
Universidad Nacional de Luján

LICENCIATURA EN SISTEMAS DE INFORMACIÓN

TRABAJO FINAL DE LICENCIATURA

# CONSTRUCCIÓN DE ÍNDICES PARA DATOS MASIVOS

---

**Autor:** Pablo Tomas Delvechio
**Director:** Gabriel Tolosa

Año 2017

# Agradecimientos

Es muy difícil agradecer a todas las personas que influyen positivamente en la formación personal y profesional sin cometer olvidos u omisiones. Sin embargo considero que es peor ser desagradecido, por lo que a continuación va una lista de personas que influyeron en mi y en la realización de este trabajo.

En primer lugar quiero agradecer a mi Director, Gabriel Tolosa, por su constante guía en mi formación académica. Este trabajo no estaría terminado sin sus consejos y disposición permanente.

En segundo lugar un agradecimiento a todo el equipo del CIDETIC, en especial a Alejandro Iglesias y Francisco Tonin Monzón que siempre estuvieron dispuestos a colaborar con la construcción y mantenimiento del cluster Hadoop, atendiendo pedidos diversos derivados de mi trabajo.

También quiero agradecer a colegas y compañeros que estuvieron aconsejando y apoyando, en especial a Santiago Banchero, Esteban Rissola y Mauro Meloni.

Un agradecimiento especial a Yamila, quien en estos últimos meses fue un apoyo decisivo para que yo tuviera las ganas y fuerzas de sentarme a escribir este trabajo. Gracias por todas esas tardes de mate y paciencia.

El apoyo de mi familia me acompañó en mi formación académica y también en la construcción de mis valores éticos y humanos. Agradezco a mis padres Monica y Fredy como a mis hermanos por haber servido de guía. Además quiero mencionar a Nelly y Pilar, que ya no están acompañándome pero siempre están en mis recuerdos, y por ultimo a Ramiro, que llego a nuestras vidas hace poco con inocencia y alegría.

# Índice general

# Capítulo 1

# Introducción

En la actualidad, las organizaciones de todo tipo y tamaño tienen a su disposición grandes volúmenes de información a muy bajo costo. Por ejemplo, la web es un repositorio de documentos que tiene una escala sin precedente y con alcance mundial. Asimismo, las organizaciones aumentan su capacidad de generar datos y, por consiguiente, una necesidad intrínseca de almacenarlos. Algunas estimaciones indican que los conjuntos de datos crecen exponencialmente [34]. Los datos almacenados (analógicos y digitales) suman 2,6 exabytes para el año 1986, 15,8 en 1993, 54,5 en el 2000 y 295 hacia el 2007. Dentro de estos valores, el 99 % de los datos en 1986 eran analógicos. Para 2007, los datos digitales conforman el 94 % de los 295 exabytes estimados [22].

Como un ejemplo ilustrativo, durante el primer Simposio Argentino de Grandes Datos[1], se muestra como el uso del Sistema SUBE por parte de usuarios de colectivos del AMBA (Área Metropolitana de Buenos Aires) genera un conjunto de datos que no puede ser recolectado por otros medios (como encuestas). Un estudio presentado en dicho simposio reporta que 3 años de datos de una empresa del sector que cuenta con el 1 % de los colectivos del sistema, reúne 40 millones de boletos vendidos y 150 millones de posiciones de GPS [37].

Realizar operaciones de procesamiento y recuperación de información en este contexto no es una tarea trivial y los motores de búsqueda comerciales no tienen alcance en el ámbito interno de las organizaciones. La industria ofrece una plataforma conocida como computación en la nube (también referida como Cloud Computing o "la nube") como una forma de abordar el problema de la escalabilidad de los procesos, del almacenamiento y sus costos asociados[2]. Sin embargo, esta opción no es viable en todos los casos (por ejemplo, por problemas económicos o legales respecto de los datos a manipu-

[1] `http://44jaiio.sadio.org.ar/agranda`

[2] `https://www.wired.com/2015/03/amazon-unlimited-everything-cloud-storage/`

lar). Por eso, las técnicas de recuperación de información de gran escala son de alto interés para la industria y la academia.

La manipulación de datos a escala masiva también genera que las técnicas tradicionales de procesamiento y búsqueda de información no sean tan eficientes como podían serlo hace una década o menos. Es necesario que los procesos de recuperación de información clásicos se adapten a esta nueva escala y evolucionen conforme crece la cantidad de datos disponibles (es decir, generar algoritmos escalables). Una opción explorada en la disciplina es la elaboración de algoritmos de cómputo distribuido junto con plataformas de ejecución de alto nivel.

El presente trabajo surge como necesidad de integrar dos áreas: La Computación Distribuida y la Recuperación de Información. La integración de estas áreas no es, ni por lejos, algo novedoso. Existe amplia literatura al respecto que data de varias décadas [35] [11]. La industria también posee algoritmos y servicios que se pueden considerar maduros en este ámbito. Sin embargo, la amplitud de métodos, herramientas y técnicas proporciona siempre nuevas ideas para explorar enfoques que permiten elaborar implementaciones alternativas que pueden resultar de interés. A lo largo de este capítulo introductorio se establece el contexto general asumido y se explican las motivaciones y objetivos del trabajo.

## 1.1. El Problema de la Recuperación de Información

La Recuperación de Información (abreviado como RI) es una disciplina que tiene su origen en la necesidad de contar con herramientas que permitan el acceso a documentos digitales de naturaleza no estructurada, mediante alguna interfaz ofrecida al usuario para que este pueda expresar sus necesidades de información en un lenguaje de consultas (*queries*) con una sintaxis determinada. Una opción, entre otras que existen [50], son las consultas de texto libre, populares en los modernos motores de búsqueda comerciales.

Baeza-Yates [4] plantea que la Recuperación de Información trata con la representación, almacenamiento, organización y acceso a ítems de información. Es un área que abarca desde la recuperación de documentos hasta la forma en que el usuario accede a los mismos, pasando en el medio por muchos procesos para que esto suceda de forma adecuada:

- Recuperación y normalización de los documentos.
- Generación de estructuras auxiliares que soporten la recuperación.

- Cálculo de rankings de documentos.
- Visualización de respuestas de consultas de usuario.
- Compresión de datos.

El proceso de la recuperación de información tiene varias etapas, que se pueden resumir a grandes rasgos en los siguientes pasos:

- El Usuario envía una consulta al motor de búsqueda a través de una interfaz, utilizando un lenguaje de consulta.
- El motor de búsqueda recibe la consulta, la evalúa y utiliza las estructuras que tiene disponibles para definir cuales son los documentos que resultan relevantes para responder la solicitud de información.
- Una vez definidos los documentos a ser recuperados, el motor genera un ranking en base a la información disponible, y devuelve el listado de documentos al Usuario.

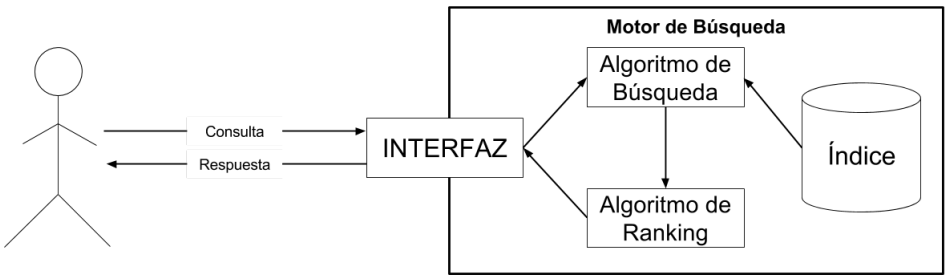



Figura 1.1: El Proceso de la Recuperación de Información

Para que los pasos anteriores funcionen eficiente y eficazmente, el motor de búsqueda debe contar con un corpus o colección de documentos que resulten de interés para los Usuarios. También es necesario realizar algunos procesos que tienen por objetivo construir estructuras auxiliares para ser utilizadas durante la recuperación. Este proceso de creación de las estructuras es previo a la resolución de consulta y consta de los siguientes pasos:

- Recuperar o definir la colección de documentos.

- Ejecutar un proceso que recorra la colección y genere las estructuras de datos que dan soporte a la recuperación.

La estructura mas popular y que a mostrado ser mas flexible en los sistemas de RI de propósito general es el índice invertido [4] [50] [10]. El proceso de generación de dicho índice se conoce como indexación o construcción del índice, y es el caso de estudio objetivo utilizado para implementar los algoritmos y medir el comportamiento de los mismos en este trabajo.

### 1.1.1. La Necesidad de la construcción de Índices

Un índice invertido es una estructura de datos que permite obtener un listado de documentos a partir de un conjunto de términos dado [4]. Los usuarios de los motores de búsqueda proporcionan términos al sistema para resolver sus necesidades de información. De esta manera, los índices invertidos permiten resolver las consultas de los usuarios a partir de los términos de las mismas haciendo uso del mencionado índice [51].

En la literatura pueden encontrarse diferencias entre el concepto de índice invertido y archivo invertido [33]. En el contexto de este trabajo, se hablara de índice invertido.

Hace décadas se estableció que utilizar un índice para resolver consultas en motores de búsqueda mejora en varios ordenes de magnitud la eficiencia de la búsqueda de información manual por parte del usuario [16].

Witten explica que "El objetivo de un índice es resolver el problema de como la información debe estar organizada para que las consultas sean resueltas de forma eficiente y las porciones relevantes de los datos sean rápidamente extraídas" [50]. El índice invertido es la estructura que permite esto.

Si bien la construcción de un índice tiene costos de tiempo y almacenamiento asociados, también resulta claro que en determinados contextos, dicho costo se compensa en función de las mejoras que proporciona el mismo para la velocidad de procesamiento de las consultas en un motor de búsqueda [4]. Esto es válido para colecciones de documentos que sobrepasen un tamaño determinado. Además, en relación al procesamiento secuencial del índice frente a cada consulta, la construcción y el mantenimiento del mismo se convierte en una tarea de bajo costo [4], y en si mismo, es de complejidad baja. De hecho, en la literatura se establece que el proceso de indexado es un problema cuyas principales restricciones están asociadas al hardware disponible en los equipos que construyen el índice [33]. Resulta claro entonces que cuando el tamaño de la colección sobrepasa los recursos disponibles en un único equipo (en general, memoria principal y CPU), se requiere cambiar el enfoque con el cual el problema se aborda.

## 1.2. Big Data

El imparable crecimiento de datos no estructurados en los últimos años plantea el desafío de disponer de esquemas de procesamiento masivos a costos razonables y con prestaciones adecuadas. El sitio World Wide Web Size[3] reporta una estimación del índice de Google de entre 46 y 47 miles de millones de páginas web.

El crecimiento de los datos no es exclusivo del ámbito de la Web. La compañía Farecast predecía para el 2008 el precio de los boletos de aerolíneas y ofrecía dichos servicios a los consumidores basándose en la información de los vuelos comerciales adquiridos a las mismas empresas de vuelo. En un año procesaba cerca de doscientos mil millones de registros. El fundador afirmaba que diez años antes hubiera sido imposible ofrecer este tipo de servicios, debido tanto a la capacidad de almacenamiento y procesamiento disponible como al enfoque en el uso de los registros transaccionales que existía en las organizaciones [44].

Soluciones especificas para llevar a cabo el procesamiento de este tipo de información pueden servir durante una etapa inicial o exploratoria, pero cuando el volumen de información supera los límites estimados de forma continua, se plantea un compromiso entre la cantidad de datos disponibles y el volumen que efectivamente puede ser procesado. Esto esta limitado principalmente por la infraestructura disponible y los algoritmos seleccionados.

Una de las áreas donde la cantidad de datos no estructurados crece de forma acelerada es la recuperación de información relacionada con la web. En consecuencia, los procesos de RI se adaptaron para que trabajen de forma masivamente distribuida. MapReduce [11] es una respuesta a esto desde hace más de una década introduciéndose, de forma progresiva y conjunta, el término Big Data, para hacer referencia a este tipo de problemas, donde la cantidad de datos disponibles obliga a cambiar el enfoque y los algoritmos utilizados.

La Unión Internacional de Telecomunicaciones (ITU) define Big Data como "Un paradigma para permitir la recolección, almacenamiento, gestión, análisis y visualización de conjuntos de datos masivos, de características heterogéneas, bajo restricciones potenciales de tiempo real" [1].

Dentro del área de Big Data, MapReduce es considerado central [39]. El software que implementa dicho modelo y que mayor adopción ha tenido por parte de la industria y la academia es Hadoop[4]. MapReduce permite el

[3] `http://www.worldwidewebsize.com/` - Consultado en abril de 2017

[4] `https://wiki.apache.org/hadoop/PoweredBy`. Compañías como Facebook y Yahoo aportan desarrolladores activamente al proyecto o proyectos relacionados, mientras que otras grandes organizaciones reportan tener clusters sobre los que ejecutan trabajos

tratamiento de grandes cantidades de datos en clusters de hardware dedicado y "económico" (*commodity hardware*), haciendo uso de conceptos derivados de la programación funcional (map y reduce) y estrategias de paralelización y distribución de carga.

En el contexto de este trabajo, se utilizan herramientas y algoritmos usados en el ámbito de Big Data para resolver un problema de construcción de índices invertidos para motores de búsqueda. La combinación de estas dos áreas ya fue explorada previamente [11] [31] [10]. Sin embargo, el foco de este trabajo se encuentra en medir la escalabilidad de diferentes soluciones, variando parámetros tanto de los datos a procesar como de la plataforma. Además, en esta propuesta se asume y utiliza hardware económico usualmente encontrado en cualquier organización, diferenciándose significativamente de la utilizada en la gran mayoría de la literatura. Cabe destacar que al momento de llevar a cabo este trabajo, no se tiene información de que dicho enfoque exista.

### 1.2.1. Commodity Hardware en un Contexto de Limitaciones

El presente trabajo utiliza para los experimentos un cluster[5] de equipos con especificaciones de hardware para usuarios de escritorio, conocidas como computadores personales o PC (por ejemplo, procesador Intel Core i5, 8 GB de memoria RAM y 400 GB de disco rígido). Este tipo de equipamiento es referido como commodity hardware.

Buscar la escalabilidad horizontal de la infraestructura no es algo original. Debido al desarrollo de la industria de hardware, es mas optimo en relación al costo tener muchos equipos pequeños y económicos unidos por una red, que construir grandes supercomputadoras. Esta tendencia se ve favorecida, entre otros aspectos, por el hecho de que los insumos para redes de alta velocidad son cada vez mas accesibles, y por el contrario, optimizar otros componentes, como el acceso a la memoria RAM en un mismo equipo, es cada vez mas costoso [14].

---

en Hadoop. Entre dichas organizaciones se puede encontrar a Adobe, Ebay, IBM y PARC entre otras

[5]Un cluster es una configuración de varios equipos que trabajan de forma coordinada a través de una red de datos. Cada equipo dentro del cluster es conocido como nodo. Un cluster suele ser construido como alternativa económica a utilizar equipos de super computo o super computadoras, donde la escalabilidad se realiza de forma vertical (es decir, se agrega mas cantidad de recursos de hardware al equipamiento disponible). Por el contrario, un cluster escala de forma horizontal. Esto significa que no se trata de agregar recursos a cada uno de los elementos del cluster, sino agregar mayor cantidad de nodos.

Es habitual en la literatura encontrar experiencias con clusters de características técnicas superiores a las descriptas anteriormente, ya sea por cantidad de nodos o hardware disponible por nodo. Por ejemplo, hardware de especificaciones mas modestas a nivel de cada nodo (2 GB de RAM y 300 GB de almacenamiento), pero disponibilidad de 80 nodos [27]. Estas y otras configuraciones similares son habituales en la bibliografía que desarrolla temas de Big Data [32].

También existe el enfoque de usar servicios basados en la nube, donde frente a la problemática de los límites de recursos, la solución es contratar mayor cantidad del recurso que cause el cuello de botella [24]. En el caso de Gunther et. al [19], se reportan dos configuraciones para el estudio en cuestión. La primera configuración posee gran cantidad de memoria RAM (34,2 GB) pero poco espacio de almacenamiento y latencia de red moderada, y 4 CPUs para procesamiento. La segunda instancia está optimizada para cómputo (8 CPUs), posee baja cantidad de memoria principal (7 GB en total), gran cantidad de espacio de almacenamiento y baja latencia de red.

