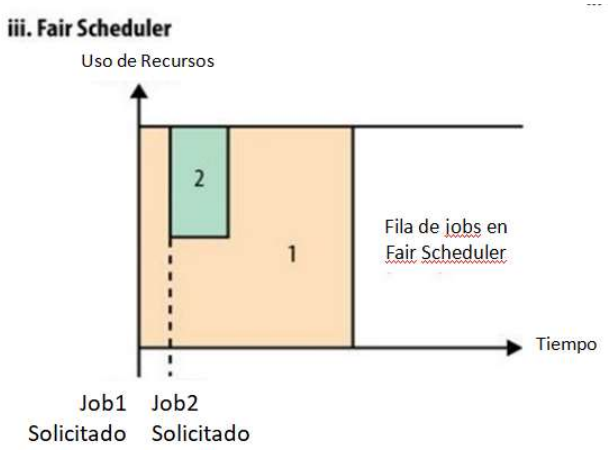


Figura 11. Diagrama Tiempo vs Recursos para Fair Scheduler (White, 2015)

- Bajo la configuración de Fair Schaduler, es posible la preanticipación (preempting) de *Jobs* con la desventaja de terminar *Jobs* incompletos para iniciar aquellos con mayor pre anticipación.

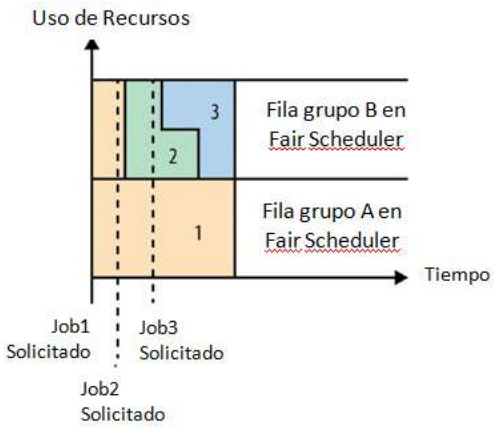


Figura 12. Diagrama Tiempo vs Recursos para Fair Scheduler con Preanticipación (White, 2015)

2.3. MapReduce.

2.3.1. Framework de ejecución y Modelo de programación MapReduce.

Previamente se detalla el framework de ejecución que Hadoop utiliza en la versión reciente, YARN, y el que se utilizó en su primera versión, MapReduce. En este contexto, aunque no es común utilizarlo actualmente después de Hadoop 2, es importante precisar que el término MapReduce se refiere a dos distintos conceptos que están relacionados; El framework de ejecución y el Modelo de Programación. El primero, trata con la manera de coordinar la ejecución de los programas que son escritos sobre este sistema. Como se comparó anteriormente en la Tabla 2, esta coordinación se realiza mediante dos procesos; *Jobtracker* y *TaskTracker*. El segundo, Modelo de Programación, es la descripción del código o especificación a llevar a cabo en el procesamiento distribuido en un dataset dividido en distintos nodos (Lin & Dye, 2010).

Como una referencia para entender esta diferencia, tomemos el diagrama de la Figura 13 de la publicación de (Dean & Ghemawat, 2004) proponiendo la operación de este modelo MapReduce. Desde el punto de vista del programador, las flechas solidas explican el flujo al describir las dos etapas de MapReduce, mientras las flechas discontinuas sirven para comprender el framework de ejecución, es decir la asignación de tareas.

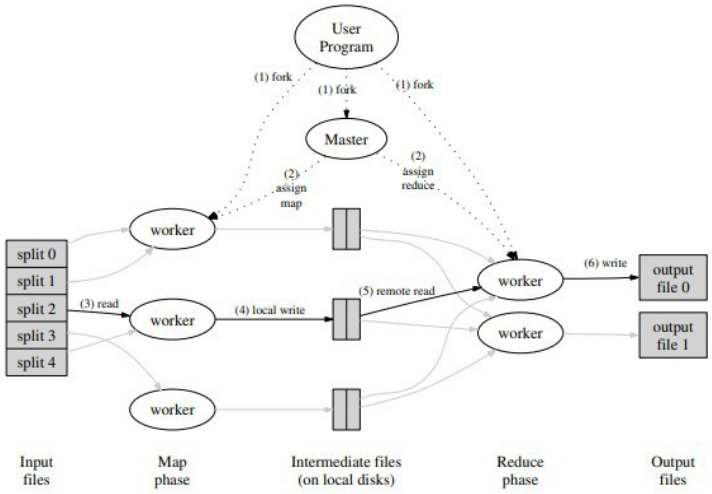


Figura 13. Vista General de ejecución (Dean & Ghemawat, 2004)

A lo largo del texto se utilizan los términos *Trabajo o Job* y *Tarea(s) o Task(s)*. Para poder comprender el proceso que se lleva a cabo cuando se pretende utilizar el framework MapReduce de programación, es indispensable comprender que:

- **Job** hace referencia a la ejecución del Programa MapReduce que se ejecuta en el clúster Hadoop en forma de Batch. Es decir, si el usuario solicita la ejecución de un programa

estructurado en MapReduce, tal ejecución es entendida como *MapReduce - Job* (o simplemente *Job*), sin importar de cuantos bloques o nodos este compuesto el archivo o el clúster, respectivamente. Por lo tanto, sólo hay un *Job* que será ejecutado.

- **Task(s)** hace referencia a distintas subejecuciones que en conjunto forman un *Job*. Por lo que el *Job* está compuesto de todas las *Tasks*, siendo estas básicamente de dos tipos: Tareas Map y Tareas Reduce. Tales tareas serán repartidas a los nodos, y sus respectivos recursos o contenedores, a través del progreso del programa entero. Todas las tareas inicialmente se encuentran en cola esperando ser asignadas a un nodo para ser ejecutadas y el *Job* termina hasta que se consume la última tarea.

Antes de explicar la programación y pasos de MapReduce, es conveniente conocer el proceso que se lleva a cabo en el clúster al solicitar la ejecución de un *Job*. La Figura 14 muestra el proceso general. Dentro de cada bloque en color azul claro, representa una entidad independiente de las demás. Idealmente cada entidad podría representar un nodo físico, aunque esta imagen los ilustra independientes para comprender el proceso. Estas entidades podrían estar alojadas en un solo nodo físico. Por ejemplo, comúnmente se tiene la entidad *Client* y la entidad *ResourceManager* en un solo nodo físico, aunque son procesos independientes dentro del nodo (White, 2015).

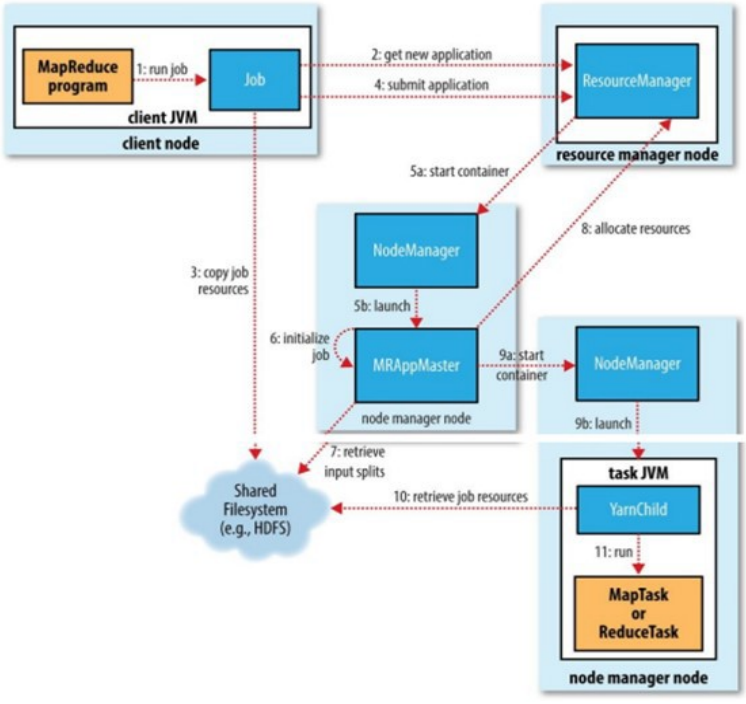


Figura 14. Ejecución de Job en MapReduce (White, 2015)

1. La entidad *Client* Solicita la ejecución de un *Job* a un proceso inicial llamado *JobSubmitter*. Este solicita un *Job-ID* al *ResourceManager* para dar seguimiento al progreso del *Job*. El proceso *JobSubmitter* verifica la información brindada en la solicitud para determinar si esta es

correcta y continuar o si devolverá un mensaje de error al usuario solicitante.

2. Al solicitar un ID, se establece comunicación con las entidades a cargo del sistema de Archivos HDFS, solicitando acceso al archivo en donde se ejecutará tal *Job*.

3. Una vez reservado el archivo a procesar, es decir la ubicación de los bloques que lo componen, se envía el programa al *ResourceManager*.

4. El *ResourceManager* solicita a un *Nodemanager* la creación del contenedor para inicializar en este el proceso *Application Master*, el cual verificará la cantidad de particiones que componen el archivo de entrada y determinar si el *Job* requerirá de más recursos en otros *Nodemanagers* para ejecutar el *Job.*

5. Conociendo la información del tamaño del archivo y la tarea a ejecutar, se informa al *ResourceManager* los recursos a ocupar para llevar a cabo la tarea. Si el *Application Master* determina que se necesitan otros contenedores, en nodos adicionales, solicita su creación, el cual recogerá el bloque que se necesite del sistema HDFS.

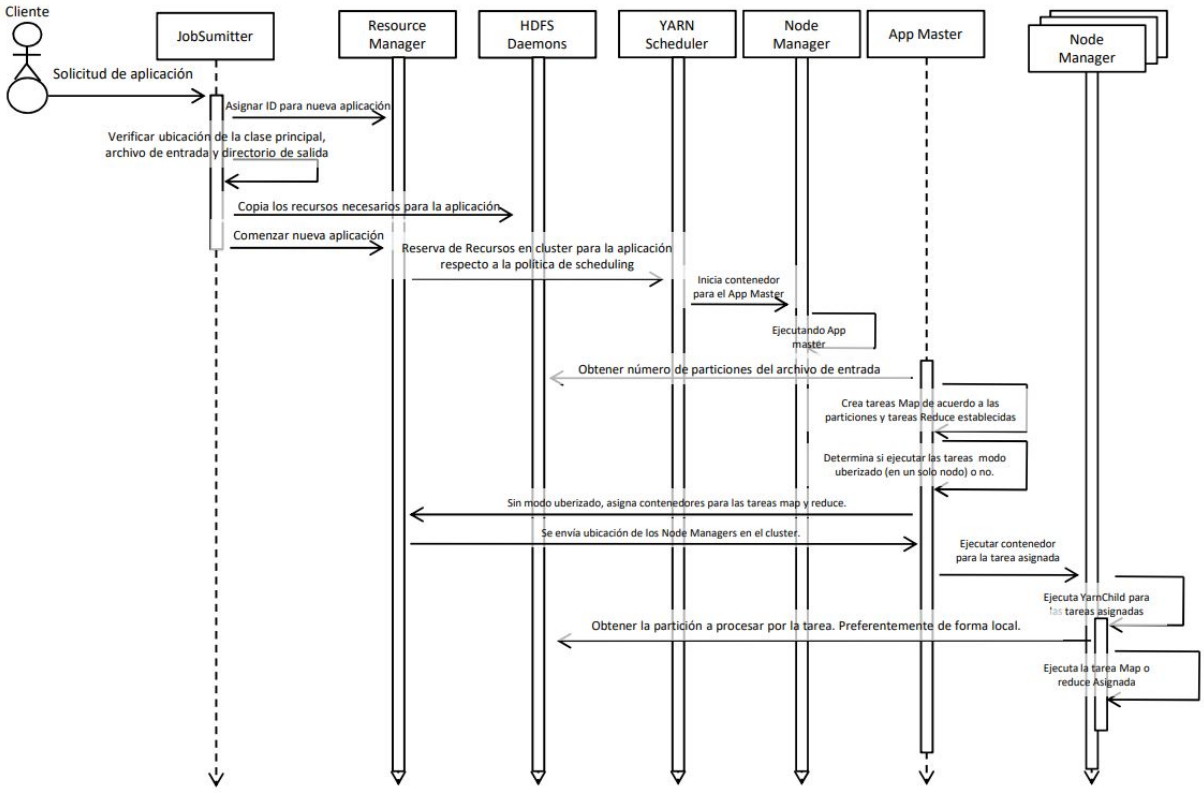


Figura 15. Diagrama Secuencial del lanzamiento de una aplicación (Job) en MapReduce bajo implementación YARN.

2.3.2. Programación en el modelo MapReduce.

Como se describió previamente, una solución para manejar un problema con datos muy grandes es por medio de la técnica *Dividir y Conquistar*. Esto significa, dividir el problema general en subproblemas de menor tamaño, independientes hasta cierto punto. Ya sea en programación paralela o distribuida, el programador debe hacerse cargo de coordinar acceso a los recursos compartidos, manejo de sincronización, como las *barreras* y evitar problemas comunes como *deadlocks* y *condiciones de carrer*a. Incluso con librerías como OpenMP o Message Passing Interface (MPI) a nivel de clúster, el programador debe dar seguimiento a la forma en que los recursos se hacen disponibles a los trabajadores. Con tales desventajas, MapReduce permite ocultar diversos detalles a nivel del sistema para el programador. Al igual que las alternativas de OpenMP y MPI, MapReduce proporciona un medio para distribuir el procesamiento sin sobresaturar al programador con detalles del cómputo distribuido, a cierto nivel de granularidad. En contraste a los tradicionales métodos de operar con archivos de información donde los datos deben ser transferidos a las unidades de cómputo, es el procesamiento o código el que es transferido a las ubicaciones donde se encuentran los datos, ofreciendo una alternativa al costo más importante en el cómputo, la comunicación.

Como el mismo nombre lo indica, este modelo está compuesto, en su más simple contexto, de dos Etapas; Map y Reduce. La estructura básica de datos en este estilo de programación es en pares Clave-Valor (Key-Value pairs). El tipo de datos que puede manejar tanto las claves como los valores pueden ser del tipo: primitivos; enteros, punto flotante, cadena de caracteres y bytes; o de estructura más compleja; listas, tuplas, arreglos, etcétera.

De esta forma, el programador debe definir el código para ambas etapas con las siguientes firmas[†‡]:

$$\text{Map}:(k1,v1)\rightarrow[(k2,v2)]$$

$$\text{Reduce}:(k2,[v2])\rightarrow[(k3,v3)]$$

Se observa que el resultado o salida de la etapa Map corresponde al mismo formato de la entrada en la etapa Reduce.

El código definido como etapa Map, será aplicado a cada clave-valor único en la entrada, (K1,V1). Al procesar todos los pares en la entrada, se generará una cantidad arbitraria de pares clave-valor intermedia (K2,V2). El código definido para la etapa Reduce será aplicado a todos los valores asociados a la misma clave, (K2 , [... V2 …]), para generar pares de salida [(K3,V3)]. En el paso intermedio entre Map y Reduce, se observa una ventaja de este framework, la cual es garantizar el agrupamiento de todos los valores por clave idéntica, y de esta forma actúan como entrada a cada reduce. Al término de la

[†] K significa clave(key), mientras la letra V significa valor (value).

[‡] El contenido dentro de los corchetes [..], se refiere a una lista de elementos; valores, claves ó claves-valores

etapa Reduce, los datos de salida son reescritos en el sistema de archivo distribuido, es decir en HDFS.

En base al modelo de programación, la siguiente imagen nos muestra el panorama general de las etapas que se debe considerar cuando se genera código con este framework.

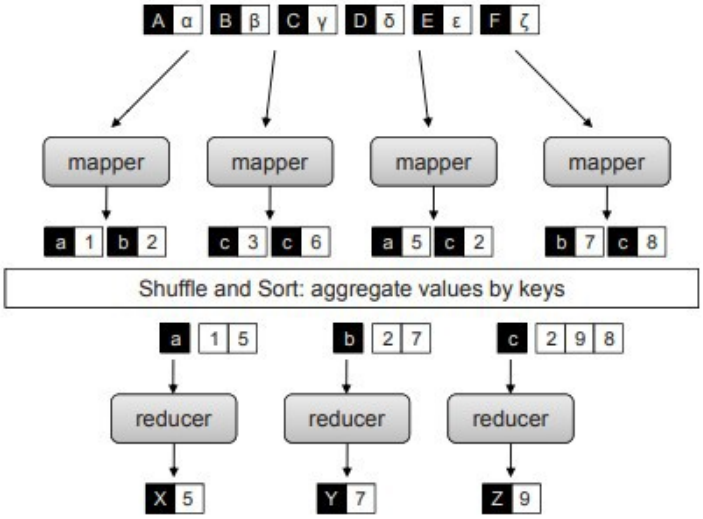


Figura 16. Vista simplificada de MapReduce (Lin & Dye, 2010)

Como ejemplo, se considera el siguiente pseudocódigo para conteo de palabras en un archivo.

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)
1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Figura 17. Pseudocódigo MapReduce para conteo de palabras (Lin & Dye, 2010)

La descripción de la tarea Map, declarándola como una clase junto con el código de su método, recibe dos argumentos que, en orden, corresponde el *primero a la clave*, en este caso un ID que hace referencia a una línea del archivo a procesar, y el *segundo al valor* de este par, que en este sencillo ejemplo representa el contenido, o todas las palabras, de esta línea. Dentro de este método Map se declara un ciclo de iteraciones, el cual indica que para cada palabra que contenga la actual línea se

emita en la salida ***la palabra como clave*** y el ***número 1 como valor***[§], generando así un total de pares clave-valor igual a todas las palabras dentro del archivo.

Para la descripción de la tarea Reduce, como argumentos se recibe, también en orden, primero la clave (la palabra), y segundo, una lista de todos los valores que tienen la misma clave (idéntica palabra). Se inicializa una variable temporal de conteo y se declara un ciclo donde se realizará cada iteración para cada valor dentro de la lista, claramente la lista será un conjunto de número 1, los cuales serán sumados actualizando la variable temporal. Al terminar el ciclo se emite finalmente la palabra o clave que se recibió y el conteo final de esta clave ó palabra. Nuestro sencillo código realiza lo siguiente si la entrada es solo una línea.

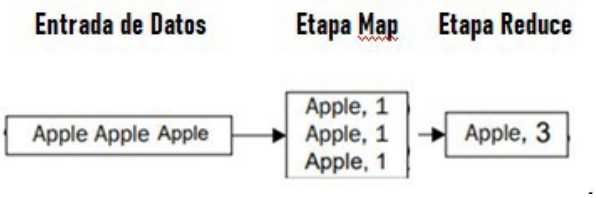


Figura 18. Diagrama del proceso para el pseudocódigo de la Figura 16

Bajo este criterio, es decir, a vista solo del programador, se debe tomar en cuenta la cantidad de ejecuciones Map y Reduce que serán aplicados en el código, a fin de tener un sentido de control al momento de programar. Una simple vista cuando se define la tarea Map, es tomando que el código definido en esta etapa será solo para un par clave-valor.

Por defecto, en Hadoop la clave en el archivo de entrada es considerada como el número de línea (o renglón) en el texto o dataset que se ingresa, mientras que el valor es el texto o contenido en tal línea. Esta es una propiedad en la entrada del archivo llamada ***FileInputFormat***[**]. Si el archivo de entrada cuenta con cuatro líneas, sin importar la cantidad de palabras, letras o caracteres que contenga cada línea, se considera como cuatro pares clave-valor. Por lo tanto el código definido será ejecutado cuatro veces, una por cada par. Esto no significa que existen cuatro tareas Map, simplemente las veces que se ejecutará el código. La cantidad real de tareas Map está definida por la cantidad de bloques en que se dividió el archivo general.

El archivo podría contener miles de líneas y estar dividido en cuatro bloques, lo que significa que el sistema Hadoop determinará cuatro tareas Map, de las cuales, cada una de las tareas ejecutará el código que tenga definido la cantidad de líneas que contenga su respectivo bloque. Si un bloque cuenta con 200 líneas, tal tarea Map ejecutara su código 200 veces, una por línea, o en otras palabras, una por clave-valor.

[§] La descripción del código indica un conteo del número de ocurrencias de palabras idénticas que se encuentren. En este caso solo se asume asignación del número 1 para todas las palabras, idénticas o no, para mayor entendimiento con la figura x.

[**] Existen distintas propiedades para la entrada de datos que pueden ser especificadas dependiendo de la finalidad del programa.

Como resultado de esta primera etapa, se generará una cantidad específica de claves y cada clave contendrá una cantidad de valores. Por lo que, desde la perspectiva del programador, el código definido en la etapa Reduce, se ejecutará la cantidad de claves que reciba del paso intermedio. Si como resultado de la etapa Map se generaron 300 claves auténticas (podrían generarse miles de claves pero al agrupar por su igualdad se garantiza su autenticidad), con sus valores agrupados, entonces el código definido en la etapa Reduce se ejecutará 300 veces, una por cada clave.

Retomando el código de la Figura 16, apliquemos sobre un archivo con cuatro líneas de texto y dividiendo el archivo en dos partes. El siguiente diagrama nos muestra el proceso.

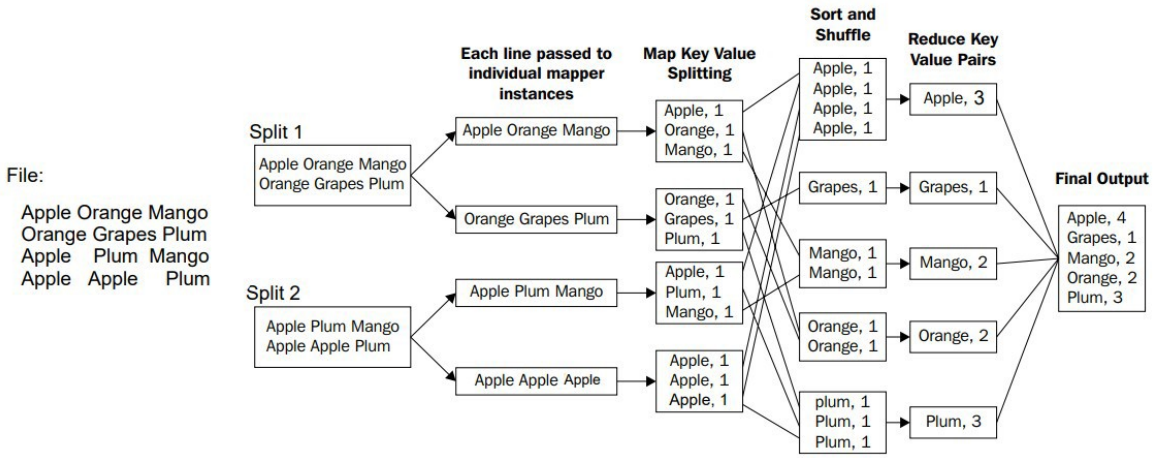


Figura 19. Diagrama para la ejecución del pseudocódigo Figura 16 (Achari, 2015)

En este diagrama, la etapa *Sort* and *Shuffle* significa la garantía del framework para agrupar los valores por claves idénticas, por lo que el programador no tiene que preocuparse por este ordenamiento.

2.3.3. Data Locality.

Este framework distribuido se enfoca en ubicar los datos en los nodos de cómputo de forma distribuida con el propósito de realizar el procesamiento de forma local. En contraste a las tradicionales formas de operar información como las bases de datos relacional en donde la transferencia de esta información se convierte en un cuello de botella cuando los tamaños son de volúmenes muy grandes. A este método se le llama *Localidad de Datos* o comúnmente *Data Locality*, la cual es la característica principal en Hadoop. De esta forma, lo que se envía a cada nodo es el código que procesará el bloque que tal nodo contenga.

Físicamente, en la primera etapa Map, se observa claramente la localidad de datos en forma de bloques, en donde el framework envía las tareas Map, una por bloque/nodo. Es en la siguiente etapa, Reduce, que se desvanece esta localidad de datos al ordenar y enviar los datos intermedios al nodo que realizará la operación reduce. Si la cantidad de claves es muy grande, la transferencia de

datos crece y con esto, el desbalance en el procesamiento, incrementando el tiempo total de la ejecución del *Job*, debido al costo en la comunicación.

Con la intención de balancear este peso en la comunicación, existen diversas soluciones para implementar tanto en el framework del sistema como en la programación del código en MapReduce.

Para el caso de mejora desde la perspectiva del framework, una propuesta sugerida en el trabajo (Li, Wu, Zhong, & Yang, 2015), en donde el objetivo es asignar las tareas Reduce a los nodos que contengan la mayor cantidad de claves que tal tarea considere procesar. Esto es, a partir de cierta cantidad de claves generadas en la tarea Map localmente en un nodo, se puede designar una tarea Reduce que procese los valores para tal clave, evitando de esta forma transferir esta cantidad de claves hacia otro nodos que probablemente cuente con pocas o ninguna clave para la designada tarea Reduce. Por supuesto esta propuesta considera la carga que cada nodo tenga asignada y delegar a otro nodo el procesamiento si no considera viable asumir la carga de trabajo.

Considerando mejorar desde la perspectiva de la programación, existen etapas anexas a Map y Reduce; Etapa de *Combinadores y Particionadores*.

Tomemos como ejemplo la imagen de la Figura 19 que trata con el conteo de palabras en un archivo. En este caso cada palabra en el bloque correspondiente, es asignada como clave en los datos intermedios, y el número 1 como el valor para cada clave. Claramente se crearán multitud de claves iguales con el valor 1. El framework Hadoop garantiza el agrupamiento por clave, por supuesto con sus valores correspondientes, y la transferencia a una tarea Reduce. El inconveniente yace en la enorme cantidad de claves iguales a ser transferida hacia la siguiente etapa. En este ejemplo, debido a que es un conteo de ocurrencias, desde la etapa Map es posible realizar un conteo local de coincidencias con el objetivo de transferir una menor cantidad claves. Esta implementación llamada *Agregación Local* (*Local Aggregation*), permite una menor cantidad datos intermedios para la siguiente etapa.

Para visualizar esta A*gregación Local*, tómese el ejemplo de la Figura 19 pero esta vez el código en la etapa Map considera un previo agrupamiento para las palabras que sean iguales. La Figura 20[††] muestra este agrupamiento local desde la etapa Map. Se puede observar que la cantidad de comunicación se reduce a un solo enlace a la salida de la etapa Map e inicio de la etapa *sort and shuffle*. A diferencia de la figura anterior, Figura 19, cada clave tiene una línea de comunicación por cada clave (representada con líneas conectoras), sin importar que sean idénticas y por supuesto que provengan de la misma ejecución o instancia.

[††] En esta figura, para ilustrar el efecto de la agregación local, se modificó el contenido de cada línea.

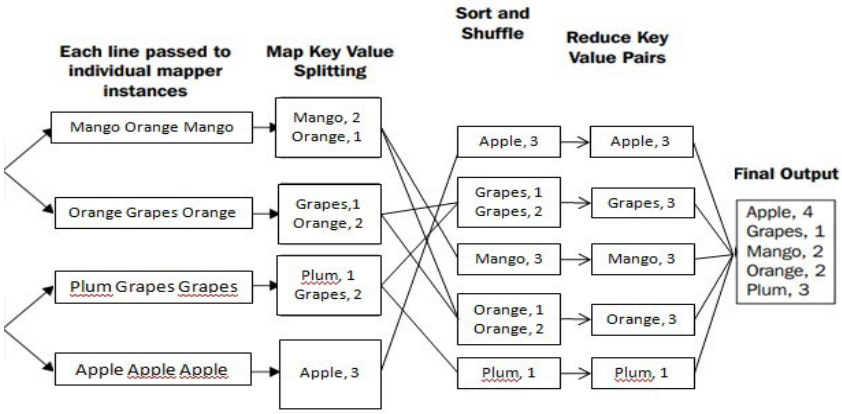


Figura 20. Diagrama para la ejecución del pseudocódigo de la Figura 16 agregando una agrupación en la etapa Map

Desafortunadamente, no todos los propósitos son candidatos de utilizar este tipo de *Agregación Local*. Tómese por ejemplo los casos en que no se requiere obtener el total de ocurrencias sino un promedio de alguna cantidad. Es por lo tanto que operaciones que presenten asociatividad y conmutatividad en el proceso general, son buenos candidatos para la *Agregación Local*.

Capítulo 3

Implementaciones

Para el caso de este trabajo se realizaron implementaciones con el propósito de experimentar la operación del sistema Hadoop sobre archivos de tamaño que permitan observar su división en diferentes bloques y de forma automática la creación de más de una tarea Map. La primera implementación muestra el algoritmo utilizado para el procedimiento de Max-Pooling con deslizamiento de todas las ventanas en cada elemento. Consecuentemente, se toma ventaja en la configuración del programa controlador para el ingreso de más de un archivo al programa completo MapReduce con el objetivo de implementar un Kernel que filtre una matriz de entrada. En estas dos implementaciones se realizaron dos algoritmos que generan la misma cantidad de claves para la etapa Reduce. De esta forma, se puede realizar una comparación en el tiempo de ejecución de los dos algoritmos, claro, además de considerar el tamaño de la matriz.

3.1. Tiempo de ejecución

Previo a mostrar las implementaciones, es importante recordar el objetivo global de este framework, esto es, dividir la entrada de datos y procesar cada partición a fin de reducir el tiempo de ejecución que tomaría procesarla secuencialmente. En otras palabras, procesar cada bloque de datos de forma paralela. En (Pacheco, 2011) se nos recuerda la importancia del no determinismo en el cómputo al implementar el paralelismo para un objetivo en particular.

Para estas implementaciones, en donde se define un código para cada etapa, es evidente que hay no determinismo en el tiempo de ejecución debido a la independencia que a cada tarea le tome concluir, más aún, la independencia de cada nodo[‡‡]. Por lo tanto, el tiempo total que le tome a un trabajo en este ambiente, Hadoop, depende también de aspectos independientes al algoritmo utilizado, como; la configuración que determina la cantidad de bloques que componen el dataset inicial, la repartición de tareas Map o Reduce y el tiempo de comunicación entre nodos, esto es, entre tareas Map a Reduce.

En este apartado se analizará el tiempo de ejecución de las implementaciones, esto es, en el algoritmo pensado para el framework de MapReduce. El final del capítulo previo, señala que existe

[‡‡] En este sentido, el framework está pensado para operar sobre nodos no dedicados para un propósito en especial, sino de acceso común, por lo que es cambiante el tiempo de ejecución debido a los recursos con que cada nodo cuente

una relación en la forma de abordar el algoritmo con el tiempo de comunicación, en específico en la cantidad de claves generadas antes de la etapa Reduce; menor cantidad de claves (auténticas o no), menor cantidad de comunicación. Ambos algoritmos utilizados para la implementación de Max-Pooling y Filtrado por medio de un kernel, generan la misma cantidad de claves. Por lo tanto, aún se mantiene independiente el tiempo de comunicación en la etapa intermedia.

3.2. Implementaciones Max Pooling y Aplicación de Kernel.

Uno de los sub campos dentro del aprendizaje de máquina son las Redes Neuronales Convolucionales, las cuales realizan extracción de características automáticamente de los datos pre procesados para su operación (wang, cui, & ke, 2023). Comúnmente estas redes neuronales convolucionales consisten de distintas capas como; convolucional, de extracción, activación entre otras. La convolución es la capa principal en las redes convolucionales ya que es aquí donde se detectan características en los datos de entrada, no únicamente en imágenes, sino en datos de audio, texto, series de tiempo etcétera. Tales características son obtenidas aplicando filtros o *kernels* en la entrada de datos a fin de crear mapas de características. Seguido de la capa de convolución se aplica una capa de extracción. Como se explica en (Özdemir, 2023) esta capa de extracción (*pooling layer*) es una etapa crucial para reducir el tamaño del modelo de la red convolucional. Esto es, reemplazando un conjunto de valores en una cantidad reducida de estos que reemplaza las características comunes de la entrada en información "valiosa", manteniendo la información útil y descartando información irrelevante. Los propósitos principales de aplicar una capa de extracción o pooling son; reducir el costo computacional al reducir el número de parámetros y, acorde a (Gholamalinejad & Khosravi, 2020), prevenir al modelo de memorizar.

Existe una similitud en la aplicación de un kernel que filtre los datos de entrada (o convolución) y la extracción de características. Para ambas técnicas se utiliza una ventana con determinado tamaño sobrepuesta en la matriz de entrada, la cual se deslizará a través de la matriz completa. Con el rango de valores que abarque o limite esta ventana sobre la matriz de entrada se realiza alguna operación. La operación que se realice es lo que distingue ambas técnicas. En general los deslizamientos son los que producen la distinción entre ventanas. Tómese como ejemplo la figura siguiente para comprender la aplicación de este concepto. En este caso solo se muestran tres ventanas que corresponden (aplicando alguna determinada operación) a los elementos de una matriz de salida.

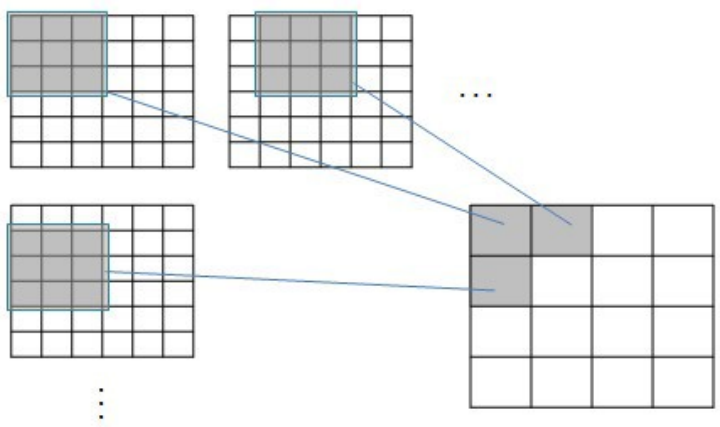


Figura 21. Representación de la Matriz de salida al aplicar ventanas en la Matriz de Entrada.