Las configuraciones descriptas anteriormente son completamente válidas y no se intenta expresar algo en otro sentido. Sin embargo, el punto de este trabajo consiste en estudiar un contexto donde escalar vertical u horizontalmente es complejo o directamente imposible. Frente a las limitaciones de almacenamiento, memoria o procesamiento, se opta por optimizar las configuraciones o algoritmos antes de elegir alternativas que impliquen incorporación de mayor cantidad de hardware. De esta manera, se considera que la plataforma es aprovechada de forma mas eficiente. Esto es opuesto a la tendencia clásica donde el enfoque de desarrollar sobre MapReduce, al ser simple y "barato" [35], genera soluciones sub-optimas porque al alcanzar los limites, se soluciona escalando horizontalmente.

Existen varios motivos por los cuales utilizar MapReduce es económico y sencillo frente otras alternativas (como clusters de "High Performance Computing" o HPC por ejemplo). Estos son algunos de ellos [25] [49]:

- Almacenamiento: Los nodos de un cluster para HPC suelen ser exclusivamente para computo, y la organización dispone de algún tipo de plataforma de almacenamiento de alta disponibilidad y con una red de altas velocidades que es un cuello de botella al transmitir los datos a cada uno de los nodos. Hadoop utiliza un concepto de localidad de datos, y consiste en que los nodos de computo son parte a su vez del almacenamiento del cluster, y esto genera que no se deba tener un servidor de almacenamiento dedicado.

- Interfaz de control de flujo de datos: MPI es muy flexible, y delega en el desarrollador el control del flujo de datos de la aplicación distribuida.

MapReduce propone un esquema de flujo de datos predefinido, una representación uniforme de los mismos para el desarrollador y esto hace que sea menos flexible pero mas sencillo de desarrollar, al no tener que ocuparse de dicho flujo.

- Coordinación de tareas: El desarrollador de aplicaciones para HPC debe tener un control y conocimiento de como se coordinan las mismas. Un desarrollador de aplicaciones MapReduce no necesita entender esta coordinación, y en ningún caso debe implementarla o lidiar con su complejidad para hacer aplicaciones funcionales sobre Hadoop (Planificación, reserva de recursos, despliegue, recuperación ante errores), porque es responsabilidad del "framework". Solo debe implementar las interfaces necesarias que son sencillas de comprender. Esto, que es un modelo de desarrollo muy restringido [49] es a su vez una de las ventajas de la plataforma.

Los puntos anteriores muestran que los conocimientos de la plataforma de un desarrollador de aplicaciones MapReduce puede ser muy inferior a sus pares de HPC. El efecto de esto es que resulta mas económico encontrar o entrenar desarrolladores para Hadoop que para HPC. Resulta claro que un efecto negativo es que tener desarrolladores con un conocimiento superficial de la plataforma puede derivar en muchos casos en soluciones sub-optimas. A tal punto esto afecta al ecosistema, que existe toda un área de trabajo en propuestas de soluciones de optimización de tareas sobre Hadoop [26].

White [49] hace diferencia entre hardware "commodity" y "low-end". La definición de "commodity" para White es referida a equipos con 2 procesadores octo-core, memoria principal con tamaños entre 64 y 512Gb y 12 a 24 discos de 1 a 4Tb cada uno.

La infraestructura utilizada el presente trabajo se asemeja mas a la definida como "low-end", que son equipos construidos con componentes baratos.

## 1.3. Motivaciones del Trabajo

Este trabajo tiene varias motivaciones, que combinan temas clásicos (como recuperación de información) con temas más novedosos (infraestructura de Big Data). A nivel académico, resulta motivador la exploración de una disciplina en pleno auge como lo es Big Data, y su combinación con la Recuperación de Información.

A nivel institucional, es una deuda empezar a generar conocimiento y experiencia en las herramientas y técnicas de Big Data, así como construir

un cluster y utilizarlo de manera regular para investigación, y que permita medir las capacidades y limitaciones del mismo.

La mayor parte de la literatura consultada que utiliza Hadoop para construir índices invertidos, suele tomar en cuenta de forma marginal el problema de la escalabilidad de dichos algoritmos. Por ejemplo, se optimiza de forma sencilla el código base para no sobrecargar la entrada/salida de almacenamiento secundario o red [35]. Hay casos donde se implementan patrones de diseño de codigo MapReduce que representan una mejora tangible en el flujo de información, pero no se proveen mediciones de dichas implementaciones [31]. En este trabajo se busca implementar un índice simple (tomado como baseline) y dos índices avanzados sobre Hadoop utilizando el framework MapReduce.

## 1.4. Objetivos del Trabajo

El presente trabajo persigue una diversidad de objetivos. Los principales están vinculados a la investigación académica. También hay un conjunto de objetivos secundarios que plantean la generación y difusión de los conocimientos de este trabajo dentro de la Universidad, en el contexto de la carrera Licenciatura en Sistemas de Información y se plantea transferir parte del conocimiento adquirido como infraestructura aprovechable por parte de grupos académicos y de investigación.

### 1.4.1. Objetivos Principales

Como se estableció, un conjunto de objetivos principales son buscados en este trabajo, a saber:

- Implementar algoritmos de indexación en un entorno distribuido de hardware económico y prestaciones limitadas.
- Probar el comportamiento de un cluster con plataformas usadas en Big Data en tareas intensivas de creación de índices.
- Medir la eficiencia de un algoritmo diseñado para MapReduce para procesar una colección de documentos con diferentes configuraciones de la plataforma.

### 1.4.2. Objetivos Secundarios

Los objetivos secundarios surgen a raíz de no contar con experiencia en Hadoop ni en tecnologías de Big Data, ni con la infraestructura necesaria

para las pruebas. Se plantea poder construir un servicio basado en Hadoop, configurarlo adecuadamente, y dejarlo a disposición de la comunidad de investigación luego de finalizar el trabajo.

- Construcción de un cluster Hadoop para desarrollar las pruebas con PCs convencionales.
- Establecer una configuración adecuada para el uso de recursos en la plataforma Hadoop. La complejidad de un sistema distribuido genera que la mejor optimización se logre mediante mediciones experimentales para determinar la configuración adecuada en base al hardware disponible y los procesos ejecutados.

## 1.5. Estructura del Documento

Este trabajo se encuentra organizado en capítulos. Estos capítulos están pensados para ordenar la exposición y darle un sentido. Los temas se abordan de forma iterativa. Es decir, se plantean de forma general. En la medida que se avanza en cada uno de ellos, los mismos son profundizados y desarrollados.

El capítulo 2 introduce las lineas teóricas y practicas sobre la que se soporta el trabajo. Aborda y explica en profundidad la Recuperación de Información y la construcción de índices como forma de organización de los motores de búsqueda. Explica ademas técnicas utilizadas para la compresión de datos, y se dan detalles generales de su uso dentro de las implementaciones realizadas para este trabajo. También se aborda la problemática de Big Data y se dan detalles del funcionamiento general de Hadoop.

El capítulo 3 describe en profundidad las implementaciones de constructores de índices implementados. Se realiza primero una explicación conceptual y al final se explica el formato físico de almacenamiento de dichos índices. Ademas, se describe como se utilizo la plataforma y Hadoop para que la implementación de los índices sea construida en base a la lógica de procesamiento de la plataforma y no hacer un uso forzado o poco practico de la misma.

El capítulo 4 describe en profundidad las pruebas realizadas. Define la infraestructura construida y las configuraciones y optimizaciones realizadas al mismo. Describe y caracteriza las colecciones utilizadas para las pruebas. Y por ultimo define los experimentos que se realizaron, la entrada utilizada, los parámetros usados y los resultados medidos. Luego expone de forma exhaustiva los resultados. Primero se explican que criterios fueron utilizados para analizar los resultados, y luego, por cada experimento realizado, se exponen las métricas de forma detallada y se explican en el contexto del

trabajo, se analiza si corresponde con las hipótesis de trabajo y se compara con los objetivos propuestas para corroborar su cumplimiento o no.

Al final, el capítulo de conclusiones 5 aborda el análisis final sobre el cumplimiento de los objetivos del trabajo.

# Capítulo 2

# Preliminares y Trabajos Relacionados

A lo largo de toda esta sección se fundamenta el marco teórico del trabajo, se desarrollan las disciplinas mencionadas y se introducen los conceptos tenidos en cuenta al adoptar las decisiones de diseño de los algoritmos. Se comienza este capítulo haciendo un desarrollo de la evolución del concepto de Big Data (Sección 2.1) y su alcance en la academia y el mundo actual, para luego pasar a detallar la plataforma de software sobre la cual se implementan estos procesos, Hadoop (Sección 2.1.1). Para ello se abordan las ideas principales y conceptos del motor de procesamiento de Hadoop, conocido como MapReduce y como se relaciona con el servicio de almacenamiento distribuido provisto por Hadoop conocido como HDFS.

A continuación se realiza una introducción a las ideas centrales detrás de la recuperación de información (IR) (Sección 2.2). Se aborda aquí además la forma en que dentro de la IR se realiza la construcción de índices, para luego discutir que efectos tiene la necesidad de distribuir esta construcción en un cluster. Se explica a continuación que características tienen 2 índices invertidos particulares, que define a cada uno y que conceptos en común tienen entre ellos. Finalmente, se desarrollan algunas ideas sobre compresión de índices y se presentan algunos algoritmos de compresión que son utilizados en el área de IR (Sección 2.3).

## 2.1. El Enfoque de Big Data

Con la difusión de uso comercial, industrial y hogareño de dispositivos conectados a Internet, la recopilación de todo tipo de datos (de negocios, transaccionales, de registros o logs, datos personales, entre otros) crecen de

forma creciente y acelerada [1]. La consecuencia de este crecimiento es la dificultad de seguir manteniendo las mismas técnicas de procesamiento y almacenamiento para esta nueva escala de datos.

Los problemas tratados por las investigaciones sobre Big Data no son nuevos. La asimetría entre la capacidad de generación de datos y su consumo (ya sea académico o comercial) es tópico de investigación desde mediados del siglo XX[1]. Por otro lado, el término Big Data es mas reciente. Una de las primeras referencias es encontrada en Cox y Ellsworth [9] en 1997 donde Big Data hace referencia a aquellos conjuntos de datos que por su tamaño no pueden ser cargados en la memoria principal o incluso el disco de un único dispositivo de computo.

Si bien en sus inicios este término es utilizado como se menciona anteriormente por Cox [9], hoy en día el concepto es mas amplio, e incluye no solo a los "datasets"[2] de escalas de los exabytes o incluso Internet como fuente de datos, sino a los ecosistemas, infraestructuras, herramientas y soluciones que se enfocan en abordar el tratamiento de dicha información.

El término Big Data se vuelve popular a finales de la primer década del siglo XXI, cuando diversos miembros de la industria comienzan a ofrecer servicios promocionados específicamente para estas necesidades, como capacidad de procesamiento de grandes cantidades de datos, o tecnología y equipamiento para el mismo propósito. Para lograr esto se aprovecha la tendencia previa del "Cloud Computing", que ofrece un entorno favorable para ofrecer servicios de este tipo. Para esta fecha, software como Hadoop[3] o servicios como Amazon EMR[4] ya cuentan con varios años en el mercado.

Cuando hoy en día los actores mas importantes de la industria IT hablan de la escala de datos procesada por sus algoritmos de Big Data, en general suelen referirse al conjunto de datos disponible en Internet. Según estimaciones de IBM[5], para el año 2014, cada día se producen 2,3 trillones de GB de datos en Internet, y la estimación para el año 2020 es que existan 40 Zettabytes de datos en la web.

Dentro de las problemáticas a las cuales se enfrentan las tecnologías de Big Data, interesa destacar la siguiente: Generalmente los datos recuperados de la

[1] http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/

[2] Dataset o conjunto de datos, es la denominación asignada a la colección de información a la cual se hace referencia y se ejecutan los algoritmos implementados. Por ejemplo, en el contexto de Internet, el conjunto de datos es toda pagina web que se pueda recuperar y analizar.

[3] http://hadoop.apache.org/

[4] http://aws.amazon.com/elasticmapreduce/

[5] http://www-01.ibm.com/software/data/bigdata/

web se pueden describir en su mayoría como datos no estructurados. Mientras que las herramientas creadas para tratar grandes cantidades de datos sirven con ligeros cambios para manipular datos estructurados, el gran crecimiento de datos en la web es de datos no estructurados, y esto plantea un desafío adicional a los desarrolladores [20]. En un gran numero de organizaciones surgen a diario necesidades de procesamiento de volúmenes de información de estas características. Es por eso que muchos miembros de la industria y la academia pueden verse beneficiados por las herramientas y modelos que se vienen desarrollando en los últimos años bajo la denominación de algoritmos para Big Data.

Dicho lo anterior, surge naturalmente una pregunta que interesa responder: ¿cual es la motivación para guardar grandes cantidades de datos? ¿Porque conservar datos que en el pasado se consideraron poco útiles al expirar su tiempo de vida? Se detallan una serie de puntos que pueden ayudar a responder parcialmente estos interrogantes.

- Necesidad de generar modelos históricos para realizar estadísticas, predicciones o trabajos científicos (Clima, Decodificación de ADN).
- Requerimientos legales (registros o "logs" de llamadas durante un periodo de tiempo para operadoras telefónicas).
- Datos Históricos de tratamientos médicos y drogas suministrados a pacientes.
- Mejorar experiencia de usuarios mediante estudio de datos de comportamiento, consumo, etc.

La mayoría de los casos enumerados anteriormente tienen por objetivo extraer información valiosa de datos "viejos" o presuntamente ya "obsoletos". Todo esto es acompañado por el abaratamiento de costos en dispositivos de almacenamiento secundario y componentes de redes de alta velocidad [5].

Es necesario establecer criterios para determinar cuando se esta en un contexto que requiere soluciones desde la perspectiva de los datos masivos. Para caracterizar este entorno, la Unión Internacional de Telecomunicaciones o *ITU* retoma una definición popular conocida como las 3V del Big Data [1]:

- Volumen: Refiere a la cantidad de datos recolectados, analizados y visualizados.
- Variedad: Hace referencia a las diferentes fuentes y formatos de los datos utilizados.

- Velocidad: Esta relacionado al marco de tiempo en el que los datos son recuperados y procesados.

Existen dos criterios adicionales que se utilizan para caracterizar el enfoque de Big Data [1]:

- Veracidad: Valor de certeza o confianza asociado a los datos manipulados.
- Valor: Potencial de negocio del resultado de analizar los datos disponibles.

El estándar ITU plantea asimismo los desafíos que son abordados por el ecosistema de Big Data [1]:

- Heterogeneidad e incompletitud de los datos.
- Escala creciente de la generación de datos y capacidades de procesamiento.
- Temporalidad de los datos. Se trata de establecer una linea de tiempo pero también de acotar los análisis a periodos determinados.
- Privacidad de los datos procesados.

En este trabajo se propone usar las herramientas y métodos creados en esta área para explorar como el área de recuperación de información en pequeñas organizaciones puede beneficiarse con estas nuevas herramientas de procesamiento masivo. Para ello, primero se debe entender que ofrece el ecosistema y que modelo de procesamiento tiene asociado.

### 2.1.1. Hadoop

Hadoop es una plataforma para desarrollo de sistemas distribuidos que tiene el objetivo de procesar datos de forma masiva [28]. Su intención es abstraer la complejidad asociada a la utilización de un "cluster" de equipos mediante el uso de modelos de programación predefinidos. Utilizar los métodos que propone el entorno permite desarrollar aplicaciones que procesan grandes cantidades de información sin la necesidad de tener gran experiencia en programación de aplicación distribuidas [28] [25]. Esto es posible gracias a que Hadoop abstrae completamente la gestión de la infraestructura subyacente.

Cuando se utiliza el término Hadoop, pueden referirse a 2 conceptos relacionados:

- El núcleo de Hadoop, que consta de dos servicios y un modelo de programación.

- El ecosistema de herramientas y servicios que se construyen sobre el núcleo de Hadoop.

El ecosistema Hadoop se enfoca en herramientas y servicios que permiten a los desarrolladores manejar grandes conjuntos de datos de manera cómoda. Los servicios y herramientas hacen referencia al software que viene de forma nativa cuando se descarga Hadoop de su pagina oficial[6]. Hadoop ofrece un servicio de almacenamiento de datos distribuido, conocido como HDFS[7], y un servicio de planificación y procesamiento, conocido como YARN[8]. Usando la API[9] de HDFS es posible acceder a los datos almacenados y generar nuevos conjuntos de datos. Usando la API de YARN es posible desarrollar flujos de trabajo y construir las aplicaciones sobre Hadoop.