Se muestra que la matriz de entrada es de 6x6, el tamaño de la ventana de 3x3 y la separación entre ventanas (el deslizamiento) es de uno.

Debido a que cada ventana corresponde a cierto elemento en la matriz de salida, se tiene que la cantidad de ventanas es el total de elementos en esta matriz. Por esta razón, podemos asignar índices que identifiquen a cada ventana. Tales índices serán los mismos para los respectivos elementos en la matriz de salida.

Con esta información de antemano, los dos algoritmos que tratan con estas dos implementaciones, Max-Pooling y convolución mediante un kernel, generan una cantidad de claves igual a la cantidad de ventanas. Por lo tanto, es posible asignar a cada clave los índices de las ventanas y por supuesto esto corresponde a los índices de la matriz de salida. Los valores de cada clave, será un acarreamiento de información de la matriz de entrada que cubra su respectiva ventana. Tal información es lo que distingue una implementación de la otra.

3.2.1. Max-Pooling

La técnica de *pooling* tiene como objetivo preservar el significado de los valores de entrada en un tamaño reducido, esto es, expresar un grupo de valores en la entrada con un solo valor que represente un determinado significado dentro del grupo. Este *submuestreo* de la entrada tiene como propósito reducir la complejidad computacional y el número de parámetros. Las técnicas comunes en el submuestreo son *Max-Pooling*, o extracción del valor máximo del grupo, y *Average-Pooling*, extracción mediante el promedio en los valores del grupo.

Se conoce que el formato de la matriz de entrada es en forma de un valor (con índices) por renglón. Esto es, índice de renglón, el índice de la columna y el valor del elemento, por supuesto colocando una coma para separar estos tres identificadores.

Previamente se explicó que el formato por defecto del clúster en Hadoop realizará la ejecución del código Map por renglón. Una ejecución será entonces considerando solo un elemento.

3.2.1.1. Primer Algoritmo

El primer algoritmo para la etapa Map generará las claves con los índices de las ventanas aplicando estas sobre toda la matriz de entrada por medio de deslizamientos (strides) parametrizables desde el código de la clase principal o de control.

El algoritmo opera de la siguiente forma: se determinan los parámetros de la matriz de entrada, tamaño de la ventana, y número de deslizamientos entre ventanas. Estos parámetros son preestablecidos al definir el código del controlador, los cuales pueden ser accedidos por los códigos Map y Reduce.

En la etapa Map, por cada elemento de la matriz de entrada, realizará *n* iteraciones, siendo *n* el total de ventanas. Si el valor siendo procesado está dentro de los límites superior e inferior de la ventana actual en iteración, se le asigna como clave los índices de tal ventana. En caso contrario, continúa con la siguiente ventana o iteración. En la Figura 22 se observan los valores de la matriz de entrada en donde se ejecuta la tarea Map. El contorno en color rojo muestra los límites de alguna ventana, los cuadros sombreados son los elementos de la matriz que ya fueron procesados previamente y el cuadro enmarcado con mayor intensidad es el elemento que está siendo procesado al momento actual.

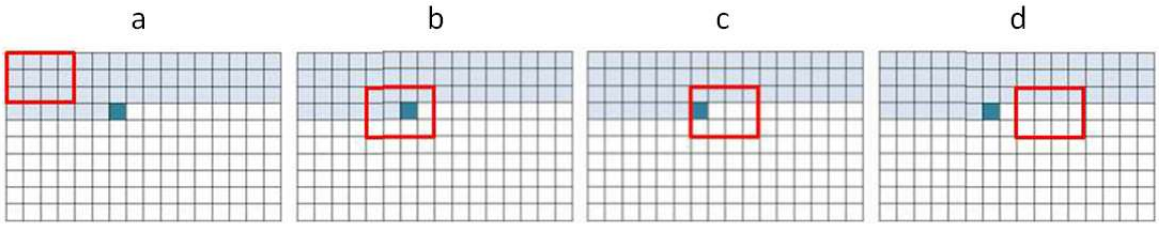


Figura 22. Representación del deslizamiento en el primer algoritmo.

Se nos muestra un solo procesamiento o instancia que corresponde al elemento *i,j* de la matriz de entrada y cuatro distintas iteraciones. Con un *stride* de dos la imagen *a* nos muestra la primera iteración, las imágenes *b, c y d* nos muestra la iteración 11°, 12° y 13°, respectivamente. La Figura 22 revela que este elemento se asignará solamente a las ventanas 11(con índices 3,2) y 12(índices 4,2) como clave al ejecutar el código Map en este elemento. Al término de las *n* iteraciones comienza un nuevo procesamiento con el elemento adyacente a la derecha, es decir, la línea de texto que contiene *j, i+1,x* siendo *x* el valor que contenga tal elemento.

Enviando estas ventanas a la siguiente etapa, Reduce, se obtiene que la cantidad de claves es la cantidad de ventanas que tienen lugar dentro de la matriz de entrada. Por lo que las claves asignadas son de la forma *(m, n)*, es decir los índices de la ventana. Como valores, estos se componen de la lista de los elementos dentro del rango de cada ventana. Debido a que la operación Max-Pooling solo requiere los valores dentro de la ventana para calcular el máximo, no se requiere enviar los

índices de estos.

Clave Valores

[m , n] [x_1 , x_2 ,…]

La etapa Reduce recibirá esta lista de valores, donde en el procesamiento, es decir el código definido para esta etapa, se considera como aplicado sólo sobre los valores de cada clave única. Para este caso, en el código Reduce se realiza un bucle iterando con cada elemento de la lista y actualizando una variable temporal con el elemento actual en iteración solo si el valor del elemento es mayor que la variable temporal. De esta forma, al final de este bucle se mantiene solo el elemento más grande de la lista. El framework por defecto en Hadoop, separa con una tabulación la clave del o los valores. Con el objetivo de ignorar la tabulación y asignar un símbolo de coma "," entre la clave y el valor, en la salida se asigna la clave como "*null*" y como valor el primer elemento en tipo texto será la clave de entrada, es decir los índices de la ventana, se agrega un símbolo de coma *","* y se coloca el contenido de la variable temporal con el elemento máximo encontrado. De esta forma, en la salida global de todas las instancias Reduce aplicadas, existe una clave única, *"null"* y una lista de valores en donde cada uno contiene como texto: índices de la ventana, un símbolo de coma *","* y el valor máximo encontrado en esta ventana. Asumiendo *k3,v3* en la salida del *Job* se tiene:

$$K3 = \text{null},$$

$$V3 = K2 + \text{coma}(,) + \max[V2]$$

El pseudocódigo utilizado para este caso es de la siguiente manera:

```
class Mapper
  method Map (Key line-offset; Value index)
    for each (window : windows)
      if (element window-span)
        Emit (Key Window-index, Value element)
    Next Window
class Reducer
  method Reduce (Key Window-index; Value List[elements])
    maxelement = max [elements]
  Emit (Key null; Value Window-Index & maxelement)
```

Figura 23. Pseudocódigo del primer algoritmo.

Debido a la naturaleza del framework MapReduce, este código es buen candidato para ser analizado como se recomienda por (Wilkinson & Allen, 2004), es decir de forma secuencial. La complejidad temporal que interesa para el propósito de comparar ambos algoritmos utilizados es mediante la notación $\boldsymbol{O}$. Como guía se utiliza la definición de tiempo de ejecución de (Cormen, Leiserson, Rivest, & Stein, 2009) que indica es el número de operaciones primitivas o pasos ejecutados. Desde esta

perspectiva, una cantidad constante de tiempo es necesaria para cada línea del pseudocódigo. Por lo tanto, la ejecución de la i-ésima línea toma una cantidad constante c_i de tiempo.

Analizando el pseudocódigo en la etapa Map con mayor detalle:

	Costo	Repeticiones
class Mapper		
method Map (Key line-offset; Value index)		
Get M_r M_c K_r K_c , stride **from** DriverClas	C1	1
Set totalWindows = $[(M_r - K_r)/stride]*[(M_c - K_c)/stride$	C2	1
for each (window : totalwindows)	C3	n
update window-span	C4	n-1
update window-indexes	C5	n-1
if (element ∈ window-span)	C6	n-1
Emit (Key Window-index, Value element)	C7	$\sum_{j=0}^{n-1} t_j$
Next Window		

Figura 24. Pseudocódigo Map del Primer Algoritmo con el costo de cada instrucción. *Mr y Mc* indican el número de renglones y columnas respectivamente, de la matriz de entrada. *Kr y Kc* indican renglones y columnas del Kernel.

Para este caso, el valor de n es igual al número total de ventanas que tengan lugar dentro de la matriz de entrada. El valor de t_j en la última instrucción, nos indica el número de veces que esta línea de código será ejecutada para la iteración o ventana j. Debido a que se trata de una condicionante tipo *if*, el valor de t_j puede ser 1 ó 0, dependiendo si el elemento esta dentro del rango de la ventana en iteración.

Para calcular el tiempo de ejecución de este pseudocódigo se realiza la suma de cada instrucción, multiplicando cada instrucción (o su costo) por su respectivo número de iteraciones, esto es

$$T(n) = c_1 + c_2 + c_3 n + c_4(n-1) + c_5(n-1) + c_6(n-1) + c_7 \sum_{j=0}^{n-1} t_j$$

Para el último elemento de esta ecuación, t_j, se sabe que será cero si el elemento no forma parte de la ventana en iteración, y uno si esta dentro del rango, por lo que en el mejor escenario, es decir la menor cantidad de veces que se ejecute la instrucción, se da cuando el elemento solo pertenece a una ventana, esto es:

$$c_7 \sum_{j=0}^{n-1} t_j = c_7$$

por lo que en este caso la ecuación resulta:

$$T(n) = C_1 + C_2 + C_3 n + C_4(n-1) + C_5(n-1) + C_6(n-1) + C_7$$

Reordenando la ecuación, recordando n como la variable independiente:

$$T(n) = (C_3 + C_4 + C_5 + C_6)n + (C_1 + C_2 + C_4 + C_5 + C_6 + C_7)$$

En la peor situación en donde t_j corresponda a la mayor cantidad de ejecuciones, se considera que el elemento pertenece a la mayor cantidad de ventanas posibles ó a todas.

$$T(n) = (C_3 + C_4 + C_5 + C_6 + C_7)n + (C_1 + C_2 + C_4 + C_5 + C_6) \quad \ldots\ldots \quad \text{(ec. 1)}$$

En ambos casos se observa una función del tipo *an + b,* es decir una función lineal. Traduciéndose que el tiempo de ejecución crece linealmente conforme crece la cantidad de ventanas.

Analizando el tiempo de ejecución de la etapa Reduce

	Costo	Repeticiones
class Reducer		
method Reduce (Key Window-Index; Value List[elements])		
asignment-operations	C1	1
maxelement = max[elements]	C2	1
Emit(Key *null*; Value *Window-Index & maxelement*)	C3	1

Figura 25. Pseudocódigo Reduce con el costo de cada instrucción

$$T(n) = C_1 + C_2 + C_3$$

3.2.1.2. Segundo Algoritmo

Para el caso de una alternativa al código anterior, el enfoque de asignar las ventanas es diferente. En el código anterior se realizan *n* iteraciones donde *n* es el total de ventanas con la adición de algunas instrucciones cuando el elemento siendo procesado pertenece a la ventana. Este segundo enfoque tiene como propósito evitar la iteración de todas las ventanas, y solamente iterar para las ventanas a las que el elemento pertenezca.

Como se podrá observar, la diferencia radica en la disminución de iteraciones ya que si el número de ventanas tienden a crecer, las iteraciones aumentan. En este segundo algoritmo, si las ventanas incrementan, su tamaño se disminuye y por lo tanto las iteraciones disminuyen. El siguiente pseudocódigo, mediante comentarios, explica cada instrucción la forma de abordar el problema:

```
class Mapper
  method Map (Key line-offset; Value [indexes,element])
  Get inputsize, poolsize, stride from DriverClas
  outputsize = (inputsize – poolsize) / stride + 1 //# Determina el tamaño de la matriz de salida

  if (⌊outputsize⌋ < outputsize) //# Si el total de ventanas no cubre toda la matriz de entrada
  outputsize = ⌊outputsize++⌋ //# aumenta el número de ventanas

  nm = ⌊ elementIndex / stride ⌋//# establece nm como la última ventana al que pertenece el elemento actual
  if (nm >= outputsize) //# si esta ventana sobrepasa los límites de la matriz de salida
    nm = outputsize - 1 //# asigna como ultima ventana el ultimo indice de la matriz de salida

  while ( nm-span > elementIndexes && n >= 0) //#mientras el elemento se encuentre dentro del rango de la ventana actual y esta no sea cero
    Emit (Key [nm]; Value element) //# agregar el elemento actual a esta ventana
    nm-- //# continua con la ventana anterior
```

Figura 26. Pseudocódigo Map del Segundo Algoritmo.

Se observó que se puede obtener de forma directa la última ventana a la que será asignado cada elemento en la línea de asignación de *nm*, asumiendo el recorrido de esta ventana de izquierda a derecha, mediante la función piso en el cociente del índice del elemento entre el desplazamiento (es decir la parte entera de tal cociente). Podemos observar que en la primera condicionante toma en cuenta aquellos elementos que hayan quedado fuera o no considerados al calcular las ventanas con la ecuación para obtener el tamaño de la matriz de salida. Debido a que en los últimos elementos de la matriz, ya sea flanco derecho o inferior, la ventana calculada mediante la función piso estará fuera del rango de la matriz de salida, por lo que, si es el caso, a tal elemento se le asigna el último índice (de columna o de renglón) de la matriz de salida. De esta forma, el ciclo *while* en la siguiente instrucción comenzará con tal ventana como la última. Cada iteración de este ciclo, asignará como clave la ventana actual, comenzando con la última, y recorriéndola en forma reversible hasta que la ventana actual o decrementada no abarque el índice del elemento siendo procesado. Otra condición para que continúe la iteración es que la ventana decrementada llegue hasta cero.

3.2.1.2.1. Análisis de tiempo de ejecución.

Nuevamente se realiza un análisis en el tiempo de ejecución ahora para el segundo algoritmo, como muestra la siguiente imagen donde se han retirado los comentarios para comodidad del análisis y se asignan los costos y las correspondientes repeticiones.

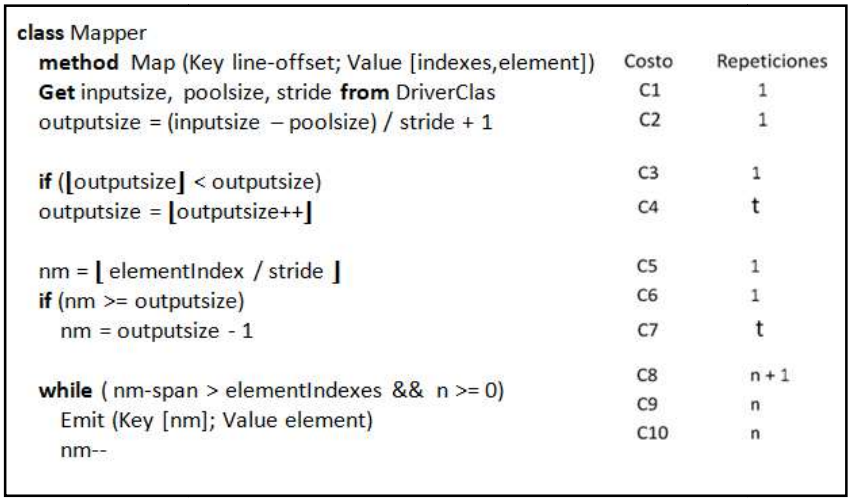


Figura 27. Pseudocódigo Map del Segundo Algoritmo con el costo de cada instrucción.

En este análisis, *n* indica la cantidad de ventanas a las que pertenece el elemento siendo procesado, contrastando al primer código donde *n* indica el total de ventanas. En el caso de *t* nos permite asignar *cero* en la situación que no se cumpla la condición o *uno* en caso contrario.

Se realiza la suma de cada instrucción con el producto de su correspondiente iteración para generar la ecuación respecto a *n*

$$(n) = c1 + c2 + c3 + c4(t) + c5 + c6 + c7(t) + c8(n + 1) + c9n + c10n$$

Reordenando la ecuación resulta

$$(n) = (c8 + c9 + c10)n + (c1 + c2 + c3 + c4 + c5 + c6 + c7 + c8) \quad \ldots\ldots \quad \text{(ec. 2)}$$

Tanto para el mejor como el peor escenario, el tipo sigue un patrón lineal $an + b$ solo removiendo las instrucciones c4 y c7 como constantes.

Para realizar una comparativa entre el primer y segundo algoritmo, debemos considerar que *n* es la variable independiente en ambos pero enfocados de distinta forma. En el primer algoritmo, *n* representa la cantidad total de ventanas y en el segundo las ventanas a las que un elemento representa. En general representa las iteraciones que un algoritmo deba ejecutar por lo que se analiza de qué depende este número en cada caso.