Hadoop ofrece un modelo de desarrollo implementado por defecto, el cual es conocido como MapReduce. Haciendo uso de YARN y HDFS, se pueden construir nuevos servicios para operaciones mas especializadas. HBase[10], Spark[11] o Storm[12] son ejemplos de servicios que se construyen utilizando la API de YARN, dado que se enfocan en operaciones que son excesivamente complejas de implementar haciendo uso de MapReduce exclusivamente.

La plataforma Hadoop ofrece múltiples puntos de ventaja sobre otras opciones para la construcción de sistemas distribuidos. Un listado no exhaustivo de los mismos es el siguiente:

- Infraestructura: Hadoop permite ser ejecutado sobre un cluster de equipos de recursos limitados, o de hardware económico. Esto es posible porque los servicios básicos de la plataforma no requieren recursos excesivos para su ejecución y funcionamiento, y el consumo de recursos para los algoritmos implementados es ajustable vía configuración. Esto permite a una organización pequeña disponer de un cluster de prestaciones modestas a un costo accesible, donde los equipos no necesariamente deben ser servidores, sino que pueden ser equipos de usuarios

[6] `http://hadoop.apache.org/`

[7] HDFS es un acrónimo de "Hadoop Distributed File System".

[8] YARN es un acrónimo de "Yet Another Resource Negotiator"

[9] API es un acrónimo de "Application Programming Interface", y refiere al conjunto de métodos, funciones y procedimientos para tomar servicios de una librería o proceso remoto.

[10] `https://hbase.apache.org/`

[11] `http://spark.apache.org/`

[12] `http://storm.apache.org/`

finales destinados a tal efecto y no requiere que sean de un fabricante particular [49]. Esto es posible porque una de las características de los servicios en Hadoop es que escalan horizontalmente [30].

- Implementación Cloud: Hadoop fue concebido para procesamiento intensivo por parte de múltiples usuarios que compiten por los recursos disponibles en el cluster. El framework puede manejar colas de procesamiento con prioridades, asignar porcentajes de recursos a diferentes usuarios, entre algunas de las capacidades de gestión de concurrencia que dispone la plataforma [49]. Esto permite a grandes proveedores de computación en la nube implementar Hadoop como servicio y cobrar por su utilización. De esta forma, los grandes proveedores de "Cloud" para Big Data usan como plataforma de base diferentes versiones de Hadoop, con capas de software personalizado sobre la misma para ofrecer servicios diferenciados. Pueden verse los casos de AWS EMR[13], HDP[14] o CDH[15]

- API: Hadoop implementa diferentes clases de API. Para los desarrolladores tiene API para extender cada sección del flujo de trabajo y de datos de una aplicación MapReduce. Esto permite al programador personalizar cada etapa según las necesidades de negocio que necesita implementar. Asimismo, las ultimas versiones incorporan una API Rest para proveer información de los servicios, de manera que los usuarios avanzados o los proveedores implementen interfaces mejoradas respecto a los servicios que Hadoop trae incorporados.

Las ventajas anteriormente planteadas otorgan a Hadoop un lugar privilegiado en el contexto actual. La plataforma puede ser utilizada tanto por personas que se inician en el ambiente de sistemas distribuidos y procesamiento de datos masivos, como por organizaciones e investigadores con amplia trayectoria en el área [28].

### Arquitectura de un Cluster Hadoop

Hadoop es un sistema distribuido. Esta integrado por un conjunto de servicios principales y provee también algunos servicios secundarios. Como se puede observar en la Figura 2.1, estos servicios tienen una arquitectura master/worker desplegada a lo largo de un cluster de nodos independientes. Al momento de instalación, uno de los nodos es elegido como master por

[13] `https://aws.amazon.com/emr/`
[14] `https://es.hortonworks.com/products/data-center/hdp/`
[15] `https://www.cloudera.com/products/enterprise-data-hub.html`

configuración, y el resto de los nodos son establecidos como workers de la misma manera. Las acciones que realizan los workers son aquellas que el master ordene o defina que pueden ser realizadas.

Los usuarios se comunican con el cluster a través de un cliente que se conecta con el nodo master en un esquema cliente/servidor. Esto significa que el nodo master esta a la espera de solicitud de servicios por parte de los clientes, y responde las peticiones de los mismos. Cuando la operación lo requiere, el master permite que los clientes se comuniquen con los workers para llevar a cabo sus operaciones.

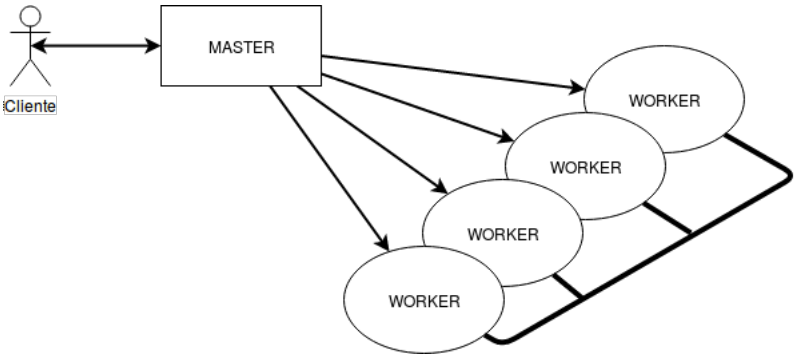



Figura 2.1: Arquitectura de alto nivel de un cluster Hadoop

El núcleo de hadoop dispone de 2 servicios. Cada uno implementa una arquitectura master/worker, tal como fue detallada anteriormente. Por un lado, el servicio de almacenamiento HDFS [47], que es una implementación de un sistema de archivos distribuido basado en GFS [17]. Por otro lado, se encuentra el servicio de planificación y procesamiento, conocido como YARN. Se dispone entonces de 2 procesos master, uno por cada servicio. Analogamente, se disponen de 2 servicios workers. Es habitual que en un nodo master se ejecutan concurrentemente los 2 servicios master. De la misma forma, es habitual que los nodos workers ejecuten ambos servicios workers de manera simultanea.

Hadoop implementa un esquema de procesamiento conocido como SIMD [48]. Esto significa que el modelo de ejecución esperado es aquel en el que un mismo bloque de instrucciones sera ejecutado de forma repetida sobre una gran cantidad de datos. Por otro lado, se parte de una filosofía conocida como *el código hacia los datos* [28]. Esto significa que se asume que los datos a procesar ya están almacenados en el sistema de archivos distribuido y Hadoop distribuye el código para ser ejecutado por cada uno de los diferentes workers.

Este enfoque es posible debido a la coordinación entre HDFS y YARN. Hadoop implementa la *localidad de datos* [49]. Este concepto establece que YARN coordina con HDFS las tareas de tal forma que maximiza las posibilidades de procesar localmente donde se encuentra almacenada la porción de datos que corresponde. Esta política busca minimizar la transferencia de datos a través de la red. HDFS provee de un conocimiento de grano fino acerca de la ubicación de cada porción de datos, lo que permite a hadoop optimizar de forma transparente las aplicaciones en su despliegue y procesamiento a lo largo del cluster.

A pesar de lo anterior, puede ocurrir que deba transferirse datos entre nodos, debido a que un nodo dispone de varios recursos de datos pero sus recursos de procesamiento alcanzaron el limite permitido. En este caso, Hadoop despliega el procesamiento en otro nodo y se realizara una transferencia de datos entre los workers involucrados. Se asume que Hadoop se ejecuta en un "datacenter" con el "throughput"[16] que se puede alcanzar en dicho contexto [49].

### MapReduce

*MapReduce* es un enfoque para el desarrollo de aplicaciones distribuidas que fue presentado en 2004 [11]. Se muestra como una estrategia para lidiar con problemas cuya computabilidad es relativamente sencilla de ser paralelizada, pero los datos a ser procesados tienen tal escala que obligan a ser procesados de forma distribuida [11].

La escala anteriormente mencionada genera que un algoritmo que puede ser desarrollado sin mayores inconvenientes y expresado de forma sencilla en pocas lineas de código, se vuelve innecesariamente complejo por detalles de paralelización, distribución de datos y manejo de errores [11].

Tanto en su planteo original como en *Hadoop*, *MapReduce* se refiere tanto a un modelo de programación como a una implementación asociada que provee un entorno para desarrollar aplicaciones.

De manera simplificada, el enfoque plantea lo siguiente: El desarrollador de aplicaciones define 2 funciones. Una función es llamada *Map* y la otra se identifica como *Reduce*. La función *Map* recibe los datos de entrada de los archivos a ser procesados. Una unidad de datos puede ser, por ejemplo, una linea de un archivo, un documento, una palabra, un caracter, una imagen, etcetera. La función *Map* procesa entonces cada unidad de datos. Transforma los datos dependiendo del procesamiento que el programador desea realizar

[16] El término "throughput" puede definirse como tasa de trabajo real de un sistema. En el contexto de un sistema de comunicación, puede definirse como la tasa de transferencia de datos útiles del mismo.

y al terminar se emite un par clave/valor. Por su parte, la función *Reduce* recibe las claves y listas de valores emitidos por los procesos *Map*. El framework se encarga de agrupar todos los valores con una misma clave para que sean procesados por el mismo Reducer (Proceso que ejecuta la función Reduce). Finalmente, en el proceso Reduce se genera pares clave/valor definitivos para la salida buscada, que es almacenada en un archivo de salida según corresponda.

Este enfoque MapReduce presenta algunas características a destacar:

Lo primero es que el Framework permite definir una *unidad de datos* la cual será paralelizada en la ejecución de las funciones *map*, para que se ejecuten de forma simultanea tanto en un equipo con varios núcleos de CPU, como en diversos equipos a través de un cluster de computadoras. Esto tiene una consecuencia: Los datos entre funciones *map* son independientes entre si. Es decir, un Mapper (Proceso que se encarga de ejecutar la función *map*) no depende de ningún otro Mapper para lograr su objetivo.

Segundo: Para lograr esta ejecución distribuida, el framework ofrece de forma nativa una manera de dividir los datos de forma controlada, para que cada host del cluster tenga disponible datos de entrada para procesar en cada función *map* que se ejecute. Esta operación de división ("split") y distribución de datos es proporcionada por HDFS [47].

Tercero, MapReduce plantea un enfoque SIMD (Single-Instruction Multiple-Data) [48]. Esto significa que los Mappers y Reducers ejecutan el mismo código de forma paralela sobre todos los datos de entrada.

Cuarto, para lograr un correcto funcionamiento, los Reducers no deben iniciar la ejecución hasta que todo los Mappers finalicen el procesamiento. Al final de la ejecución de los Mappers se establece un punto de sincronización para toda la ejecución. A partir de dicho punto, los Reducers reciben la salida de los Mappers combinadas por el framework bajo la misma clave. Al tiempo total de tareas maps ejecutadas se lo designa también como fase Map. Análogamente, se define la fase Reduce como el tiempo donde tiene lugar la ejecución de las funciones Reduce.

Quinto, como HDFS proporciona conocimiento de la *localidad de datos*, el framework intenta desplegar las tareas Maps en los hosts donde los datos a procesar se encuentran almacenados de forma local. Esto ayuda a minimizar el uso de la red.

Por último, al tener conocimiento el framework de cada tarea desplegada, y que datos de entrada le fueron asignados, en caso de un falla temporal debido a problemas imprevistos, es posible desplagar la tarea nuevamente en un nodo diferente. De esta manera, Hadoop provee tolerancia a fallas.

Para introducir la terminología propia del entorno, se explica la secuencia de ejecución de un algoritmo MapReduce en Hadoop. Se definen los conceptos

en la medida que se mencionan.

Cada ejecución de un algoritmo Hadoop es conocida como Job. Un Job de Hadoop tiene datos de entrada, normlamente enviados como archivo por parámetro al momento de ejecutar el proceso. Además debe tener una función map y puede tener función reduce definida por el desarrollador. El resultado del Job puede ser un archivo escrito en HDFS o impresa por salida estándar.

Un Job se divide en tres fases: Fase Map, que es donde se ejecutan todos los procesos Maps invocando la función map definida previamente. La fase Reduce es donde tiene lugar la ejecución de todos los Reduces haciendo uso de la función reduce y una fase intermedia conocida como Shuffle, donde todas las claves emitidas por todos los mappers son reunidas, agrupadas y ordenadas, para ser enviadas a la fase Reduce.

Cada proceso map o reduce que tiene lugar es conocido como Task, y es asignada a un worker dentro del cluster por el master de Hadoop. Cada task tiene datos de entrada con la forma clave/valor. Si la task falla, el entorno tiene la información para re-ejecutarla. Cada ejecución de una misma Task se denomina TaskAttempt.

La fase intermedia agrupa todas las claves de mismo valor que son emitidas por cada tarea Map, reuniendo las salidas de todos los workers en un mismo lugar. A continuación ordena las claves y en dicho orden son enviadas a los workers que procesaran las Tasks Reduce.

Como se establece anteriormente, el framework de procesamiento trabaja con una arquitectura master/worker. YARN es el servicio de procesamiento sobre el que se ejecuta MapReduce. El master de YARN es el servicio ResourceManager. Los workers ejecutan el servicio NodeManager. En un cluster Hadoop que ejecuta algoritmos de producción, es habitual que un nodo particular ejecute el servicio ResourceManager y el resto de los nodos ejecuten el servicio NodeManager.

Para cerrar, y debido al amplio conjunto de conceptos y nombres introducidos en la presente sección, se realiza un resumen de lo explicado.

- MapReduce es un enfoque para desarrollar aplicaciones distribuidas y escalables.

- El programador provee al entorno una función Map y otra función Reduce, que reciben datos con la estructura clave/valor.

- La función Map se ejecuta de forma distribuida en todo el cluster y procesa todos los datos de entrada, que el framework se encarga de dividir en bloques que serán procesados de forma simultanea en los diferentes nodos workers.

- La función Reduce se ejecuta al término de todas las funciones Maps, y recibe la salida de las mismas agrupadas y ordenadas por clave.
- Hadoop realiza toda la ejecución haciendo uso del servicio YARN y el motor de procesamiento MapReduce implementado sobre dicho servicio.
- YARN tiene una arquitectura master/worker, siendo master el servicio ResourceManager y workers los nodos que corren NodeManager

### MapReduce en Detalle

La explicación previa del framework MapReduce es suficiente para comprender de forma sencilla el funcionamiento de una aplicación construida sobre Hadoop. Pero en el presente trabajo se ha utilizado de forma exhaustiva, lo que implica que no alcanza con desarrollar Map y Reduce para mostrar todo los algoritmos.

Como es esperable, hablar de 2 fases es una simplificación. Para realizar una tarea MapReduce el cluster ejecuta una serie de sub fases. El desarrollador puede optar por no intervenir en las mismas, y Hadoop tiene una política por default para actuar en dicho caso. En caso contrario, el desarrollador puede extender ciertas clases y definir un nuevo comportamiento, siguiendo la API que Hadoop define en su documentación oficial[17].

Para entender mejor como funciona MapReduce, es necesario describir en forma detallada la ejecución de un Job 2.2. De forma esquemática, un job MapReduce tiene las siguientes etapas definidas:

- Fase Input
- Fase Map
- Fase Shuffle
- Fase Reduce
- Fase Output

Fase Input: Especifica el formato y estructura de la entrada para un Job MapReduce. Lo hace a través de la clase InputFormat (y extensiones de la misma). Entre sus responsabilidades está la de dividir los archivos de entrada en partes conocidas como Input Splits y asignarlas entre las diferentes tareas map que serán desplegadas a lo largo del cluster. Los input split son divisiones

[17] `http://hadoop.apache.org/docs/stable/`

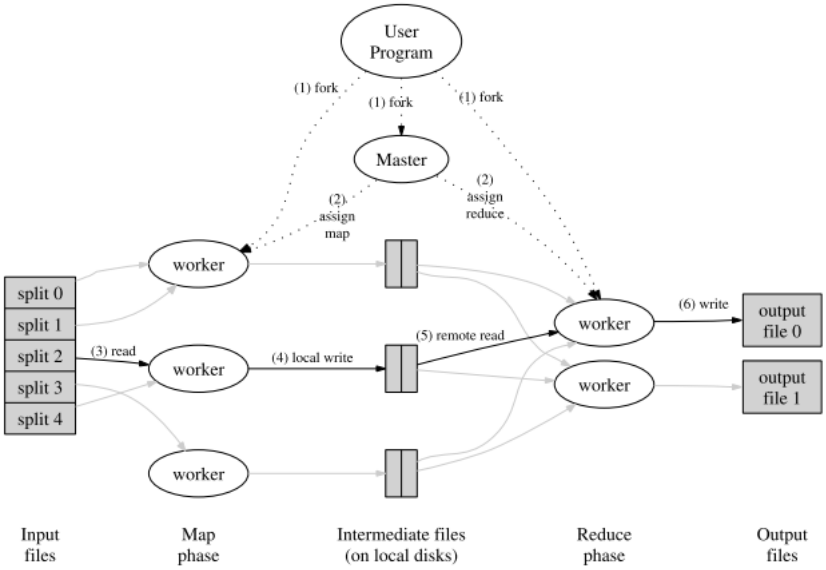



Figura 2.2: Ejecución de un Trabajo MapReduce [11]

lógicas de los datos de entrada preparadas para ser procesadas por cada uno de los procesos Map. Por ejemplo, si se provee como entrada un archivo de texto, la clase InputFormat construye un Input Split por linea y crea un par clave-valor donde la clave es el numero de linea y el valor es el contenido de dicha linea.

Fase Map: Fase que procesa todos los Input Splits, clave por clave. En un entorno distribuido, la fase Input distribuyó los InputSplits a través de los nodos worker (en general teniendo en cuenta la localidad de datos a través de HDFS), y los procesos Map se ejecutan por cada InputSplit disponible. Cada una de estas ejecuciones genera un nuevo conjunto de datos en la forma `clave-valor`, y son almacenados de forma local, notificando al nodo master que se encuentra la salida disponible para la siguiente fase.

Fase Shuffle: Esta fase tiene 2 responsabilidades principales. Primero recorre el listado de claves generados por los Mappers y reúne aquellos que tengan la misma clave. La estructura de datos ahora se transforma de `clave-valor` a `clave-lista(valores)`. La otra responsabilidad que tiene asignada esta fase es la de ordenar todas las claves, de manera que la siguiente fase procese de forma ordenada las mismas. En esta fase es en donde se produce la mayor transferencia de datos a través de la red, ya que a la hora de reunir los pares `clave-valor`, es necesario reagruparlos físicamente para realizar las operaciones. Shuffle deja preparados todos los datos para que sean utilizados por la siguiente fase.

Fase Reduce: Fase que recibe los pares `clave-lista(valores)` construidos por el proceso Shuffle y realiza el procesamiento correspondiente. Esta fase agrupa, reúne y consolida la información antes de ejecutar la siguiente fase del proceso. La fase Reduce genera una salida con el formato `clave-valor`.

Fase Output: Esta fase funciona y se construye de manera similar a la fase de Input, pero si en dicha primer fase se define la manera en que los registros son leídos desde los archivos de entrada, esta fase especifica como son escritos los datos de salida. La abstracción que provee HDFS permite que en esta fase se trabaje a nivel de archivos y directorios, implementando toda la API de un sistema de archivos como resulta esperable[18]: Abrir archivos, escribir bytes o caracteres, entre otros ejemplos. La lógica de la sincronización distribuida es transparente para el desarrollador.

### HDFS

Si bien MapReduce se puede usar como motor de procesamiento sin depender de un sistema de archivos distribuido, este se beneficia cuando se utiliza de manera integrada con el sistema de archivos distribuido que provee Hadoop. Su nombre es HDFS y es un acrónimo de Hadoop Distributed File System.

MapReduce escala muy bien cuando la cantidad de nodos del cluster crece. Puede utilizar miles de nodos de forma coordinada sin problemas [49]. Para que esta coordinación y utilización sea eficiente, es necesario contar con un sistema de archivos que se encargue de que los datos se encuentren distribuidos en el cluster antes de que sean procesados. En esta tarea HDFS hace su aporte.

Los sistemas de archivos distribuidos tienen como uno de sus objetivos de diseño generar un espacio de nombres abstrayendo la naturaleza de su implementación subyacente, que significa que en realidad se encuentra ejecutándose en mas de un nodo físico [17] [47].

HDFS provee un espacio de nombres similar al utilizado en los sistemas tipo UNIX, siendo la raíz un directorio / (caracter de barra o slash), y de ahí el resto de árbol de directorios se desprende según las necesidades y el uso del Sistema. Sin embargo, eso es una visión para los programas MapReduce y para los administradores. HDFS en realidad maneja un complejo esquema de distribución y réplica de datos. Mas alla de las similitudes de referencia con los sistemas de archivos "UNIX-like", no se provee una interfaz estándar

[18] `https://hadoop.apache.org/docs/current/api/org/apache/hadoop/fs/FileSystem.html`

para su acceso, sacrificando esto en favor de mejorar el rendimiento de las aplicaciones que lo utilizan [47].

HDFS persigue el objetivo de tener alto rendimiento para un esquema de utilización, debido a que no esta diseñado para ser un sistema de archivos de propósito general, sino específicamente para trabajar con aplicaciones MapReduce y similares, las cuales procesan grandes conjuntos de datos. HDFS esta pensado para ser eficiente bajo las siguientes condiciones [49]:

- Archivos muy grandes: Hadoop trabaja muy bien cuando debe almacenar archivos grandes. Es preferible pocos archivos muy grandes que una gran cantidad de pequeños archivos. Hadoop tiene esquemas de partición y replica de archivos. Puede dividir el archivo en muchas partes y rearmarlo en caso de necesitarlo. Al replicarlo, en caso de falla de uno o varios nodos, el archivo completo sigue estando disponible.

- El patrón de procesamiento con el que HDFS mejora su rendimiento es conocido como "Write-Once;Read-Many". Esto es, un conjunto de datos cuya cantidad de escrituras es pequeña comparada con la cantidad de lecturas al mismo conjunto. HDFS tiene alto "throughput" en la lectura de los datos secuenciales [47]. Este tipo de modelo es clásico en esquemas de procesamiento en lote o "batch".

- Una última característica es que funciona muy bien sobre hardware económico. No solo económico desde el precio, sino también desde que HDFS no requiere Hardware especializado o complejo, sino que hardware utilizado en equipos de usuario final puede utilizarse sin inconvenientes. Esto es así porque tanto Hadoop como HDFS tienen diversos mecanismos de recuperación ante errores, que permite continuar el procesamiento a pesar de que algún nodo (o conjunto de ellos) particular puede fallar.

La característica clave de HDFS esta en su posibilidad de particionar los datos, y replicar estas particiones. Ambas técnicas apuntan a tener disponibilidad y tolerancia a fallos. Todos los archivos subidos a Hadoop se particionan en bloques. Por defecto estos bloques son de 64Mb. Por ejemplo, si un cliente sube a HDFS un archivo de 256Mb, el mismo es almacenado físicamente en 4 bloques de 64Mb. Si el cluster dispone de 4 nodos workers, El servicio intenta que cada partición se almacene en cada nodo.

Los bloques creados por Hadoop son replicados. Replicar significa crear copias idénticas de los bloques y alojarlas físicamente en nodos workers diferentes. Si fuera posible, HDFS también intenta almacenarlo en diferentes racks dentro del mismo datacenter. HDFS permite que se le suministre la

distribución del datacenter, de manera que pueda ser consciente de los racks disponibles. Esto es útil porque puede suceder que por motivos administrativos se pierda acceso a un rack de nodos completo. Si HDFS conoce la topología, puede ubicar los bloques de archivos en nodos de diferentes racks. Otra consecuencia es que además de minimizar la transferencia inter-nodo, también lo hace inter-rack, asumiendo que la conectividad intra-rack tiene una velocidad de transferencia mayor que la conectividad inter-rack.

Para tener una dimensión de la capacidad de HDFS, en 2009 la empresa Yahoo reportaba un uso del sistema de archivos distribuido que reunía las siguientes métricas [46]:

- 14 Petabytes
- 4.000 Nodos
- 15.000 Clientes
- 60.000.000 Archivos

## 2.2. Recuperación de Información

La recuperación de información (también conocida como IR, por sus siglas en ingles) es una disciplina surgida de la necesidad de disponer del acceso a documentos digitales de naturaleza no estructurada.

Baeza-Yates [4] plantea que la Recuperación de Información trata con la representación, almacenamiento, organización y acceso a ítems de información. El presente trabajo se involucra de forma activa con las áreas de representación, almacenamiento y organización. Los problemas asociados con el acceso quedan fuera del alcance del análisis.

La IR ha tenido una evolución constante desde su surgimiento, y actualmente involucra campos tan diversos como procesamiento de lenguaje natural, estructuras de datos y algoritmos de compresión, por mencionar algunos ejemplos.

Desde un enfoque simplificado, se puede plantear que el proceso completo de recuperación de información consiste en las siguientes etapas:

1. Definición y/o creación de la colección de documentos.
2. Construcción de estructuras que soporten la recuperación.
3. Resolución de consultas de usuarios utilizando las estructuras previamente construidas.

Se define como colección $C$ al conjunto de documentos $d$ sobre los que un usuario tiene interés en realizar consultas $q$ y obtener un resultado. En IRtradicional, la base de documentos es previamente conocida. Con el surgimiento de la web, la colección esta distribuida a lo largo de todo Internet, y es necesario una paso de construcción previo conocido como Crawling [4].

Una vez definida la colección $C$, Los sistemas de IR proveen una interfaz para que los usuarios realicen consultas. Una consulta $q$ es una necesidad de información de un usuario expresada en algún tipo de lenguaje que pueda ser procesado por el motor de búsquedas.

Con una cantidad pequeña de documentos, se pueden utilizar mecanismos proporcionados por los sistemas operativos para resolver las consultas (Expresiones regulares o patrones de texto sobre la colección con comandos como "grep" o similares). Sin embargo, a partir de cierto volumen, se debe disponer de mecanismos mas eficientes, sobre todo métodos que no necesiten recorrer toda la colección cada vez que se desea realizar una consulta. Mas importante aun, se tiene que poder establecer un orden de relevancia a cada documento que coincida con una consulta $q$ proporcionada por el usuario. Es decir, algunos documentos son mas relevantes que otros para una consulta dada. Estos requisitos implican obtener estadísticas de la colección (Como el $TF$ y el $DF$ de un término, por ejemplo) [33].

Empiezan a aparecer límites en la capacidad de las herramientas de procesamiento de texto convencionales para resolver una consulta de IR, y esto sucede porque los usuarios de los sistemas de recuperación de información ($SRI$) tienen requisitos avanzados a la hora de construir sus consultas y simultáneamente exigen que las respuestas del sistema sean precisas en periodos de tiempos cada vez mas cortos.

Para satisfacer dichos requisitos aparecen estructuras de datos intermedias, que son construidas a partir de la colección pero permiten almacenar información adicional, además de soportar recorridos mas eficientes que hacerlo directamente sobre la colección. Estas estructuras se persisten en almacenamiento secundario y son conocidas de forma genérica como índices.

La literatura de IR ofrece muchas propuestas sobre posibles índices, pero al día de hoy, el que mas popularidad y desarrollo tiene es el conocido como Índice Invertido [50].

### 2.2.1. Construcción de Índices

Un índice invertido ($II$) es una estructura de datos que se construye a partir de la colección $C$, y permite acelerar las búsquedas. Cuando $C$ es suficientemente grande, un índice mejora el rendimiento de la recuperación en relación a la búsqueda secuencial sobre dicha colección [4].

Un $II$ esta compuesto por 2 partes: El vocabulario $V$ y las listas de ocurrencias $p$. El vocabulario es la lista de todos los diferentes términos de $C$. Para cada término se construye una lista de los documentos o posiciones en los que aparece el mismo. Esta lista es la lista de ocurrencias o lista de postings, también referida como "posting list" [10].

En las implementaciones es común que ambas partes (vocabulario y posting list) estén separadas en archivos diferentes. En general el vocabulario tiene requisitos de espacio mucho menores que la posting list [4]. En función de la cantidad de información a almacenar de cada término, el índice resultante puede ocupar del 10 al 100 por ciento, o incluso mas, que la colección original. Por otro lado, tener un índice implica a su vez contemplar la necesidad de actualizarlo. Dependiendo de la forma de implementación, esto puede ser mas o menos complejo [16].

Una decisión de diseño de los algoritmos de construcción de índices esta relacionada con definir qué recurso de hardware sera explotado. En general el compromiso se plantea entre memoria principal y almacenamiento secundario. Los algoritmos basados en memoria principal tienden a construir el índice mas rápido pero son limitados debido a la disponibilidad de memoria [33]. Por el contrario, el índice creado en almacenamiento secundario puede realizar todo el procesamiento incluso para grandes cantidades de datos pero tiende a tardar mas tiempo por las velocidades de acceso propias del dispositivo.

### Índice Básico

Este tipo de índice es similar al que se encuentra descripto en la bibliografía sobre IR [4] [33] [10]. Contiene 2 tipos de registros. Uno es el registro de vocabulario, donde se establece cual es el término indexado y suelen almacenarse estadísticas globales de dicho término que ayudan a la recuperación (frecuencia del término en la colección y cantidad de documentos donde el término aparece por ejemplo). El otro es la "posting list", el listado de ocurrencias o apariciones del término por documento. Es un conjunto de elementos donde se identifican principalmente el documento donde el término aparece, y la frecuencia con la cual el mismo se repite dentro del documento en cuestión.

Almacenando la información anteriormente mencionada, un motor de búsqueda puede implementar sobre este tipo de índices técnicas de recuperación basadas en los modelos booleano, vectorial o probabilístico por ejemplo.

#### Índice Block-Max

Los índices Block-Max [13] ($BM$) proponen una estructura auxiliar que permite al motor de búsqueda incorporar técnicas de terminación temprana o "early-terminaton" para resolución de consultas en índices basados en el enfoque WAND [6]. Las posting lists son divididas en bloques de tamaño fijo. Cada bloque guarda información sobre el mayor "impact score" (mayor frecuencia) del bloque en una cabecera de bloque. La unidad para compresión se puede situar a nivel de cada bloque en vez de la posting list completa. A esto además se agrega la ventaja de que no se necesita descomprimir cada uno cuando se realiza la recuperación, sino que la misma se puede realizar evaluando el impact score del bloque, a partir de la cual se puede descartarlo completamente o determinar que contiene postings de interés.

En lo que respecta a la estructura conceptual de un índice $BM$ se dispone de 2 estructuras diferentes: vocabulario y posting lists. Esta característica es coincidente con la mencionada para el índice básico (Sección 2.2.1). Solo que en el caso de cada posting list, se almacena información adicional como cabecera de cada bloque para poder tomar decisiones sin la necesidad de descomprimirlo.

### 2.2.2. Construcción Distribuida de Índices

El enfoque de construcción de índices invertidos para IR en un contexto distribuido es un tema ampliamente tratado en la literatura [3] [43]. En general se asume que el índice es construido en un cluster de equipos que trabajan de manera autónoma y coordinada para lograr su objetivo. Una opción para este tipo de clusters es que, construyéndose a partir de equipos económicos y mediante la configuración y algoritmos adecuados, pueden alcanzar prestaciones equivalentes a una supercomputadora [41].

El primer enfoque posible es el de un índice que sea construido en memoria principal de los nodos. Esto implica que el tamaño del índice resultante es menor que la memoria sumada de todos los dispositivos. Para lograr este objetivo los nodos deben contener la colección distribuida de forma previa a la ejecución del algoritmo. La clave en este esquema es utilizar lectura desde almacenamiento secundario al principio del algoritmo, y escritura al final del proceso. Luego, solo se utiliza memoria principal y mensajes a través de la red. La compresión de datos es utilizada también en este contexto para optimizar el almacenamiento [41].

En determinado momento del crecimiento de una colección el índice completo no pueda ser mantenido ni construido completamente en memoria principal. Enfoques diversos son propuestos para alcanzar un algoritmo que puede

trabajar de forma distribuida con este contexto. La clave en estos enfoques se trataba de generar estructuras de datos que puedan distribuirse de forma parcial a través de la red, y decidiendo si es conveniente usar almacenamiento secundario o red según el caso, para minimizar el retardo de escrituras parciales [42].