Para el algoritmo 1 se tiene que *n es*:

$$\#VentHor * \#VentVert = ColMatrizSalida * RengMatrizSal$$

Utilizando la ecuación para determinar las columnas y renglones a partir de la matriz de entrada y el tamaño de la ventana:

$$\#VentHor * \#VentVert = \left(\left(\frac{ColmMatEnt - TamVent}{Deslizamiento}\right) + 1\right) * \left(\left(\frac{RengMatEnt - TamVent}{Deslizamiento}\right) + 1\right)$$

A partir de esta valoración, se observa que el total de iteraciones/ventanas, depende del tamaño de la matriz, del deslizamiento y el tamaño de la ventana. Recordando que el tamaño de la ventana está limitado al tamaño de la matriz del lado creciente y hacia uno como valor mínimo. Para las redes neuronales convolucionales (CNN's) se recomienda un tamaño establecido del tamaño de la ventana, cuadrada, de 3, 5, 7 y 9. Por lo tanto, de la ecuación anterior, el tamaño de ventana se considera constante y para el peor escenario, el deslizamiento se considera de 1, dejando la ecuación dependiente del tamaño de la matriz. Por lo tanto a un mayor tamaño de la matriz el número de iteraciones tiende a crecer. Una forma de visualizar el crecimiento en el mejor escenario es con la matriz cuadrada en donde la ecuación anterior sigue la forma $a^2 - 2ab + 1$, de donde el primer término es el de mayor relevancia y el cual nos indica que las iteraciones, y por lo tanto el tiempo de ejecución, crece exponencialmente conforme la matriz de entrada sea más grande.

Sea *ColmMatEnt* = *RengMatEnt* = *TamMat*:

$$Total\ de\ Ventanas\ (n) = \left[\left(\frac{TamMat - TamVent}{Deslizamiento}\right)^2 + 2\left(\frac{TamMat - TamVent}{Deslizamiento}\right) + 1\right] \quad \ldots \text{ (ec. 3)}$$

Con el propósito de depender solo de un término, se establece la matriz de entrada cuadrada para visualizar el efecto en el tiempo de ejecución con una sola variable independiente; *TamMat.*

Considerando el primer término el de mayor relevancia, el total de ejecuciones *n* podría ser reemplazado con tal término cuadrático en el tiempo de ejecución:

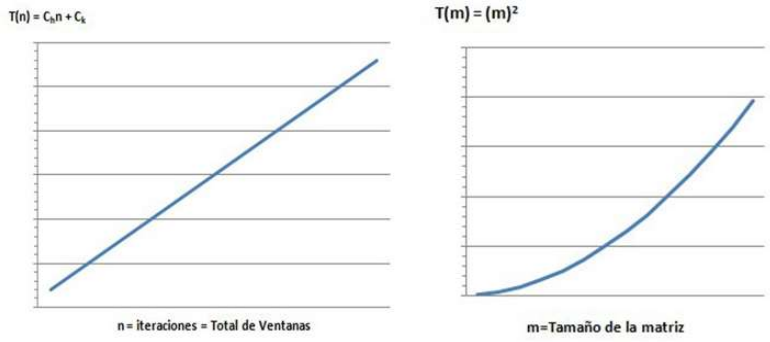


Figura 28. Estimación del tiempo de ejecución para el primer algoritmo. El gráfico Izquierdo indica el tiempo de ejecución respecto al total de ventanas aplicadas en la entrada. El Gráfico Derecho indica tiempo de ejecución respecto al tamaño de la Matriz.[§§]

[§§] La ecuación del gráfico izquierdo hace referencia a *ec.1* . Para la ecuación del gráfico derecho, hace referencia a la *ec.1* al ingresar la *ec. 3* y únicamente tomando el término que impacta en mayor cantidad, esto es, el término cuadrático. Retomando que *n = total de ventanas* y el total de ventanas viene indicado por (ec.3). Tomar en cuenta que el análisis se realiza sobre el código, por lo tanto el tiempo de ejecución es para una sola ejecución.

En el segundo algoritmo, el número de iteraciones *n* depende de las ventanas a las que un elemento pueda pertenecer. El peor escenario donde el número de iteraciones es mayor, es el caso en que alguno de los elementos tienda a pertenecer a todas las ventanas. Si la ventana, ya sea considerando solo renglones ó columnas, es la mitad de la matriz de entrada, los elementos tienden a pertenecer a todas las ventanas. Si la ventana sigue creciendo, el/los elemento(s) tiende(n) a pertenecer a todas las ventanas pero el número de ventanas tiende a reducirse y por lo tanto decrece el número de iteraciones ya que decrece el número de ventanas.

$$\max_{\boldsymbol{TamVent}\ mod\ deslizamiento\ \neq 0} n = \left\lfloor \frac{TamVentana}{deslizamiento} \right\rfloor + 1$$

Para deslizamientos de 1:

$$n = TamVentana + 1 \qquad \ldots \quad \text{(ec. 4)}$$

Nuevamente recordando que *n* representa el número de iteraciones que interesa para el tiempo de una sola ejecución, se muestra la siguiente gráfica del lado izquierdo. Para la gráfica derecha se observa que al incrementar el tamaño de la matriz la ejecución permanece constante ya que no hay dependencia en las iteraciones con el tamaño de la matriz en este segundo algoritmo.

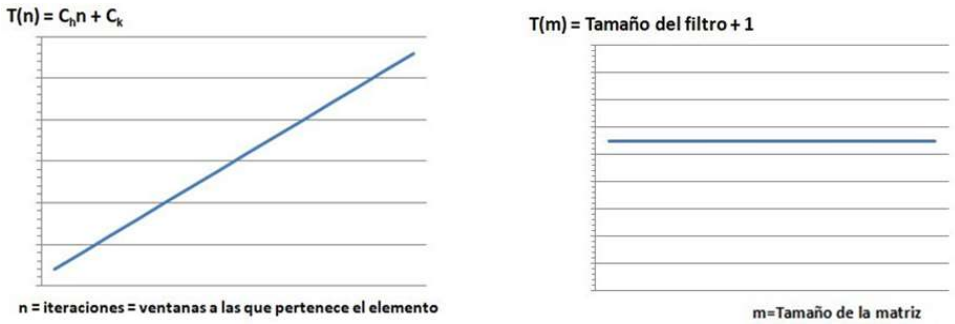


Figura 29. Estimación del tiempo de ejecución para el segundo algoritmo. El gráfico izquierdo indica el tiempo de ejecución respecto al total de ventanas a las que pertenece el elemento. El gráfico derecho indica tiempo de ejecución respecto al tamaño de la matriz [***]

Este análisis es de utilidad ya que, considerando que MapReduce no tenga ninguna división de tareas Map, el primer algoritmo tiende a crecer exponencialmente con el tamaño de la matriz, mientras el segundo crece linealmente con el tamaño de la ventana, por lo tanto es constante cuando la ventana se mantiene, independientemente si la matriz aumenta. Si la matriz aumenta, la ejecución de todo el procesamiento de la matriz, en el segundo algoritmo, crece linealmente conforme esta crece.

[***] Nuevamente considerar el tiempo únicamente para una sola ejecución.

De esta forma, al paralelizar la ejecución de tareas Map, el primer algoritmo sigue creciendo exponencialmente en la ejecución para cada tarea, mientras en el segundo se tiene un comportamiento lineal que tiende a ser constante en el tiempo de ejecución independientemente del tamaño de la matriz de entrada. Por supuesto tal linealidad se inclina más con una mayor cantidad de particiones.

3.2.2. Filtro-Kernel

En este caso, se cuenta de dos matrices: la matriz de entrada y una matriz filtro o kernel. La primer alternativa para poder utilizar una segunda entrada al procesamiento es establecer el código para el segundo archivo, es decir, definir un segundo código Map. La segunda alternativa para poder ingresar de forma independiente el filtro es mediante la distribución del archivo a cada tarea.

3.2.2.1. Ingreso del kernel mediante una segunda tarea Map.

Por medio de esta alternativa, al solicitar la ejecución del programa se especifica dentro de la clase *Driver* la utilización de la propiedad *MultipleInputFormat*. Con esto, al solicitar la ejecución del programa, se especifica dentro de la clase principal que reconozca como; primer argumento la matriz de entrada, segundo argumento el kernel y tercer argumento el directorio de salida. Con esta propiedad es posible, como primer alternativa, establecer un código Map independiente para cada archivo de entrada.

Iniciando con el código del kernel por ser el más simple, en la etapa Map, se generan claves con los índices de todas las ventanas, es decir, de todos los elementos de la matriz resultante. Los valores de cada clave serán todos los elementos del kernel con sus índices de renglón y columna.

	Costo	Repeticiones
class MapperKernel		
method Map (Key *line-Offset*; Value *index&element*)		
for all (window : windows)	C1	n
Emit(Key *Window-index*, Value *index&element*)	C2	n
next Window		

Figura 30. Código Map utilizado como alternativa sin cache distribuido para el segundo archivo de entrada, es decir, el filtro.

Se observa que el código es muy similar al primer código Map de Max-Pooling exceptuando que no hay condicionante y como valor también se agregan los índices del elemento.

$$T(n) = C1n + C2n = n\,(C1 + C2)$$

Se observa que la ecuación es lineal, dependiente del número de ventanas (al igual que el primer algoritmo) con la diferencia que sólo se ejecuta para una partición ya que el kernel es de tamaño considerablemente menor que la matriz de entrada.

3.2.2.2. Ingreso del Kernel enviando el filtro a cada tarea.

La primer alternativa resulta en un procesamiento muy costoso ya que se debe ejecutar por completo una tarea Map en una instancia además de realizar la etapa intermedia, *Sort & Shuffle* para ajustar las claves generadas desde la primer tarea Map dedicada para la matriz de entrada principal. Para evitar este costo de procesamiento, el framework de Hadoop permite, mediante la propiedad *DistributedCache*, compartir con cada tarea que lo solicite el segundo archivo. En este caso, se especifica el envío del segundo archivo a todas las tareas mediante un enlace a la dirección donde se ubica tal archivo. En términos generales, el archivo se aloja en forma de cache para cada tarea.

Únicamente dentro del código, ya sea Map ó Reduce, se solicita la ubicación del segundo archivo y su lectura de la forma convencional del lenguaje que se utilice, esto es mediante un *lector de Buffer*.

Por supuesto se debe agregar la librería correspondiente para utilizar *DistributedCache,* y resultando en una reducción en el tiempo y recursos consumidos por una segunda lógica de Map.

3.2.2.3. Primer y Segundo Algoritmo para aplicación del Kernel

Para el código Map, en la matriz de entrada se generan claves como índices de cada ventana, y se pasa como valores el elemento actual y los índices que el elemento ocupa dentro de la ventana (no de la matriz original), si es que el elemento esta dentro del rango de la ventana. Esto significa que es el mismo primero algoritmo de Max-Pooling con la excepción que, como valor, además de enviar el elemento, se envían sus índices respecto a la ventana.

class Mapper	Costo	Repeticiones
method Map (Key line-offset; Value index)		
Get M_r M_c K_r K_c , stride **from** DriverClas	C1	1
Set totalWindows = [(M_r - K_r)/stride]*[(M_c - K_c)/stride]	C2	1
for each (window : totalwindows)	C3	n
update window-span	C4	n-1
update window-indexes	C5	n-1
if (element ∈ window-span)	C6	n-1
set *newindex* as element_index_inside_the_window	C7	$\sum_{j=0}^{n-1} t_j$
Emit (Key *Window-index*, Value (*newindex* + *element*))	C8	$\sum_{j=0}^{n-1} t_j$
Next Window		

Figura 31. Primer Algoritmo Map Aplicación de Kernel

Por lo tanto solo se agrega una instrucción más dentro de la condicionante y el análisis de ejecución en la ecuación continúa siendo del tipo $an + b$:

$$T(n) = (C_3 + C_4 + C_5 + C_6 + C_7 + C_8)n + (C_1 + C_2 + C_4 + C_5 + C_6)$$

De igual manera se considera el segundo algoritmo para la aplicación del kernel, por supuesto pasando también los índices del elemento respecto a las dimensiones de la ventana:

```
class Mapper                                                    Costo   Repeticiones
  method  Map (Key line-offset; Value [indexes,element])        C1      1
  Get inputsize,  poolsize, stride from DriverClas              C2      1
  outputsize = (inputsize  – poolsize) / stride + 1

  if (⌊outputsize⌋ < outputsize)                                C3      1
  outputsize = ⌊outputsize++⌋                                   C4      t

  nm = ⌊ elementIndex / stride ⌋                                C5      1
  if (nm >= outputsize)                                         C6      1
    nm = outputsize - 1                                         C7      t

  while ( nm-span > elementIndexes &&  n >= 0)                  C8      n + 1
    set newindex as element index inside the window             C9      n
    Emit (Key [nm]; Value element)                              C10     n
    nm--                                                        C11     n
```

Figura 32. Segundo Algoritmo Map Aplicación de Kernel

Por lo tanto, la ecuación sigue la misma linealidad que la implementación anterior

$$T(n) = (C_8 + C_9 + C_{10} + C_{11})n + (C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8)$$

De esta forma, el análisis al comparar ambos algoritmos en esta implementación continúa de la misma forma, el primero respecto al total de ventanas y el segundo respecto a las ventanas que pertenezca el elemento.

En el código Reducer, se reciben los valores de cada ventana, tanto del filtro como de la matriz de entrada, por lo que por cada índice, se tendrán dos valores

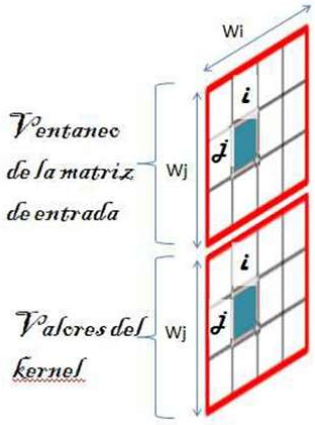


Figura 33. Representación del valor e índices que contendría un array de: a) la ventana aplicada a la matriz de entrada y b) los elementos del Kernel.

El código en este caso, considerando que el archivo que especifica el kernel solo contiene los índices y el número del elemento y no cuenta con una etiqueta que señale que es el filtro, realizará un primer bucle para almacenar cada elemento en un arreglo de tres dimensiones. La primera y segunda dimensión del

arreglo tendrán como límite el tamaño de la ventana (cantidad de columnas, cantidad de renglones, respectivamente). Se toma como ventaja que en el lenguaje utilizado, al iniciar un arreglo los valores iniciales son cero, la tercera dimensión será una bandera indicando si el elemento ya fue almacenado (recordar que se reciben dos elementos por cada índice) con un valor de uno y cero si aún no ha sido almacenado. En caso que la bandera sea uno, se multiplica el nuevo elemento con el almacenado previamente y se actualiza el valor.

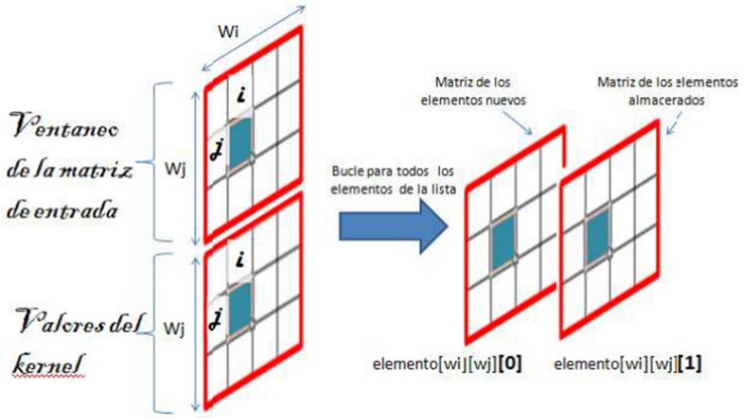


Figura 34. Representación de un bucle que asigna elementos provenientes de la primer etapa Map (Valores por cada clave) y de los elementos provenientes ya sea de una segunda etapa Map ó del archivo distribuido mediante un lector de buffer.

Un segundo bucle realiza la suma de todos los elementos multiplicados y al concluir envía como clave *null* y como valor el índice de la ventana y la suma total. La descripción de la etapa reduce en pseudocódigo es el siguiente.

class Reducer	Costo	Repeticiones
method Reduce (Key Window-Index; Value List[elementsFromMatrix,elementsfromkerne])		
for (element : elements)	C1	2n
if (element is repeated)	C2	n
element(wi,wj,1) = element(wi,wj,1) * element	C3	n
if (element is not repeated yet)	C4	n
element (wi,wj,1) = element	C5	n
next element		
sum ← 0	C6	1
for all (element[wi][wj][1])	C7	n + 1
sum ← sum + element([wi],[wj][1])	C8	n
next element[wi][wj][1]	C9	1
Emit(Key *null*; Value *Window-Index & sum*)		

Figura 35. Algoritmo Reduce de aplicación de kernel mediante múltiples tareas Map.