También esta problemática es abordada desde la perspectiva de los datos masivos utilizando algoritmos MapReduce. Con la propuesta original de MapReduce, una de las implementaciones propuestas es la de un algoritmo para construcción de índices invertidos. Dicha propuesta utiliza el motor de procesamiento presentado y simplifica la tarea de crear índices. Es importante destacar que al usar MapReduce, se tiene distribuida previamente la colección debido al Sistema de archivos distribuido subyacente, y que MapReduce se encarga de la distribución de procesos y procesamiento local. El algoritmo se simplifica a: I) Crear una función map que escanee los documentos locales, y emita un par `<término,id documento>`. II) Una función reduce que recibe `<término, list(id documento)>`, realice las operaciones de suma correspondientes y grabe a disco [11].

Posteriormente se realizan comparaciones de diferentes algoritmos para indexación con otros esquemas mas complejos. El algoritmo original [11] hace un uso intensivo de la red de forma innecesaria. Es posible mejorar el Map agregando procesamiento parcial del documento, como se propone en el algoritmo de Nutch [35]. La estrategia mas completa dentro de estas propuestas es crear de forma local las posting lists por término, y el objetivo del proceso reduce es realizar un merge de dichas listas.

Lin [31] utiliza características avanzadas de Hadoop para plantear un algoritmo ligeramente diferente. Teniendo en cuenta que MapReduce realiza una operación de ordenamiento en la fase intermedia de forma nativa a partir de las claves, plantea el uso de claves compuestas, donde el motor puede ordenar dichas claves (el programador debe indicar a Hadoop como se ordena dicho tipo de dato no nativo).

## 2.3. Compresión de Datos

En Recuperación de Información, una de las áreas clave para mejorar el rendimiento de los motores de búsqueda es la compresión de los índices (tanto en su construcción como para la recuperación posterior). El proceso de compresión mejora el rendimiento general de las aplicaciones que hacen uso de dichos índices [7]. Diversos estudios muestran que la complejidad de estas estructuras requieren que diferentes partes de lo mismos sean comprimidas utilizando estrategias distintas [8]. Una estrategia de compresión única no

permite lograr el objetivo de la forma más eficiente.

La compresión de índices tiene un alto impacto en el rendimiento a la hora de su construcción. Un método adecuado puede lograr una reducción significativa de los datos a ser escritos a disco. Por otro lado, el tiempo que agrega el proceso de compresión puede verse compensado por el ahorro de tiempo a la hora de escribir menor cantidad de datos en almacenamiento secundario.

Sin embargo, los métodos de compresión de datos deben ser evaluados teniendo en cuenta los procesos de velocidad de compresión y de descompresión del mismo y su tasa de compresión. La velocidad de descompresión es importante debido a que es un proceso que se realiza al momento de resolución de cada consulta realizada por los usuarios al $SRI$.

PForDelta y Simple16 son los 2 algoritmos que se utilizan para la compresión de datos. Se describen a continuación las características principales de cada uno. Se dejan notas acerca del algoritmo Simple9 para comprender mejor su extensión Simple16.

**PForDelta**: Frame-of-reference (FOR) y sus derivados (PFOR, FastPFOR, PForDelta), son técnicas de compresión de enteros que separan la lista de números en bloques de tamaño fijo. Estos bloques son computados con sus saltos[19] a partir del mayor y menor valor $m$ del bloque. El valor $m$ es almacenado en binario y los restantes valores del bloque son almacenados como diferencia respecto a $m$ utilizando una cantidad de bits menor a la necesaria en caso de que se almacenen como números enteros por separado. Las alternativas derivadas están relacionadas con la mejora del manejo de excepciones que surgen al aplicar el criterio explicado de manera mas eficiente. Resultados experimentales muestran que PFOR en general tiene mejor rendimiento en la recuperación para resolución de consultas cuando se utiliza en bloques de identificadores de documentos [8].

**Simple9**: Este algoritmo se enfoca en seleccionar un tamaño de palabra fijo (por ejemplo 32 bits). Entonces, dado una secuencia de números enteros, se define cual es la mínima cantidad de bits que necesita para representarlos [2]. Definido esto, puede establecer cuantos enteros pueden representarse

[19]Para entender el funcionamiento de los algoritmos de compresión de números enteros, primero debe comprenderse que significa almacenar una secuencia de enteros en forma de saltos (técnica conocida como d-gaps o delta-gaps [8] [2]). El concepto de delta-gaps plantea que dada una lista de enteros ordenada de forma ascendente, se puede almacenar el primer valor de la lista y luego la diferencia a los sucesivos valores en sus posiciones correspondientes. Una vez que estas diferencias o saltos son computadas, se obtiene una nueva lista donde los valores a almacenar necesitan mucha menor cantidad de bits para ser almacenados. Dada una lista de 10 enteros: <15, 18, 22, 34, 35, 36, 39, 42, 45, 47 >, la representación en d-gaps de la misma lista es la siguiente: <15, 3, 4, 12, 1, 1, 3, 3, 3, 2 >

en 28 bits (4 quedan como cabecera). Por ejemplo, si se tienen 28 números 1 o 2, se pueden representar cada uno en un bit, y por lo tanto se almacenan todos en los 28 bits disponibles. Combinando este tamaño fijo se obtienen diferentes cantidades de enteros a almacenarse. Por ejemplo, si los enteros se representan en 4 bits, pueden almacenarse 7 enteros. Algunas combinaciones pueden dejar bits desperdiciados. Estas combinaciones son 9 en total y se llaman selectores. Los 4 bits de cabecera indican que selector se utiliza.

**Simple16**: Simple9 almacena en los 4 bits de cabecera el valor de cual de los 9 selectores fue utilizado para esa palabra. Resulta claro que con 4 bits pueden representarse 7 selectores mas, y eso es lo que busca hacer Simple16. Se basa en relajar la restricción indica que todos los enteros se almacenen en una cantidad fija de bits. Por ejemplo, un selector nuevo define que se representan 3 enteros en 5 bits y 2 en 6 bits. Las combinaciones se eligen de forma tal que no se necesite utilizar padding. Estas técnicas de codificación son adecuadas para cantidad de enteros que son considerados bajos. Al ser esta característica típica de las frecuencias (muchos términos tienen poca frecuencia), es uno de los métodos de compresión mas razonables y resultados experimentales muestran que estos algoritmos ofrecen una tasa de compresión razonable y logra un buen rendimiento en la velocidad de descompresión [8].

# Capítulo 3

# Propuestas de Indexación

En este trabajo se implementan algoritmos del área de recuperación de información para la construcción de índices a partir de colecciones de datos no estructuradas. Para el desarrollo de las implementaciones se utilizan los conceptos de Big Data y algoritmos MapReduce. Se construye y se mide el rendimiento de dichos algoritmos sobre la plataforma Hadoop en un cluster de hardware económico dedicado a estas tareas. Asimismo, se utilizan métodos de compresión de datos para los cuales se determinan las técnicas adecuadas para comprimir los índices propuestos.

En el capítulo 2 se desarrollan las ideas principales acerca de dos índices: Básico y Block-Max. Para cada uno de los mismos se presentan sus características de alto nivel y estructuras principales. Para la implementación de estas características pueden existir diferentes enfoques. En este capítulo se repasan cada uno de los índices y se describen detalles que se tienen en cuenta a la hora de su implementación, decisiones de diseño al momento de pensar los algoritmos MapReduce que construyen los índices y particularidades propias de Hadoop.

## 3.1. Generación del índice

Los algoritmos de indexación implementados comparten algunas características en común, que se describen a continuación:

- Todos los indexadores comparten la fase de tokenización, stemming y normalización de términos[1]. Estas tareas coinciden con la fase Map, la

[1]Todas estas y algunas mas son tareas comunes al procesar documentos de texto libre. Tokenizar implica definir que és un término indexable y que no lo es [18] y Stemming implica llevar los términos a su raíz morfológica [38].

cual es idéntica para todas las implementaciones.

- Los algoritmos de compresión son utilizados en todos los indexadores para comprimir los mismos tipos de datos.
- Se mantiene idéntica la configuración del cluster para todas las pruebas en una misma cantidad de nodos.
- Los algoritmos implementan 2 versiones del índice, comprimido y sin comprimir.

La información que almacenan los índices implementados es la siguiente:

- Término indexado
- Referencia a los identificadores de documentos donde aparece el término
- Frecuencia de los términos en cada documento donde aparece

La generación del índice invertido tiene varias etapas o fases. Cada una de estas etapas implementa alguna o varias de las siguientes responsabilidades:

1. Recorrido de la colección y procesamiento a nivel de documento.
2. Análisis del documento. Consiste en decidir que tokens (secuencias de string) son candidatos a convertirse en términos.
3. Cada token debe ser analizado en base a una serie de procesos, donde se define si es un término válido o no. Entre estos criterios de análisis se encuentra el stemming de los tokens, la cantidad de caracteres mínimos, por ejemplo.
4. Construcción de las posting lists. En esta etapa hay tantas postings como pares términos-documento existen en la colección.
5. Las postings para un mismo término deben ser integradas en un proceso, y ordenadas por identificador de documento.
6. Construcción del vocabulario.
7. Escritura del archivo de postings con el formato correspondiente.
8. Compresión de las posting lists si corresponde.

Tabla 3.1: Tareas en la generación del índice invertido por fase.

| Tarea | Descripción | Fase |
|---|---|---|
| 1 | Recorrido de la colección | Input |
| 2 | Extracción de tokens de los documentos | Map |
| 3 | Tratamiento de tokens; Procesamiento de términos. | |
| 4 | Construcción de registros término-documento | |
| 5 | Agrupamiento y ordenamiento de registros | Shuffle |
| 6 | Construcción del vocabulario | Output |
| 7 | Construcción del índice | |
| 8 | Compresión de las posting lists | |

Estas tareas se realizan en diferentes fases de los algoritmos MapReduce implementados. Es importante establecer esto para poder entender donde se realiza cada parte del procesamiento. Además, estas tareas pueden realizarse de forma conjunta para optimizar recursos. Por ejemplo, La construcción del archivo donde se almacenan las posting lists se realiza en paralelo a la compresión de la lista porque no necesita terminar la compresión de todos los elementos para almacenarlos.

Las tareas del listado anterior se asignan a las diferentes fases del proceso MapReduce como se explica a continuación (Ver Tabla 3.1). La fase Input realiza de forma automática el ítem 1. Los ítems 2, 3 y 4 se realizan en la fase Map. Durante el Shuffle tienen lugar las tareas definidas en 5. La fase Reduce integra la salida del Sort distribuido, construye la estructura de datos de las posting lists y pasa esta estructura a la siguiente fase. Por ultimo, la fase Output se encarga de ejecutar las actividades de 6, 7 y 8.

Implementado de la manera anterior resulta claro que la estructura del índice es una tarea involucrada en la ultima fase de procesamiento. Esto permite que se reutilicen los procesos de fases anteriores.

### 3.1.1. Algoritmo para Índice Baseline

Como se comenta en la Sección 2.2.1, la implementación del índice básico esta compuesto por estructuras que constan de un vocabulario y un conjunto de posting lists. Además de los problemas a resolver propios del proceso de construcción del índice, se utilizan conceptos avanzados de la plataforma Hadoop para poder explotar los algoritmos de forma exhaustiva aprovechando las capacidades de YARN y MapReduce.

Este algoritmo básico es utilizado como baseline de la comparación.

### Consideraciones

Cuando se habla de índice baseline, en realidad se hace referencia a un par de algoritmos casi idénticos, con la diferencia de que uno almacena la información comprimida respecto del otro. Esta decisión (Si el índice debe comprimirse o no) se toma en tiempo de ejecución, vía parámetros al Job MapReduce.

El índice baseline se implementa mediante 3 archivos físicos. Un archivo almacena todo el vocabulario de la colección. Las postings list están compuestas por 2 elementos: Identificador de documentos y frecuencia del término en cada documento. La decisión de diseño es dividir estos elementos en 2 archivos, uno de identificadores y otro de frecuencias.

Cada registros del vocabulario contiene los siguientes campos:

```
<Término> <TamañoPosting>
```

Donde:

- Término: El término $t$ indexado.
- TamañoPosting: La cantidad de elementos en la posting list. Coincide con la frecuencia en documentos de $t$, conocida como $df$.

Por otro lado se encuentra el archivo de postings lists, conocido como posting file y que contiene los siguientes campos:

```
1 PostingFile = <PostingList[1]> <PostingList[2]>
     ... <PostingList[N]>
2 PostingList[i] = <Termino> <Posting[1]> <Posting
     [2]> ... <Posting[N]>
3 Posting[j] = <IdDocumento> <Frecuencia>
```

Donde:

- IdDocumento: Un identificador de documento $d$ que permita ubicarlo unívocamente dentro de la colección
- Frecuencia: Cantidad de apariciones de $t$ en $d$

### Implementación en MapReduce

Como se describe en la sección 2.1.1 cuando se habla del funcionamiento del Framework MapReduce, se deja claro que el desarrollador interviene en las 2 fases principales del proceso: La fase Map, encargada de la construcción

y distribución de los pares clave-valor, y la fase Reduce, encargada de agrupar y agregar según corresponda todas las claves dispersas por los procesos Map a lo largo del Cluster.

El desarrollador puede también intervenir en otras sub fases en caso de necesitar personalizar parte de la ejecución. Hadoop provee un mecanismo de extensión de clases a través de la creación de nuevas clases que implementen una interfaz determinada vía API de MapReduce.

En particular para el proyecto, se extendieron las siguientes clases:

- Mapper
- Reducer
- Partitioner
- FileOutputFormat
- RecordWriter

Estas clases son extendidas para personalizar el comportamiento de las diferentes fases que tiene el flujo de datos planteado en la sección 2.1.1. Se detalla a continuación que clases intervienen en cada fase y que responsabilidades cumplen cada una, indicando además el flujo de datos a lo largo de toda la aplicación.

Las clases Mapper y Reducer son extendidas mediante la re-definición de los métodos `map()` y `reduce()` respectivamente.

La fase Input se encarga de abrir el archivo que se encuentra en HDFS y representa la colección en un formato determinado, separarlo en lineas (debido a que es un archivo de texto) y proveer cada función `map()` de una linea. Debido al formato del archivo de colección, este comportamiento garantiza que cada función `map()` recibe un documento completo. La entrada a cada task map que se ejecuta es entonces un par clave-valor donde la clave es el numero de linea y el valor es un string con el contenido del documento (Figura 3.1).

Cuando el proceso Mapper procesa la entrada, emite como salida nuevos pares clave-valor, donde la clave esta compuesta por dos datos: Término y Identificador de Documento. El valor del par contiene la frecuencia del término en el documento que esta siendo procesado. Cada map emite tantos pares como combinaciones término-documento contenga. Estos pares son emitidos en disco local, y desde ahí son recuperados por la siguiente fase (Figura 3.2).

El Reducer implementa un flujo de datos que recibe una estructura de datos clave-valor tal como es emitida por la fase Shuffle. La clave contiene

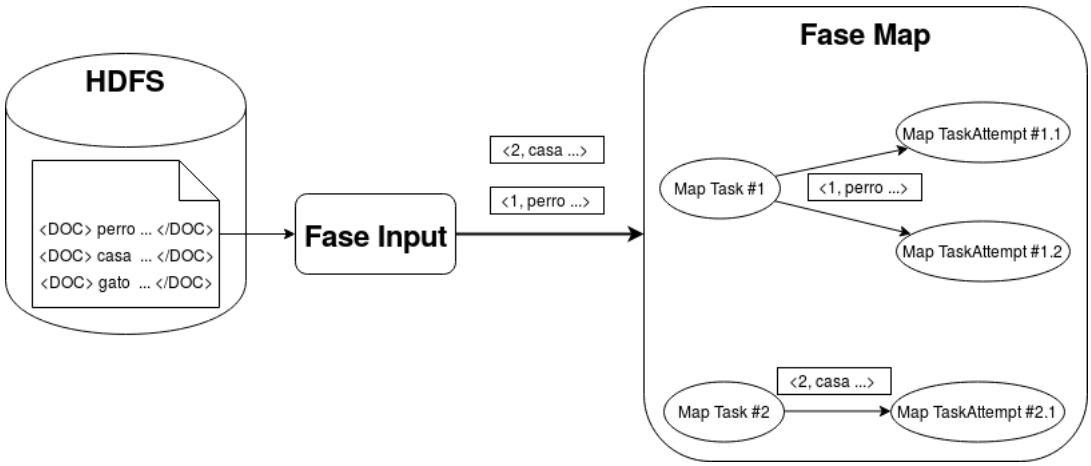



Figura 3.1: Flujo de datos desde la fase Input a la fase Map

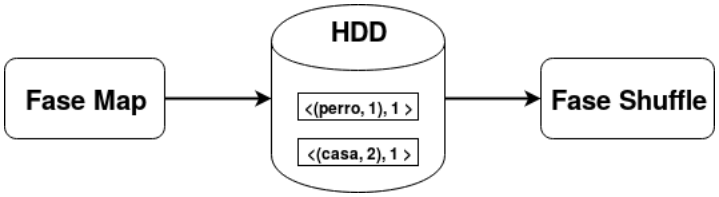



Figura 3.2: Flujo de datos desde la fase Map a la Fase Shuffle

información sobre el término y el documento. El valor son las frecuencias acumuladas. El reduce procesa de forma ordenada todas las claves (mismo término en cada uno de los documentos donde aparece), registra las frecuencias respectivas, y construye la posting en memoria (Figura 3.3). Debido a la intervención de la clase Partitioner, cada `reduce()` tiene garantizado que recibe todas las claves de un mismo término.