Con análisis en la ejecución considerando *n* como el tamaño de la ventana, la primera instrucción realiza un bucle *for* del doble del tamaño de la ventana, ya que se reciben dos de estas; una de la matriz de entrada y otra del kernel. Reordenando la ejecución de todas las instrucciones:

$$(n) = (2c_1 + c_2 + c_3 + c_4 + c_5 + c_7 + c_8)n + (c_6 + c_7 + c_9)$$

Esta etapa Reduce nuevamente sigue un patrón lineal con dependencia del tamaño de la ventana. Por lo tanto, al solo variar el tamaño de la matriz de entrada y no la ventana, el tiempo de ejecución se mantiene constante.

Para la utilización de archivo mediante cache distribuido se utiliza el siguiente algoritmo

	Costo	Repeticiones
class Reduce		
method Reduce (Key Window-Index; Value List[elements belonging to this window])		
Get TotalElements ***from*** CacheFile	C1	1
for each (element : TotalElements)	C2	n
element(wi,wj,1) = element	C3	n-1
next element	C4	n-1
for each (element : elements belonging to this window)	C5	n
element(wi,wj,1) = element(wi,wj,1) * element	C6	n-1
next element	C7	n-1
for each (element[wi],[wi],[1])	C8	n
sum ← sum + element[wi] [wi] [1]	C9	n-1
next element[wi] [wi] [1]	C10	n-1
Emit (Key *null*; Value *Window-Index + sum*)	C11	1

Figura 36. Algoritmo Reduce de aplicación de Kernel mediante Cache Distribuido

En donde se observa un primer ciclo que genera un arreglo a partir de los elementos del archivo distribuido. El segundo ciclo multiplica los elementos del arreglo generado previamente con los elementos recibidos de la tarea Map, para la ventana en turno. Además, un tercer ciclo realiza la suma total de los elementos en el arreglo generado. Al finalizar el código, se emite como clave *null* y valores la suma total.

$$T(n) = (C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8 + C_9 + C_{10})n + (C_1 + C_3 + C_4 + C_6 + C_7 + C_9 + C_{10} + C_{11})$$

Con linealidad $an + b$.

Capítulo 4

Experimentación

Para poder llevar a cabo las implementaciones de la sección previa se utiliza un servidor con las siguientes especificaciones.

Tabla 3. Información del servidor utilizado

Procesador, Intel Xeon E5 2.10GHz	32 CPUs (16 threads - 2 sockets)
Memoria Principal	32086MB ≈ 31GB
Almacenamiento	≈ 1.5 TB

El servidor cuenta con la instalación de la plataforma de código abierto Proxmox con el objetivo de crear un clúster de maquinas virtuales que conformen un clúster Hadoop y ejecutar las distintas pruebas.

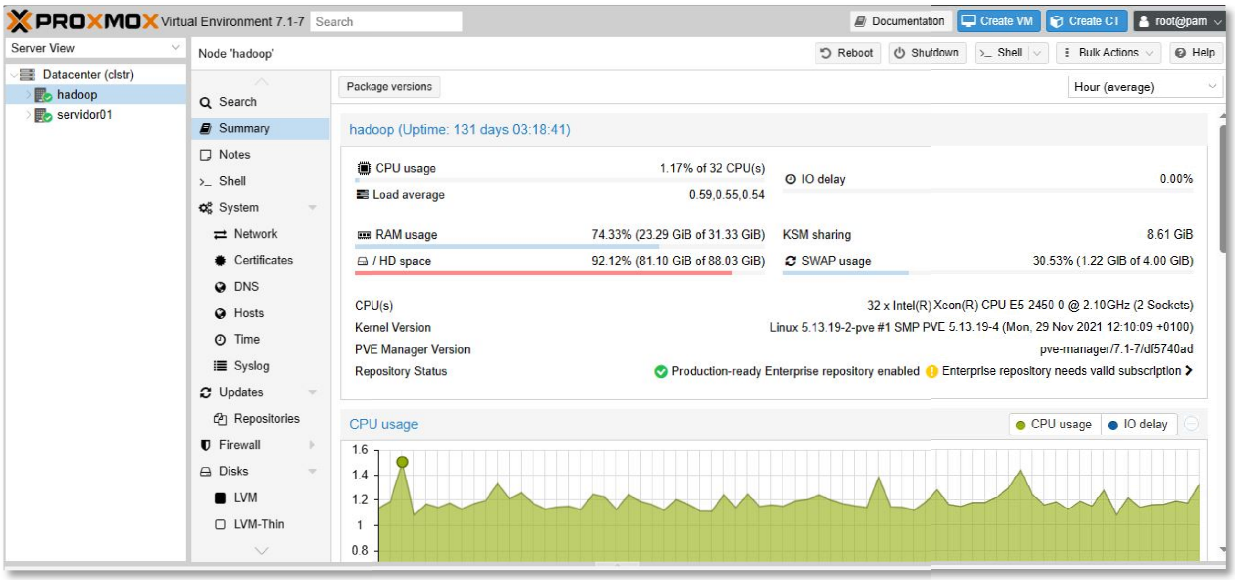


Figura 37. Dashboard de Proxmox instalado para los experimentos.

Se instala en cada máquina virtual como Sistema Operativo Ubuntu 22.04 y se realiza la instalación manual de Hadoop en cada una de las máquinas virtuales, de ahora en adelante referidas como nodos. A cada nodo se le asigna una cantidad de 100GB como almacenamiento y se varía la cantidad de memoria principal (referida en adelante como RAM) y la cantidad de CPU virtuales de cada nodo.

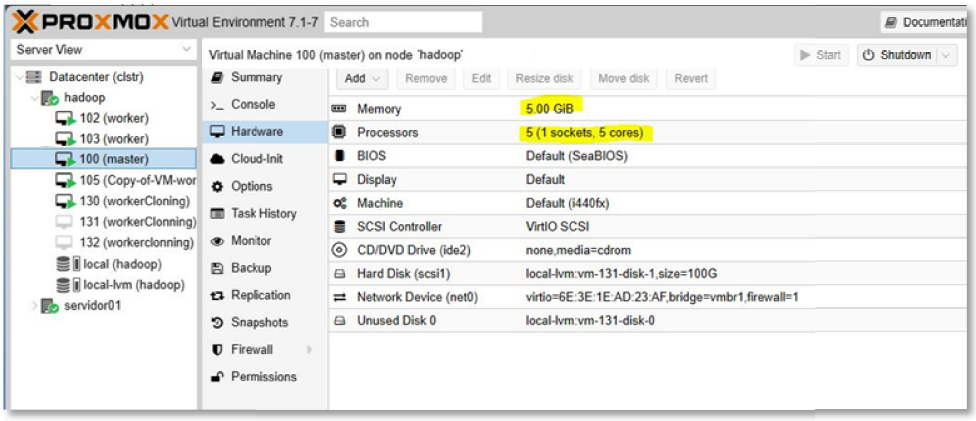


Figura 38. Configuración de la máquina virtual

La cantidad varía para ajustarse a los límites del servidor utilizado para las pruebas y observar el comportamiento variando en la cantidad de nodos de 4 hasta 6 nodos de los cuales sólo uno se designa como nodo maestro y la cantidad restante como nodo trabajador.

Una vez con el sistema operativo instalado en cada nodo, se descarga Hadoop y se instala la implementación en cada nodo. La implementación se lleva a cabo en este caso colocando el paquete de Hadoop dentro del directorio */usr/local*. Dentro del directorio Hadoop se agregan dos directorios adicionales nombrados como */tmp* y */logs* los cuales serán utilizados como directorios que el framework utilizará para almacenar información temporal que utiliza tanto HDFS como YARN cuando se solicite la ejecución de algún programa.

Se instalan los programas necesarios para la comunicación entre los nodos con privilegios de súper usuario y se genera claves de confianza las cuales se deberán enviar a cada nodo para acceder sin solicitar permisos de conexión remota:

```
sudo apt install openssh-server
sudo apt install ssh
ssh-keygen -t rsa
```

Se crea un usuario nuevo en específico para las ejecuciones relacionadas con Hadoop en todos los nodos con los privilegios de administrador o ejecutando los comandos desde el súper usuario. En este caso, se nombra como: *hadoop* con los siguientes comandos y se asigna un password para tal:

```
sudo adduser hadoop
sudo adduser hadoop sudo
passwd hadoop
```

Se especifica la resolución de nombres de anfitrión desde el archivo */etc/hosts* para todos los nodos colocando la dirección IP de cada nodo y el pseudo-nombre a utilizar:

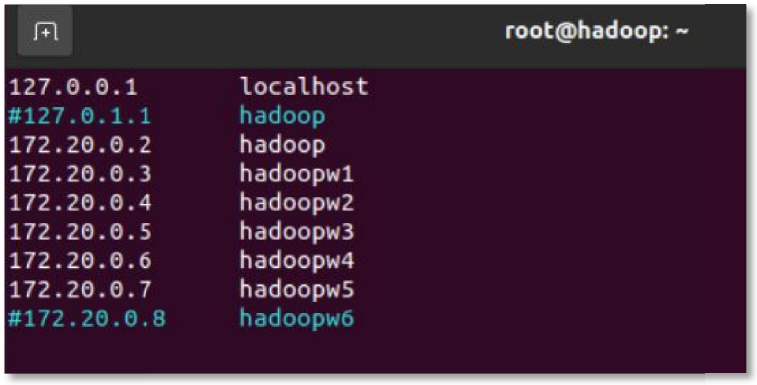


Figura 39. Configuración archivo hosts

Esta designación debe ser global para todos los nodos de forma que se reconozcan entre sí utilizando el mismo nombre. En este punto es importante notar que el usuario debe ser el mismo en todos los nodos pero el nombre de anfitrión o *hostname* debe ser único por cada nodo, coincidiendo la dirección *ip* de cada nodo con el nombre asignado desde el archivo */etc/hosts*; por ejemplo, para el nodo trabajador 2, la *ip* que tiene asignada es 172.20.0.4 y *hostname* esta designado como *hadoopw2*.

En cada nodo, desde el usuario creado, se generan llaves de confianza y se comparten con el resto de nodos:

```
ssh-keygen -t rsa -P ""
cp .ssh/id_rsa.pub .ssh/authorized_keys
ssh-copy-id "computer name"
```

Cada nodo debe tener instalado *openjdk* versión 8:

```
sudo apt install openjdk-8* -y
vi .bashrc
```

y agregada la ruta como variable de ambiente desde el archivo .bashrc,

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin
```

Se guarda el archivo y se aplican los cambios con el comando *source .bashrc*:

```
source .bashrc
```

Se asigna propietario al directorio */usr/local/hadoop* y se cambian los permisos:

```
sudo chown -R hadoopuser /usr/local/hadoop
sudo chmod -R 777 /usr/local/hadoop
```

Para poder ejecutar los comandos de Hadoop, se agregan las siguientes líneas en el archivo *.bashrc* y se aplican los cambios efectuados con *source .bashrc*

```
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export HDFS_NAMENODE_USER=hadoopuser
export HDFS_DATANODE_USER=hadoopuser
export HDFS_SECONDARYNAMENODE_USER=hadoopuser
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export HADOOP_YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
```

Dentro del archivo donde se aloja la configuración de ambiente de Hadoop, de igual forma para cada nodo se agregan las siguientes líneas en el archivo *hadoop-env.sh.* Tales líneas también se les puede retirar el símbolo de comentario y colocar la configuración que se ajuste al escenario, en lugar de ser agregadas:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_CONF_DIR=${HADOOP_HOME}/etc/hadoop
export HADOOP_LOG_DIR=${HADOOP_HOME}/logs
```

Para los propósitos de este texto, la siguiente configuración se aplicó a todos los nodos, *master y workers:*

Para el archivo: */usr/local/hadoop/etc/hadoop/core-site.xml*

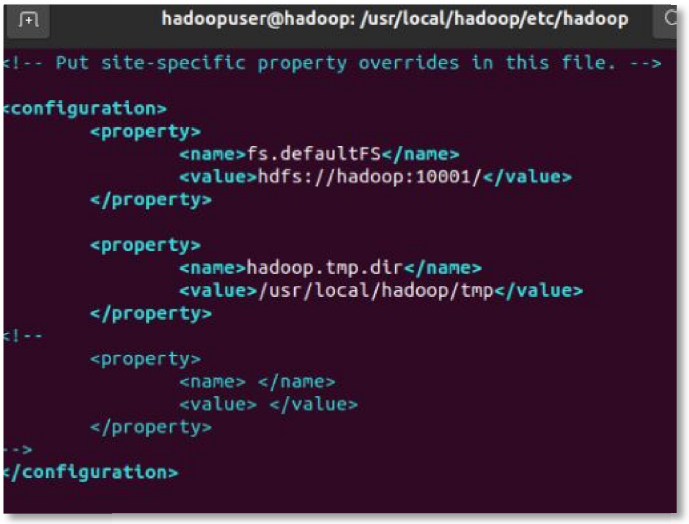


Figura 40. Configuración del archivo *core-site.xml*

Para el archivo: */usr/local/hadoop/etc/hadoop/hdfs-site.xml*

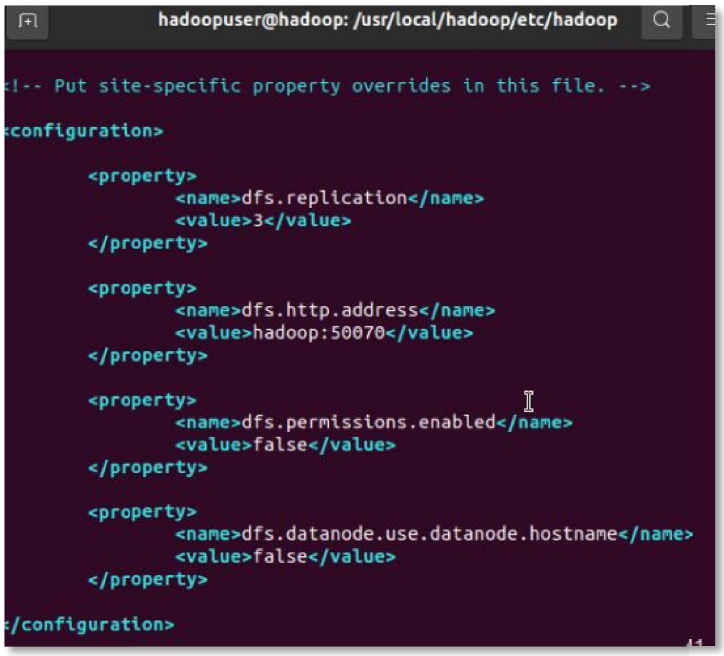
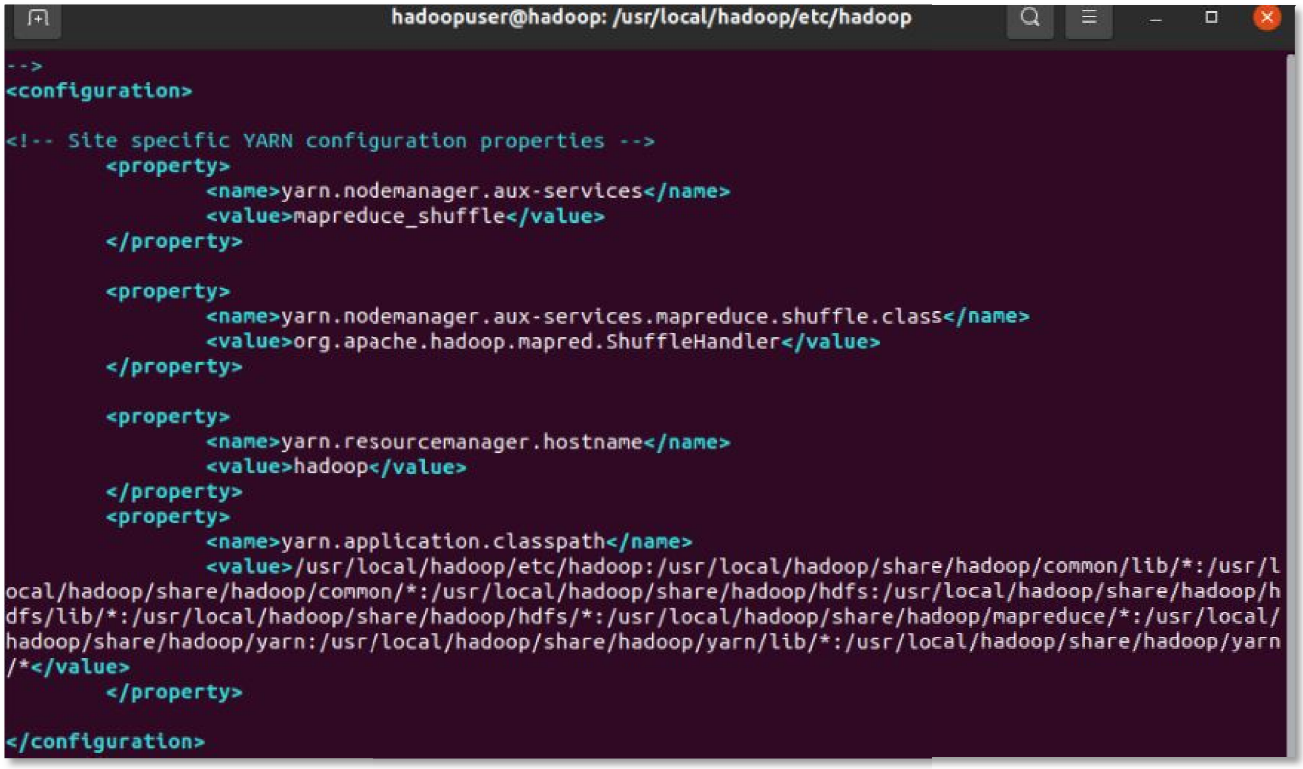


Figura 41. Configuración del archivo hdfs-site.xml

Para el archivo: */usr/local/hadoop/etc/hadoop/yarn-site.xml*




```
<configuration>

<!-- Site specific YARN configuration properties -->
        <property>
                <name>yarn.nodemanager.aux-services</name>
                <value>mapreduce_shuffle</value>
        </property>

        <property>
                <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
                <value>org.apache.hadoop.mapred.ShuffleHandler</value>
        </property>

        <property>
                <name>yarn.resourcemanager.hostname</name>
                <value>hadoop</value>
        </property>
        <property>
                <name>yarn.application.classpath</name>
                <value>/usr/local/hadoop/etc/hadoop:/usr/local/hadoop/share/hadoop/common/lib/*:/usr/l
ocal/hadoop/share/hadoop/common/*:/usr/local/hadoop/share/hadoop/hdfs:/usr/local/hadoop/share/hadoop/h
dfs/lib/*:/usr/local/hadoop/share/hadoop/hdfs/*:/usr/local/hadoop/share/hadoop/mapreduce/*:/usr/local/
hadoop/share/hadoop/yarn:/usr/local/hadoop/share/hadoop/yarn/lib/*:/usr/local/hadoop/share/hadoop/yarn
/*</value>
        </property>

</configuration>
```

Figura 42. Configuración del archivo yarn-site.xml

Para el archivo: */usr/local/hadoop/etc/hadoop/mapred-site.xml*

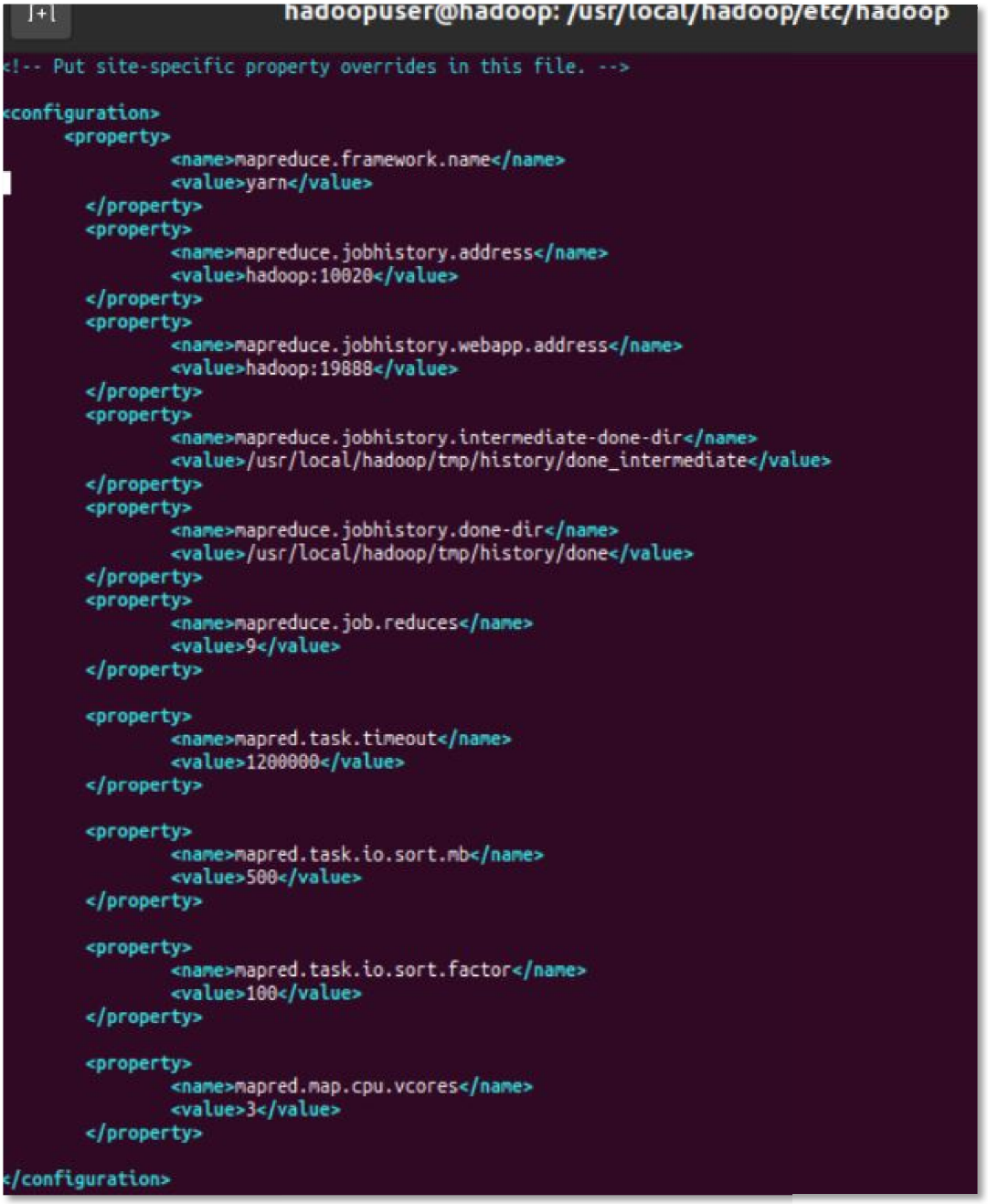


Figura 43. Configuración del archivo mapred-site.xml

Con las configuraciones realizadas en todos los nodos se inicializa Hadoop, invocando los procesos que coordinan su ejecución desde el nodo maestro. Para el ambiente HDFS se da formato a todo el clúster, aplicando para todos los nodos que lo conforman, desde el nodo maestro con el comando:

```
hdfs namenode -format
```

Una vez con el formato creado desde el nodo maestro, se inicializan los procesos de HDFS:

```
start-dfs.sh
```

Al inicializar HDFS, se inicializan los procesos de YARN:

```
start-yarn.sh
```

Se confirma que los procesos se encuentran activos ejecutando el comando *jps* primero en el nodo maestro:

```
hadoopuser@hadoop:~$ jps
4679 NameNode
31099 Jps
4941 SecondaryNameNode
5134 ResourceManager
```

Y después, en los nodos trabajadores:

```
hadoopuser@hadoopw2:~$ jps
2720 DataNode
2881 NodeManager
32982 Jps
```

Estos procesos deben estar activos en todos los nodos trabajadores del clúster Hadoop, indicando que el clúster se encuentra operando correctamente.

Para el empaquetado de los códigos a ejecutar en Hadoop, se utiliza *Apache maven* el cual debe reconocer que la versión de *openjdk* en uso es la versión 8 actualizando los complementos de este ejecutando el siguiente comando o mediante la ruta que corresponda al IDE en uso:

```
mvn --update-plugins
```

Para la configuración del proyecto que adjuntará todas las clases con el código de cada etapa, dentro del archivo *pom.xml* se coloca la siguiente configuración:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>hadoop.example.org</groupId>
    <artifactId>MaxPoolingForLoop</artifactId>
    <version>5000xEdge</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    <dependencies>
                <dependency>
                        <groupId>org.apache.hadoop</groupId>
                        <artifactId>hadoop-common</artifactId>
                        <version>3.2.1</version>
                </dependency>
    </dependencies>
    <repositories>
                <repository>
                        <id>apache</id>
                        <url>http://maven.apache.org</url>
                </repository>
    </repositories>
</project>
```

Figura 44. Configuración archivo pom.xml

Desde esta configuración se especifica que las clases empaquetadas en el archivo *jar* resultante se ubicaran siguiendo la ruta *hadoop.example.org* en *groupId*. El archivo *jar* resultante tendrá como nombre lo que contenga la etiqueta *artifactId* y asignándole una *versión*. Se utiliza el contenido propuesto de la imagen para la versión de Hadoop instalada en el clúster.

Con la información de *maven* anterior se especifican tres códigos para la ejecución del programa, uno para la clase de control o *driver*, uno para la clase *Map* y otro para la clase *Reduce*. Los códigos para cada una de estas clases se pueden encontrar en los anexos de este texto.

Dentro del código de la clase *Driver*, en las líneas correspondientes a la configuración, se varían los parámetros del tamaño de la matriz. Para ambas implementaciones, Max-Pooling y Aplicación de un Kernel, la especificación de la configuración queda de la siguiente manera:

```java
Configuration conf = new Configuration();
conf.set("m", "23000");
conf.set("n", "23000");
conf.set("w", "5");
conf.set("s", "3");
```

En donde se pasa como parámetros para la matriz de entrada la cantidad de renglones con la etiqueta *m*, la cantidad de columnas *n*, el tamaño del kernel, tanto para columnas y renglones con la etiqueta *w* y los

pasos o deslizamientos de la ventana con etiqueta *s*.

Se coloca un nombre al *Job* a ejecutar y se especifica el nombre de la clase *driver*:

```
Job job = new Job(conf, jobName: "MatrixMultiplication");
job.setJarByClass(MultipleInputDriver.class);
```

Para especificar el tipo de datos en que interpretará el archivo de entrada y se escribirá el archivo resultante se especifica con las líneas:

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

Para identificar la clase Map y Reduce se utilizan las propiedades *setMapperClass* y *setReducerClass* y se colocan como argumentos el nombre de estas.

```
job.setMapperClass(Mapper1.class);
job.setReducerClass(Reduce.class);
```

Para el formato en que se procesara el archivo de entrada y se devolverá el archivo de salida resultante se colocan las líneas:

```
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
```

Para poder utilizar la propiedad de C*ache Distribuida*, se debe especificar mediante la propiedad *addCacheFile()* de la clase *DistributedCache* importada con el paquete *org.apache.hadoop.filecache.DistributedCache* de la API de Hadoop y colocando como argumento el nombre de tal archivo o si pasará como argumento al ejecutar el *Job*. Esta propiedad, para este trabajo, se utiliza únicamente en la implementación de Aplicación de Kernel para enviar la matriz de tal kernel a los nodos del cluster.

```
//      DistributedCache.addCacheFile(new Path(args[1]).toUri(), job.getConfiguration());
```

Al final se especifica cuál será el archivo que reconocerá como entrada, y el archivo de salida. Recordar que el archivo de entrada debe estar previamente almacenado en el cluster HDFS para que reconozca

tales argumentos. La propiedad *setOutputPath* indica el nombre que se le dará al directorio en donde almacenará el resultado.