Cada proceso `reduce()` entrega a la ultima fase del proceso, la fase Output, un par término-posting list. El Output recorre la posting y genera los archivos de índice con el formato definido, los cuales son almacenados en HDFS (Figura 3.4).

Dentro de la fase Shuffle, tiene lugar la tarea de definir las particiones del espacio de claves. Esta partición implica dividir las claves entre los diferentes procesos Reducers. Esta partición debe ser realizada de tal forma que garantice que claves idénticas lleguen al mismo Reduce. Esto es necesario porque el proceso Reduce asume que cada clave llega con todos los valores procesados por todos los Maps. En este caso es mas complejo, porque al ser clave

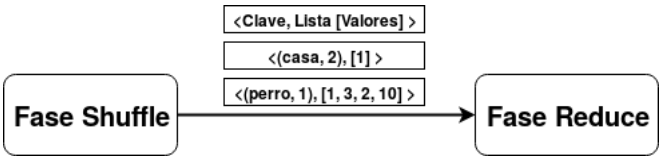



Figura 3.3: Flujo de datos desde la fase Shuffle a la fase Reduce

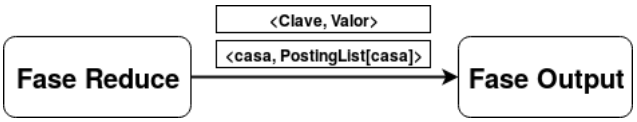



Figura 3.4: Flujo de datos desde la fase Reduce a la fase Output

compuesta, es necesario garantizar que todos los pares cuyo primer elemento de la clave (que es el "término") es idéntico son asignados al mismo Reduce. Esto se logra redefiniendo la clase Partitioner.

La clase Partitioner es utilizada por el framework dentro la fase de Shuffle, y tiene lugar como sub-fase antes de que el entorno realice el sort distribuido[2]. Esto tiene como consecuencia que el sort se realiza sobre las claves que pertenecen a la misma partición. Por lo general, es deseable que las particiones estén balanceadas, o dicho de otra forma, cada partición contenga una cantidad similar de claves.

Cuando el proceso Reducer necesita escribir el índice en HDFS, no lo hace de forma directa, sino que delega esta responsabilidad en un objeto llamado FileOutputFormat. Este objeto tiene como misión encargarse de lidiar con detalles de alto nivel del sistema de archivos (crear los archivos y abrirlos, por ejemplo) e invocar al objeto RecordWriter para que escriba los datos de salida en el formato adecuado. RecordWriter implementa el formato físico del archivo de salida (o los archivos de salida en caso de ser mas de uno). En el caso de este trabajo, el formato del Índice propiamente dicho se encuentra en los RecordWriter extendidos.

Entre los parámetros que permiten especificar la forma de construcción del índice, se encuentra la definición de si se desea almacenar un índice utilizando técnicas de compresión o no. Por la forma que Hadoop permite extender las clases, fue necesario implementar 2 RecordWriter, donde uno almacena la información directamente, y otro donde previamente es comprimida.

En el caso de los índices comprimidos, no se utilizaron técnicas de com-

[2] `https://developer.yahoo.com/hadoop/tutorial/module5.html#partitioning`

presión que incluye Hadoop, debido a que son métodos de compresión de propósito general y se privilegio comprimir los índices usando técnicas de compresión especificas.

### Formato Físico del Índice sin Compresión

A continuación se define cual es el formato físico del índice, o de otro modo, la forma en que los datos son almacenados para que puedan ser utilizados por un motor de búsquedas. Este formato difiere ligeramente del formato conceptual, sobre todo por motivos de optimización de los algoritmos o del almacenamiento.

Conceptualmente se tienen 2 elementos, que son el vocabulario $V$ y un conjunto de posting lists $l[p]$ (una por término en el vocabulario). A su vez, una $l[p]$ esta conformado por dos listas: una lista de identificadores de documentos $docId$ y otra de frecuencias asociadas a dichos identificadores $f$.

Es posible optar por tener el conjunto de $l[p]$ separado en 2 archivos, uno que represente las listas de $docId$ y otro que contenga las listas de $f$, siempre que sea posible recuperar la correspondencia entre ambos (Figura 3.5).

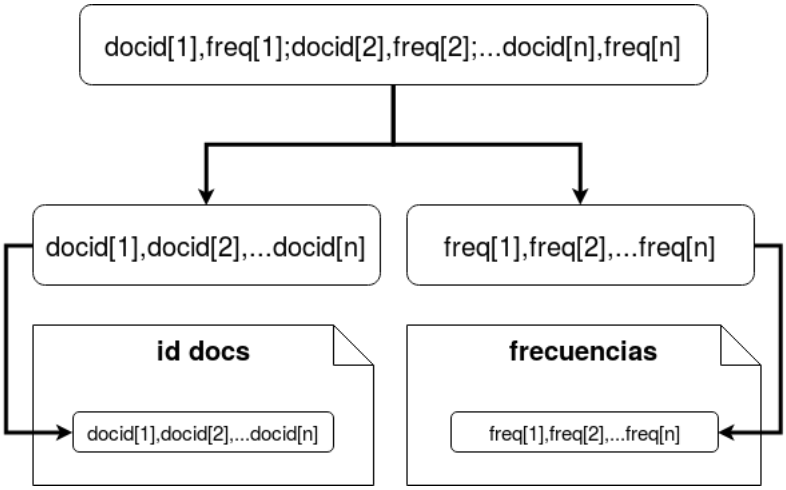



Figura 3.5: Implementación de la posting list en 2 listas separadas

Este ultimo esquema físico es el elegido en la implementación del índice básico. Esto tiene como consecuencia que al final del proceso de indexado se tienen 3 archivos, uno para $V$ y 2 para el índice como tal. Cabe destacar que estos 2 últimos archivos son almacenados en formato binario, mientras que el $V$ es un archivo de texto. Además el par de archivos del índice tienen 2

versiones: Una comprimida y una sin compresión. En esta sección se especifica el formato para la versión no comprimida.

**Vocabulario**

El registro del vocabulario para el índice que no se comprime tiene el siguiente formato:

```
término:tamaño posting list:byte inicio posting list\n
```

El caracter de dos puntos separa los campos del registro. el caracter de nueva linea es el indicador de fin del registro. El término se ubica como primer campo. El segundo campo es la cantidad de elementos de la posting list. El tercer campo indica el byte de inicio de la $l[p]$ tanto en el archivo de identificadores como en el archivo de frecuencias. Al ser la misma cantidad de elementos, la correspondencia esta garantizada y el byte de inicio es idéntico para ambos archivos.

**Archivos de índice**

Desde el punto de vista físico, los 2 archivos que representan el índice invertido almacenan una secuencia de enteros de 32 bits. Esto limita la cantidad de documentos a procesar y la frecuencia de aparición de un término al valor $2^{32} - 1$. Dentro de estos archivos no hay ningún tipo de valor de separación ni de control.

### Formato Físico del Índice con Compresión

Almacenar el índice en un formato comprimido implica agregar una estructura adicional a las $l[p]$. Al utilizar para comprimir los datos técnicas de compresión de enteros que comprimen en bloques, es necesario representar estos bloques en el archivo.

Las técnicas de compresión de números funcionan de la siguiente forma: Reciben bloques de números de tamaño fijo y devuelven una lista de números (habitualmente de menor tamaño que la original). El bloque de números debe ser provisto al algoritmo de compresión en formato de delta-gaps. Esta lista de números es la que se almacena finalmente.

Sin embargo, esta transformación que se realiza mediante la compresión hace que el numero de enteros que se obtiene como resultado sea de tamaño variable. Dicho de otra manera, el tamaño de la lista de enteros comprimida no depende del tamaño de la lista de entrada. Esto genera que se debe contar con una cabecera de bloque la cual contiene el tamaño del bloque (Figura 3.6).

**Vocabulario**

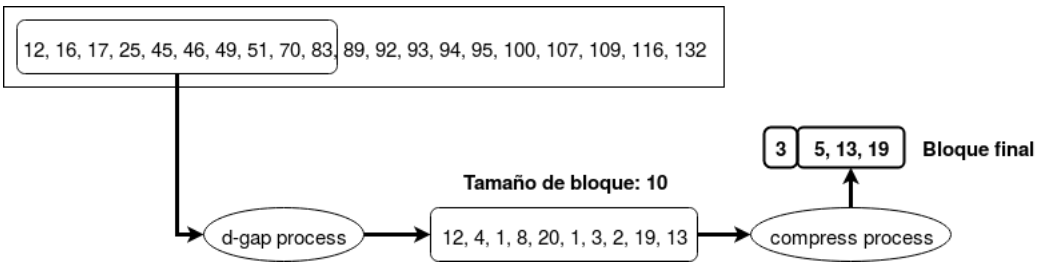



Figura 3.6: Compresión de bloque

El registro del archivo $V$ para el índice comprimido es similar al registro para la versión sin compresión, pero al contener estructuras de tamaño variable debe manejar por separado el inicio de la posting list en cada uno de los archivos.

```
término:tamaño posting list:inicio doc ids,inicio frequencies\n
```

Se agrega en esta estructura un separador adicional representado mediante el caracter coma para separar elementos dentro de un campo. Los elementos *inicio doc ids* e *inicio frequencies* representan el numero de byte a partir del cual comienza la posting list del término referido.

**Archivos de índice**

Los archivos de índice tienen un formato idéntico entre si, salvo por el hecho de que los bloques que se almacenan lo hacen mediante algoritmos de compresión diferentes. Mientras que los bloques en el archivo de *docId* son comprimidos usando PForDelta, los bloques de $freq$ son almacenados utilizando el algoritmos Simple16.

El formato de ambos archivos es el siguiente:

```
posting file = <block[1]><block[2]>...<block[n]>
block[i] = <block header><block body>
block header = <block size>
block body = <element[1]><element[2]>...<element[n]>
```

Tanto el campo `<block size>` como los diferentes elementos del cuerpo del bloque son enteros de 32 bits. El tamaño máximo del bloque implementado es de 128 elementos. El bloque comprimido tiene el mismo límite, aunque resulta esperable que la cantidad de enteros luego de la compresión sea menor.

### 3.1.2. Algoritmo para Índice Block-Max

El índice Block-Max tiene una implementación que requiere su almacenamiento mediante bloques de números. Esto genera que el almacenamiento sea muy similar a la forma en la cual el índice básico es almacenado cuando es comprimido. Sin embargo, Block-Max requiere que la cabecera de bloque contenga mayor cantidad de datos que el bloque presentado anteriormente.

#### Consideraciones

A diferencia del índice básico, el cual era implementado en 3 archivos, se decide implementar el índice Block-Max uniendo las $l[p]$ correspondientes a los $docId$ y las $freq$.

Los campos que conforman el vocabulario son los siguientes:

```
<Término> <TamañoPosting>
```

Donde:

- Término: El término indexado.
- TamañoPosting: La cantidad de postings de la posting list. Coincide con el $DF$ del Término.

La estructura de postings es diferente al baseline, como es de esperarse, por la introducción del bloque como estructura dentro del índice:

```
Posting File = <PostingList[1]> <PostingList[2]> ... <PostingList[N]>
Posting List = <Término> <Bloque[1]> <Bloque[2]> ... <Bloque[N]>
Bloque = <CabeceraBloque> <Lista[IdDocumento]> <Lista[Frecuencia]>
CabeceraBloque = <MaxDocIdBloque> <MaxFreqBloque>
Lista[Elemento] = <Elemento[1],...,Elemento[n]>
```

Donde:

- IdDocumento: Un identificador de documento que permita ubicarlo unívocamente dentro de la colección
- Frecuencia: Cantidad de apariciones de `<Término>` en `<IdDocumento>`
- MaxFreqBloque: La frecuencia que tiene el mayor valor dentro del bloque (Requerido por Block-Max)
- MaxDocIdBloque: El IdDocumento correspondiente al `<MaxFreqBloque>` (Requerido por Block-Max)
- `len(Lista[Elemento]) == n` es el tamaño del bloque. En todo nuestro trabajo es de 128 elementos [13].

### Implementación en MapReduce

Los detalles de implementación comentados en 3.1.1 se aplican a esta implementación de la misma forma. Por la manera en que se diseñaron los algoritmos, solo el OutputFormat y su correspondiente RecordWriter son diferentes a los del índice básico.

Este diseño asegura que se puedan implementar otros índices mediante la introducción de estas 2 clases modificadas y reutilizar el resto del flujo de trabajo implementado. Por otro lado, y al igual que sucede en el algoritmo básico, cuando la cantidad de reducers es mayor a uno, se tiene como resultado n archivos de salida, uno por reducer.

### Formato Físico del Índice sin Compresión

A continuación se define cual es el formato físico del índice, o de otro modo, la forma en que los datos son almacenados para que puedan ser utilizados por un motor de búsquedas. Este formato difiere ligeramente del formato conceptual.

Como se describe en las secciones precedentes, se tienen el vocabulario $V$ y un conjunto de posting lists $l[p]$ (una por término en el vocabulario). Para el índice Block-Max, la $l[p]$ esta conformado por un conjunto de bloques que contiene 2 sub listas: una lista de identificadores de documentos $docId$ y otra de frecuencias asociadas a dichos identificadores $f$.

A diferencia del índice básico, la decisión de implementación adoptada es mantener el bloque completo en el mismo archivo. Esta decisión implica que dicho bloque esta constituido por una cabecera y un cuerpo, donde dicho cuerpo contiene 2 partes, una lista de elementos $docId$ y la otra parte es la lista de $freq$ (Figura 3.7).

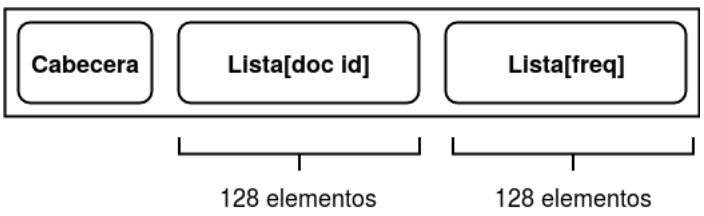



Figura 3.7: Estructura de un bloque Block-Max

#### Vocabulario

El registro del vocabulario para el índice que no se comprime tiene el siguiente formato:

```
término:tamaño posting list\n
```

El caracter de dos puntos separa los campos del registro. el caracter de nueva linea es el indicador de fin del registro. El término se ubica como primer campo. El segundo campo es la cantidad de elementos de la posting list. Al tener bloques de tamaño fijo y la longitud de la posting list, es posible calcular el inicio de cada $l[p]$ en el momento en que el motor de búsqueda cargue el vocabulario.

**Archivos de índice**

El archivo de índice sin compresión tiene un formato similar al índice con compresión del algoritmo básico (ver 3.1.1). Sin embargo, la diferencia fundamental es que los bloques pueden computarse sin la necesidad de almacenar apuntadores. El tamaño de bloque es siempre igual al tamaño máximo de bloque, salvo el ultimo bloque de una posting list, que tiene un tamaño de <tamaño posting list> mod <tamaño máximo bloque>. En el caso de los bloques comprimidos, el problema es que la lista comprimida no tiene un tamaño fijo para la misma cantidad de elementos de entrada.

El formato de ambos archivos es el siguiente:

```
posting file = <block[1]><block[2]>...<block[m]>
block[i] = <block header><block body>
block header = <maxDocId><maxFreq>
block body = <list[docIds]><list[freqs]>
list[data] = <element[1]><element[2]>...<element[n]>
```

Todos los elementos que conforman los bloques son enteros de 32 bits. El tamaño máximo del bloque implementado es de 128 elementos. Esto significa que un bloque completo tiene 128 $docIds$, 128 $freq$ y 2 elementos en la cabecera, lo que es un total de 258 enteros.

### Formato Físico del Índice con Compresión

El formato físico del índice Block-Max comprimido es muy similar a la estructura física del índice básico comiprimido (Ver 3.1.1).

**Vocabulario**

El registro del archivo $V$ para el índice comprimido es similar al registro para la versión sin compresión, pero al contener estructuras de tamaño variable debe manejar por separado el inicio de la posting list en cada uno de los archivos.

```
término:tamaño posting list:byte inicio primer block\n
```

**Archivo de índice**

El archivo de índice es idéntico al de Block-Max sin compresión, salvo por el hecho de que al ser un bloque comprimido, tiene una longitud de elementos variable. Por esto es necesario agregar a la cabecera del bloque el largo de cada sección del mismo.

El formato físico para este índice es el siguiente:

```
1 posting file = <block[1]><block[2]>...<block[m]>
2 block[i] = <block header><block body>
3 block header = <maxDocId><maxFreq><size docId list
     ><size freq list>
4 block body = <list[docIds]><list[freqs]>
5 list[data] = <element[1]><element[2]>...<element[n
     ]>
```

Los diferentes elementos del cuerpo del bloque son enteros de 32 bits. El tamaño máximo del bloque implementado es de 128 elementos. El bloque comprimido tiene el mismo límite, aunque resulta esperable que la cantidad de enteros luego de la compresión sea menor.

## 3.2. Patrones de Diseño en la Implementación

La propuesta original [11] de indexación distribuida con MapReduce plantea un enfoque novedoso en cuanto al procesamiento distribuido pero tal como es presentada en esa primer versión tienen algunos problemas que se deben considerar:

- La utilización de Combiners separados no esta garantizada y es ineficiente [31].
- Se desaprovecha la capacidad de ordenamiento distribuido usando como clave datos atómicos.

Existen una serie de patrones diseñados para que los algoritmos desarrollados sobre Hadoop mejoren su rendimiento en dichos aspectos y que fueron utilizados en las implementaciones de este trabajo. A continuación se explican dichos patrones para poder entender que aportan al proceso general.

### 3.2.1. Value-to-Key Pattern

En la propuesta original de indexación con MapReduce [11], se plantea una generación de registros desde los Mappers con los siguientes campos:

```
<Término, Id Documento>
```

Este enfoque es ineficiente debido a que un término dado que aparece N veces en el documento que se esta procesando genera el mismo registro N veces. 2 soluciones se proponen a este problema. La primera es la utilización de un Combiner. Sin embargo la utilización de Combiners no esta garantizada por la plataforma [49].

Un segundo enfoque de mejora es contar dentro de cada Mapper la cantidad de veces que aparece el término y agregarlo como un campo dentro de una estructura en el valor:

```
<Término, posting(Id Documento, Frecuencia)>
```

Esto mejora el proceso generando una menor cantidad de registros como salida de las tareas map. Sin embargo tiene la dificultad de que en la fase reduce se necesitan cargar en memoria todos los valores de los registros para ordenarlos por el identificador de documento, que es la forma en que el índice invertido es construido.

MapReduce permite ordenar la clave de los registros generados durante la fase Map en la fase Shuffle y lo hace de forma distribuida. El patrón value-to-key [31] propone crear claves compuestas que sean ordenadas por mas de un campo y eso permite ahorrar trabajo para la tarea que el Reduce necesita realizar aprovechando una operación de ordenamiento masiva y distribuida. Entonces, la propuesta es hacer el siguiente cambio en la estructura que genera el Mapper:

```
<Cláve(Término,IdDocumento), Frecuencia>
```

Los Reducers siguen recibiendo los registros ordenados por término. Pero ahora para el mismo término también llegan ordenados por identificador de documento. Para lograr esta estructura compuesta que no es nativa de Hadoop el tipo de dato usado como clave es una clase desarrollada para este trabajo llamada KeyPair. Para que Hadoop pueda utilizarlo como estructura de datos y aparte permita ordenar primero por término y luego por identificador de documento hay que implementar en dicha clase las interfaces `Writable` y `WritableComparable`.

La interfaz `WritableComparable` le provee a Hadoop la posibilidad de poder ordenar las claves en base a un criterio definido por el desarrollador.

La función de comparación recibe 2 registros y compara primero en base a los campos términos. Si son iguales compara los campos identificador de documento de ambos registros. La salida debe ser -1, 0 y 1 si el primer parámetro es menor, igual o mayor al segundo parámetro (Algoritmo 1).

**Algoritmo 1:** Pseudo-código para la función de comparación de registros.

**Entrada:** keyPar: Registro generado por un map
anotherKeyPar: Otro registro a comparar
**Salida** :-1, 0 o 1 si keyPar es menor, mayor o igual a anotherKeyPar.

```
1 Funcion comparar (keyPar, anotherKeyPar)
2     aTerm ← aKeyPar.getTerm();
3     anotherKeyPar ← anotherKeyPar.getTerm();
4     comparacion ← sonIguales(aTerm, anotherTerm);
5     si comparacion = IGUALES entonces
6         aDocId ← aKeyPar.getDocId();
7         anotherDocId ← anotherKeyPar.getDocId();
8         comparacion = sonIguales(aDocId, anotherDocId);
9     devolver comparacion;
```

En conclusión, el patrón value-to-key permite utilizar el ordenamiento que tiene lugar en la fase Shuffle de Hadoop para que los registros lleguen ordenados por término e identificador de documento.

La creación de claves compuestas proporciona otra ventaja adicional, que no es utilizada en este trabajo pero vale la pena destacar porque dada la implementación actual, permite pensar en futuras lineas de investigación. En la generación de índices distribuidos, una decisión de diseño es como se procesa la colección a lo largo del cluster. Los dos enfoques básicos son particionar por documento (cada nodo procesa un sub conjunto de documentos y almacena la porción del índice correspondiente a dichos documentos) o particionar por término (todas las postings de un mismo término son procesadas por el mismo nodo).

En la propuesta de indexación con Hadoop del presente trabajo, la decisión de como particionar viene dada por la forma en que Hadoop genera la partición de claves, vía la implementación de la clase Partitioner. En dicha implementación, la partición se hace por documentos.

Sin embargo haciendo uso del concepto de partición en Hadoop y aprovechando el uso del patrón value-to-key es posible particionar por el otro elemento de la clave, es decir, por el documento.

Algo que tampoco se hace en este trabajo pero que es factible de ser implementado y estudiado en trabajos futuros, es la implementación de índices bi-dimensionales mediante algunas modificaciones al particionador y al reducer. Los índices bi-dimensionales plantean combinar las ventajas de los dos enfoques de partición en un mismo índice [15]. De esta forma, pensando al cluster como una matriz de nodos de procesamiento, a nivel de filas tiene el índice particionado por términos y a nivel de columnas el índice particionado por documentos.

### 3.2.2. In-Mapper Combiner Pattern

Este patrón esta relacionado con el uso de Combiners durante la fase Map de un Job Hadoop. Un Combiner es presentado como un proceso que se realiza por worker después de la ejecución de las tareas Map [11]. El objetivo del Combiner es recolectar claves idénticas de los datos emitidos por los Maps y lograr una agregación local antes que los mismos sean transmitidos por la red.

La idea de utilizar esta estrategia es evitar los cuellos de botella en las escrituras en almacenamiento secundario y las transferencias de red haciendo una agregación local parcial por worker. Como resulta claro, el Combiner mejora el rendimiento general pero su output debe ser idéntico en cuanto al tipo de datos que el Map que se ejecutó anteriormente. Por otro lado, al ser una optimización del proceso, el framework decide frente a cuanta cantidad de pares emitidos es conveniente ejecutar un Combiner, dando lugar a una mejora parcial en la generación de los registros.

El patrón Combiner in-mapper plantea que esa agregación se puede hacer por Map, retrasando el envío de claves hasta que todo un documento se encuentre procesado [31]. Así, en lugar de generar un registro por cada término, se puede emitir por cada término único en cada documento. Por Ley de Zipf, la mayoría de los documentos repite con mucha frecuencia unos pocos términos, por lo que se puede esperar una reducción sensible en la cantidad de registros emitidos [11].

El riesgo que conlleva este patrón es el de escalabilidad. Al tener una estructura en el proceso Mapper que retiene la generación de registros a la espera de procesar todo un documento, si la estructura donde se almacene la información supera la cantidad de memoria principal asignada al proceso Map, dicho documento no podrá ser procesado. En general, es esperable que documentos típicos de la web de los que solo se procesa el contenido de texto no superen una cantidad de bytes manejable, aunque debe considerarse la decisión de diseño frente a problemas que puedan surgir en el procesamiento.

La estructura que acumula la información del documento es una estruc-

tura asociativa término - frecuencia. La misma entonces representa un histograma de frecuencias para los términos dentro del documento procesado.

Este patrón permite optimizar y mejorar el rendimiento de las aplicaciones MapReduce en la fase Map. Evita la generación de pares intermedios innecesarios que son motivos de cuello de botella en operaciones de entrada/salida a disco y transferencia de red.

## 3.3. Propuesta para los Algoritmos de Indexación

A partir de los patrones desarrollados a lo largo de este capitulo, se enuncia a continuación la propuesta de motor de indexación que se implementa como algoritmo MapReduce. Los algoritmos de las fases Map (Algoritmo 2) y Reduce (Algoritmo 3) que se presentan en esta sección son idénticas para los algoritmos de construcción de índices Básico y Block-Max. Además resulta oportuno realizar una comparación global de la propuesta aquí presentada con otras dos propuestas de indexación distribuida con MapReduce.

**Algoritmo 2:** Pseudo-código para la función MAP del motor de construcción de índices.

**Entrada:** Clave: Numero de linea del archivo de entrada, *lineNumber*
Valor: Un documento en formato Trec, *documentContent*
**Salida :** Clave: Par de término y id documento, *par*
Valor: Frecuencia del término en el documento, *freq*

```
1 TokenizedDocument dt ← new Tokenizer(documentContent);
2 PartialPostingList partialPL ← new PartialPostingList();
3 mientras dt.hasMoreTokens() hacer
4 |   String term ← dt.nextToken();
5 |   partialPL.addPosting(term, dt.getDocId(), 1);
6 para cada posting in partialPL hacer
7 |   Par par ← posting.getKey();
8 |   Integer freq ← posting.getValue();
9 |   emit(par, freq);
```

### 3.3.1. Detalles de la Implementación

La implementación propuesta en este trabajo es un algoritmo MapReduce cuyo pseudo-código esta mostrado en los algoritmos 2 y 3.

**Algoritmo 3:** Pseudo-código para la función REDUCE del motor de construcción de índices.

**Entrada:** Clave: Par de valores término documento, *par*
Valor: Lista de frecuencias, *frequencies*
**Salida :** Clave: Término, *term*
Valor: Posting list del término, *postingList*

```
1 Text term ← par.getTerm();
2 Integer totalFreq ← 0;
3 para cada freq in frequencies hacer
4 |  totalFreq ← freq + totalFreq;
5 PostingList postingList ← getPostingList();
6 Integer docId ← par.getDocId();
7 postingList.addPosting(docId, totalFreq);
8 si nextTerm() ≠ term OR not(hasMoreTokens()) entonces
9 |  emit(term, postingList);
```

La función `map()` (Algoritmo 2) se encarga principalmente de la extracción de términos de los documentos de la colección. Dichos documentos son recibidos como entrada de la función, y una clase Tokenizer determina que tokens de cada documento son validos para convertirse en términos del índice.

Para cada término disponible, se calcula su frecuencia relativa y se almacena dicho cálculo en una estructura conocida como posting list parcial, que es una posting list acotada al documento que se esta procesando. Una vez procesado completamente el documento, la posting list parcial es procesada y se construyen los registros a ser emitido con el formato clave-valor, donde la clave es un campo de la clase `KeyPar` (Ver 3.2.1). El valor es la frecuencia acumulada del término en el documento procesado.

Una desventaja del algoritmo 2 es que la estructura `PartialPostingList` se almacena en memoria. Se asume con este enfoque que los documentos de texto plano recuperados de la web no requieren grandes cantidades de RAM. La estimación a nivel mundial es que el tamaño en bytes de un sitio web promedio pesa 3 MB[3], contando elementos multimedia y texto.

En el caso que se necesiten procesar documentos que superen el limite de memoria disponible por proceso el enfoque adoptado para la construcción de las claves permite dividir el documento en varias partes y que el mismo sea procesado en paralelo por varias funciones `map()`. La fase de Shuffle se

[3]Estimación realizada por el sitio `http://httparchive.org/trends.php#bytesTotal&reqTotal`, consultado en Julio de 2017

encargara de reunir los registros de idéntica clave (término-documento), y la función `reduce()` recibe los datos de forma integrada, sin preocuparse si el documento en cuestión fue procesado por uno o mas procesos en paralelo.

Todo esto es posible debido al esquema de construcción de claves elegido(Ver 3.2.1). Mientras se mantenga esa estructura en la entrada/salida de cada función, el framework garantiza que el resto de las funciones siguen respondiendo de la misma forma y de esta manera se observa como la forma de trabajo de la plataforma acompaña el crecimiento de la aplicación.

El algoritmo 3 recibe las frecuencias para un mismo término en los diferentes documentos. Construye una estructura de la clase `PostingList`, y una vez procesados todos los documentos para el término actual, emite un registro con el término como clave y la `PostingList` como valor.

El objeto `PostingList` tiene funciones útiles para el procesamiento posterior. Además de almacenar los documentos donde aparece un término junto a sus frecuencias asociadas, el objeto permite iterar sobre las postings y separarlas en bloques, entre otras funciones. Por eso es preferible a una lista de números o incluso a una estructura de tipo diccionario o hash.

### 3.3.2. Diferencias entre Algoritmos Básico y Block-Max

Como se desprende de la explicación de los algoritmos `map()` (Algoritmo 2) y `reduce()` (Algoritmo 3) implementados, los mismos no intervienen en ningún momento en la construcción del archivo de índice final. Esto permite reutilizar ambas funciones en los dos tipos de algoritmos.

La pregunta que surge entonces es en que parte del proceso se realiza la escritura del archivo final. La fase de Output tiene por objetivo implementar la salida física del algoritmo. Para ello, se involucran 2 clases: `FileOutputFormat` y `RecordWriter`. Esta ultima clase es la que implementa el formato especifico del archivo.

La implementación dispone de 4 clases que heredan de la clase base `RecordWriter`:

- `BaselineRecordWriter`
- `BaselineCompressRecordWriter`
- `BlockMaxRecordWriter`
- `BlockMaxCompressRecordWriter`

Cada una implementa un formato especifico de salida. Para su utilización, debe especificarse en tiempo de ejecución vía parámetros al momento de correr el algoritmo. De esa manera, Hadoop crea la instancia de la clase correspondiente. El `RecordWriter` correspondiente construye los registros del vocabulario y del archivo de índice, con el formato que corresponda según el caso.

### 3.3.3. Otros Enfoques de Construcción de Índices

La propuesta original de MapReduce [11] propone que la función `map()` emita pares `<término,docId>`. Esto significa que el algoritmo propone emitir un registro de salida por cada ocurrencia de un término en todos los documentos. Este enfoque es el mas sencillo de desarrollar, pero completamente ineficiente desde el punto de vista de uso de los recursos (principalmente los relacionados con la transferencia en la red y al almacenamiento secundario).

Por otro lado, la función `reduce()` debe procesar todas las ocurrencias para un término dado realizando un sort en memoria, dado que la entrada a la función es `<término, list(docIds)>`.

La forma de procesamiento propuesta por Dean y Ghemawat [11] genera funciones `map()` que requieren muy poco procesamiento y memoria y en contraste funciones `reduce()` con una alta necesidad de uso de memoria y capacidad de procesamiento.