```
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.waitForCompletion( verbose: true);
```

Con esta configuración desde el código driver, las clases *Map y Reduce* pueden acceder a los parámetros descritos mediante una instancia serializada del tipo *Configuration*, como el total de columnas y renglones de la(s) matriz(ces).

Capítulo 5

Resultados

En esta sección se exponen los resultados obtenidos a partir del código descrito en las dos implementaciones del capítulo anterior. Alternativamente, el código utilizado para ejecutar en el clúster de máquinas virtuales se encuentra en el anexo a este trabajo.

5.1. Max-Pooling

Para ambas implementaciones, Max-Pooling, y Aplicación de un kernel sobre una matriz de entrada, se realizó la ejecución de los dos algoritmos del Capítulo 3, esto es, el código utilizando *ciclo for* recorriendo la ventana sobre la Matriz de entrada en la etapa Map (primer algoritmo) y obtener el valor máximo en la etapa Reduce. Para el segundo algoritmo, se realiza el recorrido de la ventana únicamente sobre el elemento siendo procesado. Con los resultados obtenidos y descritos a continuación, los tamaños de las matrices se limitaron a un tiempo no mayor de 24 horas, por lo que en las siguientes tablas, la etiqueta de *Indefinido* indica que el tiempo superó tal límite. De la misma forma, aquellos resultados con la etiqueta *2 Intentos Fallidos* indica que el framework no pudo completar el primer *job* y reinició la ejecución, donde el segundo intento tampoco pudo completarse de forma satisfactoria. Para la duración, en las siguientes tablas se describió en la forma *hh:mm:ss* (*horas:minutos:segundos*, respectivamente).

Tabla 5 - 1. Primer algoritmo MaxPooling. 3 nodos trabajadores 1 nodo maestro.

1 Maestro	8Gb RAM 8 CPU's	
3 Trabajadores	7Gb RAM 7 CPU's	
Tamaño Matriz	**Ventanas Generadas**	**Duración**
1k * 1k (1000000)	108900	0:05:26
2k * 2k (4000000)	440896	1:12:39
3k * 3k (9000000)	994009	1:11:33
4k * 4k (16000000)	1768900	11:56:06
5k * 5k (25000000)	2768896	19:26:40
6k * 6k (36000000)	3988009	Indefinido

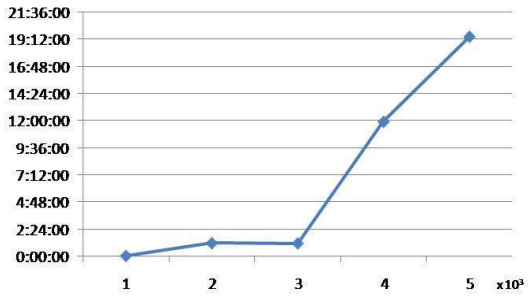


Tabla 5 - 2. Segundo algoritmo MaxPooling. 3 nodos trabajadores 1 nodo maestro.

1 Maestro	8Gb RAM 8 CPU's	
3 Trabajadores	7Gb RAM 7 CPU's	
Tamaño Matriz	**Ventanas Generadas**	**Duración**
5k * 5k (25000000)	2775556	0:01:59
10k * 10k (100000000)	11102224	0:06:12
14k * 14k (196000000)	21771556	0:18:12
18k * 18k (324000000)	35988001	0:37:41
20k * 20k (400000000)	44435556	0:55:02
24k * 24k (576,000,000)	63984001	1:41:33
28k * 28k (784,000,000)	87048900	4:27:33

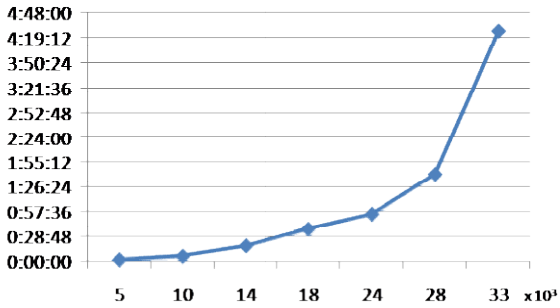


Tabla 5 - 3. Primer algoritmo MaxPooling. 4 nodos trabajadores 1 nodo maestro.

1 Maestro	6 Gb RAM 6 CPU's	
4 Trabajadores	6 Gb RAM 6 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
1k * 1k (1000000)	108900	0:05:35
2k * 2k (4000000)	440896	1:13:04
3k * 3k (9000000)	994009	5:55:26
4k * 4k (16000000)	1768900	12:09:11
5k * 5k (25000000)	2768896	19:03:28
6k * 6k (36000000)	3988009	Indefinido

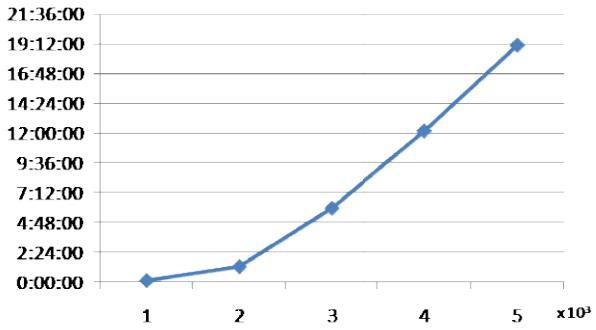


Tabla 5 - 4. Segundo algoritmo MaxPooling. 4 nodos trabajadores 1 nodo maestro.

1 Maestro	6Gb RAM 6 CPU's	
4 Trabajadores	6Gb RAM 6 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
5k * 5k (25000000)	2775556	00:02:48
10k * 10k (100000000)	11102224	00:17:27
14k * 14k (196000000)	21771556	00:44:06
18k * 18k (324000000)	35988001	2 Intentos Fallidos
20k * 20k (400000000)	44435556	2 Intentos Fallidos
24k * 24k (576,000,000)	63984001	2 Intentos Fallidos

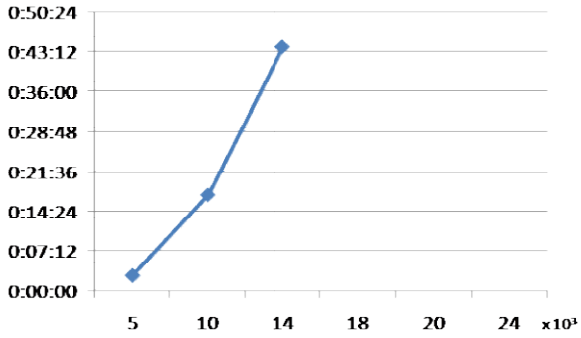


Tabla 5 - 5. Primer algoritmo MaxPooling. 5 nodos trabajadores 1 nodo maestro.

1 Maestro	5Gb RAM 5 CPU's	
5 Trabajadores	5Gb RAM 5 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
1k * 1k (1000000)	108900	0:05:50
2k * 2k (4000000)	440896	1:14:26
3k * 3k (9000000)	994009	5:34:10
4k * 4k (16000000)	1768900	7:42:56
5k * 5k (25000000)	2768896	18:41:29
6k * 6k (36000000)	3988009	Indefinido

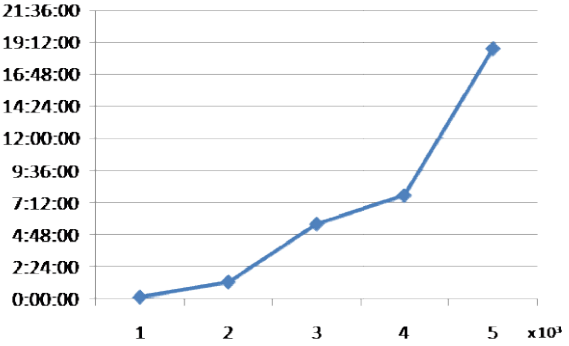
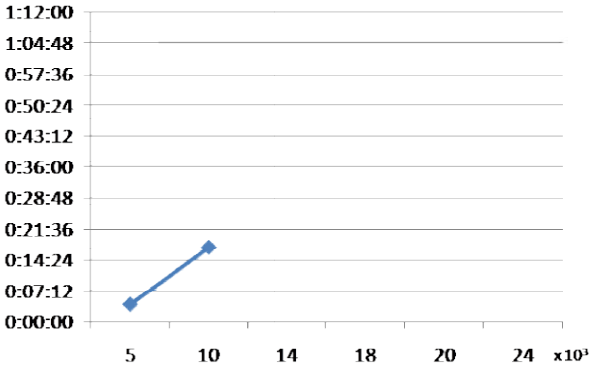


Tabla 5 - 6. Segundo algoritmo MaxPooling. 5 nodos trabajadores 1 nodo maestro.

1 Maestro	5Gb RAM 5 CPU's	
5 Trabajadores	5Gb RAM 5 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
5k * 5k (25000000)	2775556	0:03:53
10k * 10k (100000000)	11102224	0:17:19
14k * 14k (196000000)	21771556	2 Intentos Fallidos
18k * 18k (324000000)	35988001	2 Intentos Fallidos
20k * 20k (400000000)	44435556	2 Intentos Fallidos
24k * 24k (576,000,000)	63984001	2 Intentos Fallidos



5.2. Filtro-Kernel

Para la aplicación de un kernel sobre la matriz de entrada, se utilizó el cache Distribuido que el Framework ofrece con el objetivo de enviar a las tareas (de forma global a cada nodo) el archivo con el contenido del filtro, tomando ventaja que este es de un peso reducido ya que el tamaño del filtro aplicado es de 5 renglones y 5 columnas.

Tabla 5 - 7. Primer algoritmo Aplicación de Kernel. 3 nodos trabajadores 1 nodo maestro.

1 Maestro	7 Gb RAM 7 CPU's	
3 Trabajadores	8 Gb RAM 8 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
1k * 1k (1000000)	108900	0:05:17
2k * 2k (4000000)	440896	01:08:14
3k * 3k (9000000)	994009	02:21:52
4k * 4k (16000000)	1768900	11:42:01
5k * 5k (25000000)	2768896	18:48:22
10k * 10k (100000000)	11088900	Indefinido

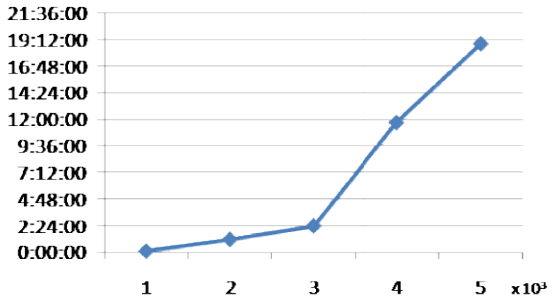


Tabla 5 - 8. Segundo algoritmo Aplicación de Kernel. 3 nodos trabajadores 1 nodo maestro.

1 Maestro	7 Gb RAM 7 CPU's	
3 Trabajadores	8 Gb RAM 8 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
5k * 5k (25000000)	2775556	0:03:33
10k * 10k (100000000)	11102224	0:09:08
14k * 14k (196000000)	21771556	0:30:11
18k * 18k (324000000)	35988001	0:39:35
20k * 20k (400000000)	44435556	0:57:03
28k * 28k (784,000,000)	87048900	1:35:28

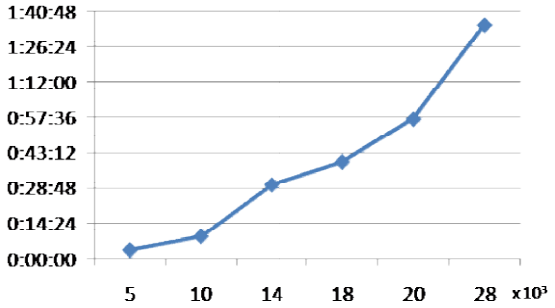
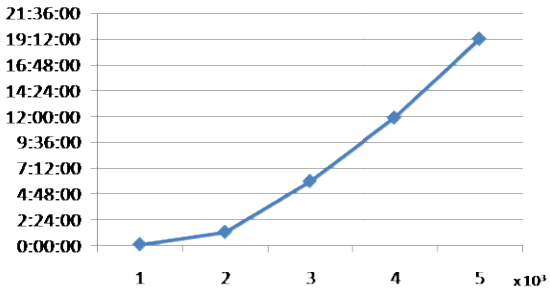


Tabla 5 - 9. Primer algoritmo Aplicación de Kernel. 4 nodos trabajadores 1 nodo maestro.

1 Maestro	6Gb RAM 6 CPU's	
4 Trabajadores	6Gb RAM 6 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
1k * 1k (1000000)	108900	0:05:10
2k * 2k (4000000)	440896	1:13:35
3k * 3k (9000000)	994009	5:57:39
4k * 4k (16000000)	1768900	11:51:04
5k * 5k (25000000)	2768896	19:10:24

Tabla 5 - 10. Segundo algoritmo Aplicación de Kernel. 4 nodos trabajadores 1 nodo maestro.

1 Maestro	6Gb RAM 6 CPU's	
4 Trabajadores	6Gb RAM 6 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
5k * 5k (25000000)	2775556	0:04:30
10k * 10k (100000000)	11102224	0:15:08
14k * 14k (196000000)	21771556	0:41:40
18k * 18k (324000000)	35988001	1:19:42
20k * 20k (400000000)	44435556	1:55:41
28k * 28k (784,000,000)	87048900	2 Intentos Fallidos



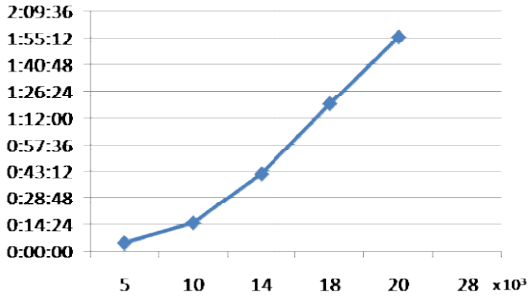


Tabla 5 - 11. Primer algoritmo Aplicación de Kernel. 5 nodos trabajadores 1 nodo maestro.

1 Maestro	5Gb RAM 5 CPU's	
5 Trabajadores	5Gb RAM 5 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
1k * 1k (1000000)	108900	0:05:24
2k * 2k (4000000)	440896	1:31:14
3k * 3k (9000000)	994009	6:03:02
4k * 4k (16000000)	1768900	11:53:21
5k * 5k (25000000)	2768896	19:16:14

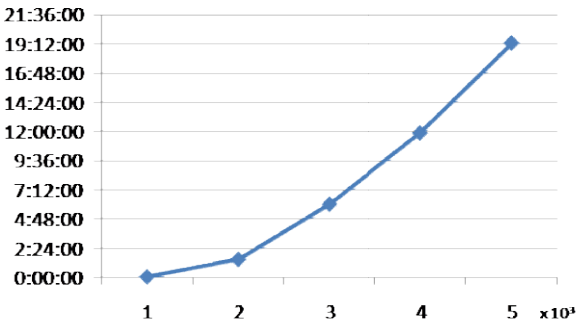
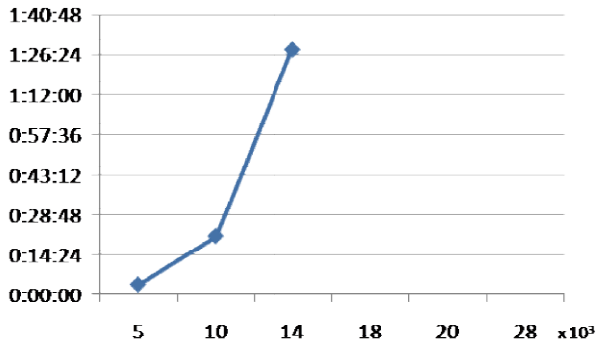


Tabla 5 - 12. Segundo algoritmo Aplicación de Kernel. 5 nodos trabajadores 1 nodo maestro.

1 Maestro	5Gb RAM 5 CPU's	
5 Trabajadores	5Gb RAM 5 CPU's	
Tamaño Matriz	Ventanas Generadas	Duración
5k * 5k (25000000)	2775556	0:03:32
10k * 10k (100000000)	11102224	0:21:08
14k * 14k (196000000)	21771556	1:28:17
18k * 18k (324000000)	35988001	2 Intentos Fallidos
20k * 20k (400000000)	44435556	2 Intentos Fallidos
28k * 28k (784,000,000)	87048900	2 Intentos Fallidos



Conclusiones

A partir de los resultados obtenidos se puede observar que el segundo algoritmo resultó como una mejor alternativa para abordar el seccionamiento de la matriz de entrada con tiempos considerablemente menor a los del primer algoritmo y por lo tanto con la posibilidad de incrementar el tamaño de la matriz de entrada. Debido a esto, a pesar de la cantidad de recursos disponibles para un clúster operando Hadoop, alternativas en la descripción de la lógica del programa para obtener los resultados, esto es el archivo de salida, es de gran relevancia.

El framework de MapReduce, siguiendo el patrón arquitectural Maestro-Trabajador, es posible manejarlo para una gran cantidad de propósitos conociendo la forma en que las divisiones del archivo de entrada serán procesadas. Esto es bien representado con la comparación en el tiempo de procesamiento utilizando ambos códigos bajo la misma cantidad de recursos asignados a cada nodo. Alternativamente, las distintas funciones que las versiones actuales de Hadoop ofrecen, son de utilidad para el procesamiento por *lote*, tal como la función *DistributedCache* para distribuir archivos a cada tarea cuando estos son de un tamaño considerablemente pequeño.

Por lo anterior, de este trabajo se concluye:

- Si la lógica de programación pretende utilizar un archivo en común para cada tarea o etapa, Map o Reduce, en lugar de determinar una segunda lógica de ejecución Map, es mejor alternativa enviar el archivo a los nodos mediante *DistributedCache* considerando un tamaño limitado en tal archivo, evitando así tiempo de procesamiento y utilización de recursos.

- Considerables alternativas en lógica de Programación. Ajustar diferentes maneras de abordar el objetivo del procesamiento al framework de MapReduce tiene un impacto importante para el procesamiento de Big-Data.

- Permitir variar las características por defecto en el framework como cambiar el tiempo de respuesta de cada nodo para reportar su estatus así como la cantidad de memoria permitida para la utilización de disco en la etapa intermedia.

Como propuesta a futuro se busca poder abordar de manera cíclica distintas ejecuciones en forma de *pipelining* y con una mayor cantidad de filtros sobre la lógica de programación que se utilizó con la estrategia de deslizamiento sobre el elemento al tiempo que se obtienen los tiempos de ejecución con una mayor cantidad de recursos.

Adicionalmente, debido a los tiempos de ejecución logrados con el segundo algoritmo en el seccionamiento de ambas matrices, resulta como una alternativa viable para utilizarse sobre la operación en multiplicación de matrices. Lo anterior debido al alto costo que esta operación resulta en matrices de gran tamaño. Considerando esta propuesta a futuro, se agrega en el apéndice de este trabajo

una introducción a la multiplicación de matrices utilizada sobre el framework de MapReduce y la forma de abordar esta operación. Dentro de este apartado, primero se describe la forma de llevar a cabo la ejecución de dos aplicaciones MapReduce; la primera ejecución realizando un reordenamiento de la matriz de entrada con el propósito de obtener las claves y valores en la forma que la segunda ejecución utilizará y la cual realizará la última operación, adición, sobre las claves recibidas y devolviendo el resultado en formato HDFS. Igualmente se explica y describe el algoritmo propuesto bajo una sola ejecución MapReduce como se puede comprender en (Leskovec, Rajaraman, & Ullman, 2019).

Esta propuesta de trabajo a futuro resulta conveniente ya que existen distintas formas de abordar el problema con la propuesta del seccionamiento del segundo algoritmo como; una reducción en el número de claves generadas, incrementando el tamaño de la ventana; o una agregación local para la etapa intermedia, implementando un combinador.

Anexo – Códigos

a) Código Obtención de ventana por medio del cociente Entero.

i. Código Controlador[†††]

```
package hadoop.example.org;

import java.io.PrintStream;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.filecache.ClientDistributedCacheManager;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class Driver {
    public static void main(String[] args) throws Exception {

        if(args.length != 3)
        {
            System.err.println("Ingresar files de entrada y salida <in_dir> <in2_dir> <out_dir>");
            System.exit(2);
        }
        Configuration conf = new Configuration();
        conf.set("m", "30000");
        conf.set("n", "30000");
        conf.set("w", "5");
        conf.set("s", "3");
        Job job = new Job(conf, "KernelByQuotient");
        job.setJarByClass(Driver.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        DistributedCache.addCacheFile(new Path(args[1]).toUri(), job.getConfiguration());
        job.setMapperClass(Mapper1.class);
        job.setReducerClass(Reducer.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[2]));
        job.waitForCompletion(true);

    }
}
```

[†††] Ejemplo para matriz de entrada de 30000 Renglones/Columnas, Ventana de 5 renglones/columnas, deslizamientos de 3 pasos.

ii. Código Map.

```
package hadoop.example.org;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class Mapper1 extends Mapper<LongWritable, Text, Text, Text> {
    private static int inputheight, inputwidth, poolsize, stride, outputheight, outputwidth;
    private static String FILE_TAG = "F1";

    public void setup(Mapper<LongWritable, Text, Text, Text>.Context context) {
        Configuration configuration = context.getConfiguration();
        inputheight = Integer.parseInt(configuration.get("m"));
        inputwidth = Integer.parseInt(configuration.get("n"));
        poolsize = Integer.parseInt(configuration.get("w"));
        stride = Integer.parseInt(configuration.get("s"));
        outputheight = (inputheight - poolsize) / stride + 1;
        outputwidth = (inputwidth - poolsize) / stride + 1;
    }
    public void map(LongWritable rowKey, Text value, Mapper<LongWritable, Text, Text, Text>.Context context)
            throws IOException, InterruptedException {
        Text outputKey = new Text(), outputValue = new Text();
        String[] indicesAndValue = value.toString().split(",");
        int g = Integer.parseInt(indicesAndValue[0]);
        int h = Integer.parseInt(indicesAndValue[1]);
        int f = Integer.parseInt(indicesAndValue[2]);
        int windowrow = 0;
        int windowcol = 0;

        double a = (double)(inputheight - poolsize) / (double)stride + 1.0;
        outputheight = (Math.floor(a) < a) ? ++outputheight : outputheight;

        a = (double)(inputwidth - poolsize) / (double)stride + 1.0;
        outputwidth = (Math.floor(a) < a) ? ++outputwidth : outputwidth;

        int m = g / stride;
        int n = h / stride;

        m = (m >= outputheight) ? (outputheight - 1) : m;
        n = (n >= outputwidth) ? (outputwidth - 1) : n;

        int maxm = m * stride + poolsize;
        while(maxm > g && m >= 0){
            int maxn = n * stride + poolsize;
            while (maxn > g && n >= 0){
                windowcol = h - n * stride;
                windowrow = g - m * stride;
                outputKey.set(m + "," + n);
                outputValue.set(windowrow + "," + windowcol + "," + f);
                context.write(outputKey, outputValue);
                --n;
                maxn = n * stride + poolsize;
            }
            --m;
            maxm = m * stride + poolsize;
        }
    }
}
```

iii. Código Reduce

```
package hadoop.example.org;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class Reducer extends Reducer<Text, Text, Text, Text> {
    private static int poolsize;
    private Path cacheFilePath;
    public MultipleInputReducer() {
    }
    public void setup(Reducer<Text, Text, Text, Text>.Context context) throws IOException, InterruptedException {
        super.setup(context);
        Configuration configuration = context.getConfiguration();
        poolsize = Integer.parseInt(configuration.get("w"));
        try {
            cacheFilePath = DistributedCache.getLocalCacheFiles(context.getConfiguration())[0];
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
    public void reduce (Text key, Iterable<Text> textValues, Reducer<Text, Text, Text, Text>.Context context)
            throws IOException, InterruptedException {

        BufferedReader br = new BufferedReader(new FileReader(cacheFilePath.toString()));
        double suma = 0.0;
        double[][][] Filtering = new double[poolsize][poolsize][3];

        String line;
        while ((line = br.readLine()) != null) {
            String[] stringValues = line.split(",");
            int row = Integer.parseInt(stringValues[0]);
            int column = Integer.parseInt(stringValues[1]);
            double element = (double)Integer.parseInt(stringValues[2]);
            Filtering[row][column][1] = element;
            Filtering[row][column][2] = 1.0;
        }

        br.close();

        Iterator variaIter = textValues.iterator();
        while(variaIter.hasNext()){
            Text textValue = (Text)variaIter.next();
            String[] stringValues = textValue.toString().split(",");
            int row = Integer.parseInt(stringValues[0]);
            int column = Integer.parseInt(stringValues[1]);
            double element = (double)Integer.parseInt(stringValues[2]);
            Filtering[row][column][0] = element;
            if (Filtering[row][column][2] == 1.0) {
                Filtering[row][column][1] *= Filtering[row][column][0];
                Filtering[row][column][2] = 2.0;
            }
        }
        for(int i = 0; i < poolsize; ++i) {
            for(int j = 0; j < poolsize; ++j) {
                suma += Filtering[i][j][1];
            }
        }

        context.write(null, new Text(key.toString() + "," + Double.toString(suma)));

    }
}
```

b) Código Obtención de ventana por medio de Ciclo For.

i. Código Controlador ‡‡‡

```
package org.example;

import java.io.PrintStream;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.filecache.ClientDistributedCacheManager;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class Driver {

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: MatrixMultiply <in_dir> <out_dir>");
            System.exit(2);
        }

        Configuration conf = new Configuration();
        conf.set("m", "2000");
        conf.set("n", "2000");
        conf.set("w", "5");
        conf.set("s", "3");

        Job job = new Job(conf, "Kernel");
        job.setJarByClass(Driver.class);

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

        DistributedCache.addCacheFile(new Path(args[1]).toUri(), job.getConfiguration());

        job.setMapperClass(Map.class);
        job.setReducerClass(Reducer.class);

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[2]));
        job.waitForCompletion(true);
    }
}
```

‡‡‡ Matriz de entrada de 2000 Renglones/Columnas, Ventana de 5 renglones/columnas y deslizamientos de 3 pasos.

ii. Código Map.

```
package org.example;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class Map extends Mapper<LongWritable, Text, Text, Text> {
    public Map() {
    }

    public void map(LongWritable key, Text value, Mapper<LongWritable, Text, Text, Text>.Context context)
            throws IOException, InterruptedException {

        Configuration conf = context.getConfiguration();
        String[] indicesAndValue = value.toString().split(",");
        Text outputKey = new Text(), outputValue = new Text();

        int inputHeight = Integer.parseInt(conf.get("m"));
        int inputWidth = Integer.parseInt(conf.get("n"));
        int stride = Integer.parseInt(conf.get("s"));
        int poolSize = Integer.parseInt(conf.get("w"));
        int outputHeight = (inputHeight - poolSize) / stride + 1;
        int outputWidth = (inputWidth - poolSize) / stride + 1;

        int g = Integer.parseInt(indicesAndValue[0]);
        int h = Integer.parseInt(indicesAndValue[1]);
        String f = indicesAndValue[2];
        int windowrow = 0;
        int windowcol = 0;

        for(int i = 0; i < outputHeight; ++i) {
            for(int j = 0; j < outputWidth; ++j) {
                if (g >= i * stride && g < i * stride + poolSize && h >= j * stride && h < j * stride + poolSize) {
                    windowcol = h - j * stride;
                    windowrow = g - i * stride;
                    outputKey.set(i + "," + j);
                    outputValue.set(windowrow + "," + windowcol + "," + f);
                    context.write(outputKey, outputValue);
                }
            }
        }
    }
}
```

iii. Código Reduce

```
package hadoop.example.org;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.filecache.DistributedCache;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class Reducer extends Reducer<Text, Text, Text, Text> {
    private static int poolsize;
    private Path cacheFilePath;
    public MultipleInputReducer() {
    }
    public void setup(Reducer<Text, Text, Text, Text>.Context context) throws IOException, InterruptedException {
        super.setup(context);
        Configuration configuration = context.getConfiguration();
        poolsize = Integer.parseInt(configuration.get("w"));
        try {
            cacheFilePath = DistributedCache.getLocalCacheFiles(context.getConfiguration())[0];
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
    public void reduce (Text key, Iterable<Text> textValues, Reducer<Text, Text, Text, Text>.Context context)
            throws IOException, InterruptedException {

        BufferedReader br = new BufferedReader(new FileReader(cacheFilePath.toString()));
        double suma = 0.0;
        double[][][] Filtering = new double[poolsize][poolsize][3];

        String line;
        while ((line = br.readLine()) != null) {
            String[] stringValues = line.split(",");
            int row = Integer.parseInt(stringValues[0]);
            int column = Integer.parseInt(stringValues[1]);
            double element = (double)Integer.parseInt(stringValues[2]);
            Filtering[row][column][1] = element;
            Filtering[row][column][2] = 1.0;
        }

        br.close();

        Iterator variaIter = textValues.iterator();
        while(variaIter.hasNext()){
            Text textValue = (Text)variaIter.next();
            String[] stringValues = textValue.toString().split(",");
            int row = Integer.parseInt(stringValues[0]);
            int column = Integer.parseInt(stringValues[1]);
            double element = (double)Integer.parseInt(stringValues[2]);
            Filtering[row][column][0] = element;
            if (Filtering[row][column][2] == 1.0) {
                Filtering[row][column][1] *= Filtering[row][column][0];
                Filtering[row][column][2] = 2.0;
            }
        }
        for(int i = 0; i < poolsize; ++i) {
            for(int j = 0; j < poolsize; ++j) {
                suma += Filtering[i][j][1];
            }
        }

        context.write(null, new Text(key.toString() + "," + Double.toString(suma)));

    }
}
```

Apéndice

Multiplicación de Matrices mediante MapReduce. Panorama General.

La multiplicación de Matrices es una operación largamente estudiada, como se describe en (Qasem, Sarhan, Qaddoura, & Mahafzah, 2017) a la fecha descrita, utilizando el framework MapReduce buscando mejorar los tiempos de operación y considerando matrices de un gran tamaño, esto es, matrices que superen valores de 1000 renglones y 1000 columnas buscando una solución para archivos de entrada arriba de 100MB. Como una implementación que permita observar y entender la secuencia en multiplicación de matrices con MapReduce, se detallará el caso más esencial donde los valores de las matrices son agrupados con el propósito de generar claves que correspondan a cada sumando de los valores resultantes en la matriz final.

El procedimiento de la multiplicación se describe de la siguiente forma (Leskovec, Rajaraman, & Ullman, 2019):

> Sea A una matriz cuyos elementos se identifican en la forma a_{ij} en el renglón *i* y columna *j*, y una segunda matriz B cuyos elementos se identifican en la forma b_{jk} en el renglón *j* y columna *k*, se obtiene su producto ***P = A B*** como la matriz *P* cuyos elementos son p_{ik} en el renglón *i* columna *k* con valores determinados por:
>
> $$p_{ik} = \sum_j a_{ij} b_{jk}$$
>
> Observando entonces que el número de columnas de A es igual al total de renglones de B.

Cada elemento de una matriz se identifica mediante la relación del; índice de renglón, índice columna y el valor del elemento. Por lo que la matriz A se describe mediante las tuplas (i,j,a_{ij}) y la matriz B se describe mediante las tuplas (j,k,b_{jk}). Es posible, como en este caso, que los índices de cada elemento de una matriz sean dados directamente como entrada para su procesamiento, en vez de tener que ser escritos de forma explícita. Por lo tanto, en el framework MapReduce, la descripción de la función Map construiría los componentes *I, J y K* de cada tupla para la posición de cada elemento. Por lo que, para la multiplicación $A * B$ se obtendría una tupla en donde el único elemento en común es el componente *J*, esto es (i,j,k,a_{ij},b_{jk}), de donde se busca que la tupla como último elemento equivalga a la multiplicación de los elementos, $(i,j,k,a_{ij}*b_{jk})$ obteniendo una tupla de 4 elementos. La composición de esta tupla se llevaría a cabo desde una primera ejecución de

MapReduce. En forma de cascada, una segunda ejecución MapReduce realizaría la sumatoria de cada elemento que tenga (i,k) en común, es decir los resultados de la multiplicación de tales elementos:

1° MapReduce:
   a. Map: por cada elemento de ambas matrices, se crea el par clave-valor $(j\ ,\ [A,i,a_{ij}])$ ó $(j\ ,\ [B,k,b_{jk}])$ con los caracteres A y B como indicativo de la matriz de origen respectivamente.
   b. Reduce: Se recibe la clave j y como valor todos los elementos a los que se les asignó j. Como resultado se generan claves (i,k) y como valor el producto de la multiplicación $a_{ij}*b_{jk}$ para todos los elementos recibidos.

2° MapReduce
   a. Map: Se recibe el resultado de la ejecución MapReduce anterior con clave (i,k) y valor una lista de todos los elementos $(a_{ij}*b_{jk})$ con la misma clave, lo cual será transmitido directamente en la etapa Reduce
   b. Reduce: En esta etapa, cada instancia recibe una clave (i,k) con la correspondiente lista de valores. A la salida se genera la misma clave y como resultado la suma de todos los elementos.

El objetivo principal de utilizar el framework MapReduce en un sistema distribuido es evitar la comunicación de transmitir datos a las unidades de procesamiento. Por esta razón, la solución anterior de utilizar dos ejecuciones MapReduce no resulta conveniente. Para resumir la ejecución en cascada de dos aplicaciones MapReduce, se describirá la multiplicación de ambas matrices en una solo ejecución MapReduce con el costo en el incremento de procesamiento descrito en (Leskovec, Rajaraman, & Ullman, 2019). Desde esta perspectiva, el enviar el procesamiento, aunque este sea mayor, a las unidades donde se alojan las divisiones de los datos, completamente justifica el peso en la comunicación de las etapas intermedias.

Se identifica que un solo elemento de cualquiera matriz de entrada es necesario para el cálculo de distintos elementos en la matriz resultante. Por lo tanto, el procesamiento de cada elemento en la matriz de entrada producirá como clave los índices del elemento de la matriz de salida que requiera de este valor para obtener su resultado, esto es, el total de índices que no tenga relación entre matrices a multiplicar. Como valor se adjuntará el valor del elemento y el índice que tenga relación con la matriz multiplicadora. En la segunda etapa, Reduce, cada instancia recibe una clave, es decir los índices del elemento de salida y como valores la lista de elementos que requiere para su cálculo. Como procesamiento, se multiplica cada elemento con su correspondiente y se realiza la sumatoria de tales productos.

El siguiente algoritmo para esta implementación realiza el procesamiento por cada elemento de cada matriz.

```
class Mapper
  method Map (Key line-offset, Value Flag,r,c,element_rc)
    Get totalOfRow&Columns of A & B
    if element belongs to A
      for each column of B
        Emit (asKey(i,k), asValue[j,a_ij])
    if element belongs to B
      for each row of A
        Emit (asKey(i,k), asValue[jb_jk])
```

Figura 45-Pseudocódigo Map para Multiplicación de Matrices. §§§

El formato de la matriz de entrada está formado tomando en cuenta que cada línea contiene un elemento de la matriz. El siguiente elemento, en la línea subsecuente, corresponde al mismo renglón pero posterior columna. Los valores por línea, separados por comas "," son; etiqueta de la matriz, índice de renglón, índice de columna y valor del elemento.

Para la etapa Reduce el algoritmo a utilizar sería de la siguiente forma.

```
class Reducer
  method Reduce (Key (i,k), Value List[j,elements])
    Get totalOfRow&Columns of A & B
    sum ← 0
    for each (element : elements)
      if element is stored
        element(j) = element(j) * element
        sum ← sum + element(j)
      if element is not stored
        element(j) = element
    next element
```

Figura 46- Pseudocódigo Reduce para Multiplicación de Matrices.

Se observa que en esta etapa Reduce se utiliza de un arreglo de una dimensión llamado *element()* en donde se almacenan los elementos que corresponden a esta *clave*. Por medio de condicionales *if* se determina si sólo se asigna el valor al arreglo, para el caso de los elementos que no hayan sido almacenados, o si se realiza la multiplicación y adición si el elemento ya fue almacenado.

§§§ Los valores de Flag y element puede ser A ó B y a ó b, respectivamente, dependiendo de la matriz que se esté procesando. *r* y *c* representan los índices del elemento que corresponden a *i, j ,k* dependiendo de la matriz a la que pertenezca el elemento en procesamiento.

Bibliografía

Achari, S. (2015). *Hadoop Essentials.* Packt Publishing.

Apache Flink®. (2023). Retrieved Julio 2023, from Stateful Computations over Data Streams | Apache Flink: https://flink.apache.org/

Dean, J., & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters.*

Gholamalinejad, H., & Khosravi, H. (2020). *Pooling Methods in Deep Neural Networks, a Review.* arXiv preprint arXiv:2009.07485.

Ghosh, S. (2014). *Distributed Systems: An Algorithmic Approach.* Florida, US: Taylor And Francis.

Han, J., Pei, J., & Tong, H. (2022). *Data Mining: Concepts and Techniques.* Cambridge, Massachusetts, US: Morgan Kaufmann.

Ishwarappa, & Anuradha, J. (2015). A Brief Introduction on Big Data 5Vs Characteristics and Hadoop Technology. *ICCC-2015.* Odisha, India.

Jules S. Damji, B. W. (2020). *Learning Spark.* O'Reilly Media.

Leskovec, J., Rajaraman, A., & Ullman, J. D. (2019). Matrix Multiplication. In *Miningx of Massive Datasets.* Cambridge University Press.

Li, J., Wu, ,. J., Zhong, S., & Yang, X. (2015). Optimizing MapReduce Based on Locality of K-V Pairs and Overlap between Shuffle and Local Reduce. *44th International Conference on Parallel Processing.* Beijing China: IEEE Computer Society 2015.

Lin, J., & Dye, C. (2010). *Data-Intensive Text Processing with MapReduce.* Morgan & Claypool Publishers.

Naeem, M., Tauseef, J., Diaz Martinez, J., Aziz Butt, S., Montesano, N., Imran Tariq, M., et al. (2019). Trends and Future Perspective Challenges in Big Data. *Intelligent Data Analysis and Applications.* Arad, Romania: Springer.

Özdemir, C. (2023). Avg-topk: A new pooling method for convolutional neural n etworks. *Expert Systems with Applications* .

Pacheco, P. (2011). *An Introduction to Parallel Programming.* Elsevier Science.

Qasem, M. H., Sarhan, A. A., Qaddoura, R., & Mahafzah, B. A. (2017). Matrix multiplication of big data using MapReduce: A review. *IT-DREPS Conference, Amman, Jordan.* Amman, Jordan.

Ryzko, D. (2020). *Modern Big Data Architectures, A Multi-Agent Systems Perspective.* Wiley.

Shi, Y. (2022). *A Brief Introduction on Big Data 5Vs Characteristics and Hadoop Technology.* Beijing, China; Omaha, Nebraska: Springer Nature Singapore.

Thomas H. Cormen, C. E. (2009). *Introduction to Algorithms.* MIT Press.

Trujillo, G., Kim, C., Jones, S., Garcia, R., & Murray, J. (2015). *Virtualizing Hadoop. How to Install, Deploy, and Optimize Hadoop in a Virtualized Architecture.* Pearson Education.

Wang, J., Xu, C., Zhang, J., & Zhong, R. (2022). *Big data analytics for intelligent manufacturing systems: A Review.* Shanghai.

wang, y., cui, z., & ke, r. (2023). *Machine Learning for Transportation Research and Applications.* Elsevier.

White, T. (2015). *Hadoop The Definitive Guide.* O'Reilly Media.

Wilkinson, B., & Allen, M. (2004). *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers.* Pearson/Prentice Hall.