Posteriores discusiones mejoran el enfoque planteado. McCreadie y otros proponen una mejora al enfoque original y por otro lado otro enfoque novedoso en su momento [35]. Para el enfoque original, proponen mejorar la estructura que emite la función `map()` agregando la frecuencia del término en el documento (conocida como *tf*), quedando la estructura de salida con el formato `<término, (docId, tf)>`. A su vez, presentan un algoritmo que denominan "Estrategia de indexación MapReduce de un único paso" (Single-pass MapReduce Indexing Strategy).

El enfoque de McCreadie [35] plantea que no es necesario emitir registros en las funciones `map()` hasta que el proceso no pueda utilizar mas memoria. Cada map procesa tantos documentos como pueda, construyendo un índice parcial, y una vez que la memoria se termina, emite tantos pares `<término, posting list>` como tenga procesados. Es tarea de la fase reduce realizar la mezcla ordenada de las posting lists.

Si en el enfoque de Dean y Ghemawat [11] se emiten muchos registros muy pequeños, la propuesta de McCreadie [35] es generar una sensiblemente menor cantidad de salidas en la fase map, pero cada salida tiene una cantidad de información considerablemente mayor.

La propuesta de este trabajo se ubica en un punto intermedio entre ambas propuestas. Cada salida emitida en la fase map contiene toda la información de un término para un documento dado. El enfoque mejora notablemente el uso de recursos en la fase map respecto a la forma "ingenua" en que el tema es encarado en [11]. Por otro lado, la utilización del patrón value-to-key [31] en este trabajo permite que la construcción de las posting lists completas en la fase reduce sean mucho mas simples que en la propuesta de McCreadie [35], donde el enfoque necesitaba, además de la información propia de las posting lists parciales, información sobre cual proceso map había realizado la emisión y en que orden se había realizado. El enfoque planteado en el trabajo actual evita esos inconvenientes, y la posting list es construida directamente en la fase reduce procesando los datos en el orden en que son recibidos.

## 3.4. Métodos de Compresión

En el presente trabajo se utilizan 2 métodos de compresión. Ambos usan estrategias de compresión de enteros por bloques. Esto significa que para una lista de enteros, la compresión se realiza armando subconjuntos de tamaño fijo de los datos. Por ejemplo, Si se tiene un bloque cuyo tamaño es 64 elementos, y una lista de 80 números, la compresión se aplica sobre los primeros 64 números, y luego se crea otro bloque a partir de los 16 enteros restantes.

Para bloques de identificadores de documentos ordenados como lista de enteros, el estándar de compresión seleccionado es PForDelta. Para las frecuencias, cuyo patrón es secuencias de números no ordenadas pero de valor relativamente bajo, se prefiere Simple16 [8]. Se utiliza para ambos casos la implementación de Lemire para cada uno de los respectivos algoritmos [29].

Por otro lado, Hadoop ofrece métodos de compresión de forma nativa. En 2 etapas del flujo de trabajo de MapReduce se pueden utilizar estas técnicas. En la salida de los procesos Map es posible comprimir los pares intermedios que serán consumidos por la fase Shuffle. La salida final del proceso MapReduce también puede ser comprimida. En todos los casos, Hadoop realiza dicho proceso de forma transparente al desarrollador, configurando una serie de parámetros de configuración del Job. Sin embargo, en este trabajo no se utilizaron ninguna de estas características ofrecidas por el Framework.

En el caso de la compresión final, los algoritmos utilizan métodos de compresión PForDelta y Simple16, por lo cual no resulta necesario comprimir nuevamente. Para utilizar estas técnicas, primero se deben computar los delta-gaps de las listas de números. Una vez que se obtiene la lista de delta-gaps, el problema se transforma en buscar estrategias eficientes de compresión para listas de números pequeños. Este tipo de técnicas es un tema amplia-

mente tratado en la academia y existen muchos algoritmos que hacen esto de forma eficiente.

Representar una lista de números en formato de delta-gaps tiene desventajas. Si la lista de enteros es demasiado grande y se necesita un valor N intermedio de la lista, obtenerlo implica computar los N-1 valores anteriores. El enfoque mas común para resolver esto es dividir la lista de enteros en bloques, y computar los saltos al interior de cada bloque.

Las dos técnicas de compresión utilizadas asumen que la posting list esta dividida en bloques, y cada bloque tiene sus elementos computados con delta-gaps. Esto es útil en ciertos tipos de índices, como block-max, que naturalmente construye su estructura de índice a partir de bloques, y de forma natura puede aplicarse técnicas de compresión en el proceso de construcción del índice.

# Capítulo 4

# Experimentos y Resultados

Este capítulo presenta y desarrolla el ambiente, datos y herramientas con los cuales se realizan las pruebas de los algoritmos explicados y se analizan los resultados de los mismos. Específicamente se tratan los siguientes temas:

- Datasets utilizados: Se explican y caracterizan los datos utilizados en las pruebas.
- Cluster de pruebas: Se describen los nodos que componen el cluster, la configuración de los mismos relacionada con el procesamiento de los datos y otros detalle de hardware relevantes.
- Métricas: Se construyen las métricas con las cuales se comparan los diferentes experimentos.
- Análisis de experimentos: Exposición de resultados y análisis.

## 4.1. Datasets

Cada Dataset es una colección entera sobre la cual se aplica todo el proceso de construcción de indices ya explicado. Por lo tanto se dispone de 3 colecciones sobre las que se realizan mediciones y pruebas. En lo que sigue los términos colección o Datasets son utilizados como sinónimos.

### 4.1.1. Origen

Tres Datasets fueron utilizados para realizar las pruebas de este trabajo. El primero es un dataset "original" y los restantes fueron construidos a partir del primero.

El primer Dataset es un crawl derivado del proyecto WebBase de la Universidad de Stanford [23]. Esta colección es referenciada con el nombre C1.

El segundo Dataset se crea con el siguiente procedimiento: Se construye un nuevo archivo con el contenido de C1 por duplicado (Utilizando el comando cat de Linux y pasando C1 como parámetro dos veces). Luego se ejecuta un script que renumera los documentos. De esta manera los documentos duplicados tienen ahora un identificador distinto y constituyen elementos diferentes a los fines del proceso de creación del índice. Con este procedimiento se busca tener una colección del doble de tamaño de la original y se denomina C2.

La tercera colección (llamada C3 para este trabajo) se crea de manera análoga a C2 pero triplicando C1 en lugar de duplicarlo y renumerando los documentos, quedando una colección del triple de tamaño que la original.

### 4.1.2. Caracterización y Estadísticas

Las colecciones construidas tienen un conjunto de estadísticas que pueden resultar de interés para tener una dimensión de los datasets utilizados. Algunas de estas características se pueden observar en la Tabla 4.1.

| Estadísticas | C1 | C2 | C3 |
|---|---|---|---|
| **Tamaño de la colección** | 64,08 GB | 128,17 GB | 192,25 GB |
| **Cantidad de documentos** | 7077922 | 14155844 | 21233766 |
| **Tokens en la colección** | 5008412936 | 10016825872 | 15025238808 |
| **Términos Diferentes** | 34370021 | 34370021 | 34370021 |
| **Tokens promedio por documento** | 708 | 708 | 708 |

Tabla 4.1: Estadísticas de la colección

### 4.1.3. Formato de la Colección

El formato de la colección es tomado de un formato utilizado en las conferencias TREC. Por convención, todos los documentos de la colección se encuentran almacenados en el mismo archivo. El formato básico para cada documento es el siguiente:

```
<DOC>
```

```
<DOCNO> Doc Id </DOCNO>
CONTENT
</DOC>
```

Este formato coincide con el que genera la salida del proceso WebParser del software Lemur[1]. Por simplicidad, se procesó la colección de tal manera que todo el contenido entre las etiquetas `DOC` de apertura y cierre no tengan saltos de linea. Esto genera que se tenga un documento por linea del archivo de la colección. Con este formato y organización, se puede utilizar sin modificaciones la clase `TextInputFormat`[2] que es provista por Hadoop. Esta clase garantiza que cada proceso Map tendrá como entrada una linea del archivo provisto. Al coincidir cada linea con un documento, se garantiza que cada proceso Map tiene como entrada un documento entero sin tener que agregar ninguna linea de código. La decisión adoptada hace menos flexible al algoritmo, pero a la vez mas sencillo de entender.

## 4.2. Construcción del Cluster de Pruebas

Al momento de realizar las pruebas de los algoritmos, se tuvo que evaluar la posibilidad de construir un cluster en la nube de algún proveedor o crear un cluster propio, de recursos modestos pero libre de costos económicos y que permita acumular una experiencia en un área poco desarrollada en la Universidad de Luján[3]. Se encontró la posibilidad de realizar una Vinculación con el Centro de Investigación, Docencia y Extensión en TICs (CIDETIC)[4] de dicha institución en calidad de alumno que realiza su Trabajo Final de Carrera.

El Centro pone a disposición un conjunto de equipos de usuario final sobre los cuales se instala Hadoop, seleccionando uno de los equipos como nodo master y el resto como slaves o workers. Las características del Hardware utilizado es la siguiente:

- 7 equipos de usuario final. 1 utilizado como master, 6 como workers
  - CPU: Intel Core i5 2400 3.10GHz
  - RAM: 8 GB

[1] `http://www.cs.cmu.edu/~lemur/LemurGuide.html#data`

[2] `https://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/TextInputFormat.html`

[3] `http://www.unlu.edu.ar/`

[4] `https://cidetic.unlu.edu.ar/`

  - Red: Gigabit Ethernet
  - Disco: HDD 500 GB

- Switch DLink Modelo DGS-3120-24TC-SI Gigabit Stackable L2/L3

El Sistema Operativo instalado en los equipos es Ubuntu 14.04 LTS y sobre los mismos se instala la versión 2.5 de Hadoop.

### 4.2.1. Instalación

Existe documentación diversa y amplia sobre la forma de instalación de Hadoop. En esta sección solo se mencionan los pasos mas importantes para realizar dicha instalación.

Lo primero que se debe hacer es descargar la versión mas reciente de Hadoop. Esta descarga es recomendable hacerla siempre desde el sitio oficial[5]. De las opciones disponibles, es recomendable descargar la versión estable mas actual disponible. En general se ofrece un archivo de extensión tar.gz que contiene los binarios de Hadoop.

A partir de acá, las instrucciones se aplican a todos los nodos, salvo que se indique lo contrario.

Una vez descargado, se descomprime el archivo y se ubica su contenido en la carpeta donde Hadoop se instalara.

```
1 tar xfz hadoop-2.5.0.tar.gz
2 sudo su
3 mkdir /usr/local/hadoop
4 cp -R hadoop-2.5.0/* /usr/local/hadoop
```

Como en otros servicios, se deben crear un usuario y grupo del sistema con los cuales se ejecutaran los procesos que se realicen con Hadoop. Dicho usuario y grupo deben ser dueños de los archivos descomprimidos.

```
1 addgroup hadoop
2 adduser --ingroup hadoop hduser
3 chown hduser.hadoop /usr/local/hadoop -R
```

Se debe instalar en el sistema el JDK correspondiente (Java 7) y el software ssh. Una vez que ssh esta instalado, se debe generar una clave privada para el equipo. Ademas agregar dicha clave a las claves de confianza o claves autorizadas.

[5] `http://hadoop.apache.org/releases.html` Consultada el 19 de Agosto de 2017

```
1 apt install default-jdk ssh
2 su hduser
3 ssh-keygen -t rsa -P '' # Si pregunta alguna cosa,
      dejar en blanco y apretar enter
4 cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

A continuación se deben configurar las variables de entorno para usar Hadoop. Un conjunto de las variables que normalmente se deben configurar es el siguiente:

```
1 export JAVA_HOME=/usr/lib/jvm/default-java
2 export HADOOP_INSTALL=/usr/local/hadoop
3 export PATH=$PATH:$HADOOP_INSTALL/bin
4 export PATH=$PATH:$HADOOP_INSTALL/sbin
5 export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
6 export HADOOP_COMMON_HOME=$HADOOP_INSTALL
7 export HADOOP_HDFS_HOME=$HADOOP_INSTALL
8 export YARN_HOME=$HADOOP_INSTALL
9 export HADOOP_COMMON_LIB_NATIVE_DIR=
     $HADOOP_INSTALL/lib/native
10 export HADOOP_OPTS="-Djava.library.path=
     $HADOOP_INSTALL/lib"
```

Habitualmente estas lineas deben ir en el archivo `.bashrc` del usuario `hduser` creado anteriormente. La variable `JAVA_HOME` también debe ser configurada en el archivo `hadoop-env.sh`.

A continuación, los archivos de configuración que se mencionan deben buscarse en la ruta donde se copiaron los archivos de Hadoop, específicamente en el directorio `<HADOOP DIR>/etc/hadoop/`. Se listan a continuación los archivos mas importantes de una instalación y el contenido mínimo que deben tener.

```
1 <configuration>
2    <property>
3       <name>fs.default.name</name>
4       <value>hdfs://master:54310</value>
5    </property>
6    <property>
7       <name>hadoop.tmp.dir</name>
8       <value>/usr/local/hadoop/hadoop_store/tmp</
           value>
9    </property>
```

```
10 </configuration>
```

Texto 4.1: core-site.xml

```
 1 <configuration>
 2    <property>
 3       <name>yarn.nodemanager.aux-services</name>
 4       <value>mapreduce_shuffle</value>
 5    </property>
 6    <property>
 7       <name>yarn.nodemanager.aux-services.
            mapreduce.shuffle.class</name>
 8       <value>org.apache.hadoop.mapred.
            ShuffleHandler</value>
 9    </property>
10    <property>
11       <name>yarn.resourcemanager.hostname</name>
12       <value>master</value>
13    </property>
14 </configuration>
```

Texto 4.2: yarn-site.xml

```
1 <configuration>
2    <property>
3       <name>mapreduce.framework.name</name>
4       <value>yarn</value>
5    </property>
6 </configuration>
```

Texto 4.3: mapred-site.xml

```
 1 <configuration>
 2    <property>
 3       <name>dfs.replication</name>
 4       <value>3</value>
 5    </property>
 6    <property>
 7       <name>dfs.namenode.name.dir</name>
 8       <value>file:///usr/local/hadoop/hadoop_store
            /hdfs/namenode</value>
 9    </property>
```

```
10     <property>
11        <name>dfs.datanode.data.dir</name>
12        <value>file:///usr/local/hadoop/hadoop_store
             /hdfs/datanode</value>
13     </property>
14  </configuration>
```

Texto 4.4: hdfs-site.xml

Deben además crearse los directorios especificados en los archivos anteriormente mostrados:

```
1  mkdir -p /usr/local/hadoop/hadoop_store/tmp
2  mkdir -p /usr/local/hadoop/hadoop_store/hdfs/
      namenode
3  mkdir -p /usr/local/hadoop/hadoop_store/hdfs/
      datanode
```

A continuación se debe asignar a un nodo el rol de master, y el resto de workers. En `<HADOOP DIR>/etc/hadoop/` existen 2 archivos para esto. Uno se llama `masters` y otro `slaves`. El contenido de estos archivos es de una IP o nombre de host por linea.

Una vez configurado los nodos por rol se deben intercambiar las claves ssh entre todos los nodos. Para ello, se hace uso de un comando utilitario que viene con ssh. En todos los nodos se debe ejecutar:

```
1  ssh-copy-id hduser@master
2  ssh-copy-id hduser@slave1
3  ssh-copy-id hduser@slave2
4  ssh-copy-id hduser@slave3
5  ssh-copy-id hduser@slave4
6  ssh-copy-id hduser@slave5
7  ssh-copy-id hduser@slave6
```

El ultimo paso consiste en formatear el sistema de archivos distribuido HDFS desde el nodo master únicamente:

```
1  hdfs namenode -format
```

Una vez que termine el proceso, se pueden iniciar los servicios con los comandos `start-dfs.sh` y `start-yarn.sh`.

Se puede ejecutar el comando `jps` para verificar que los procesos estan corriendo:

```
1 hduser@master $ jps
2 27443 Jps
3 26634 NameNode
4 27056 ResourceManager
```

Texto 4.5: Procesos en el nodo master

```
1 hduser@slaveX $ jps
2 27443 Jps
3 26758 DataNode
4 27150 NodeManager
```

Texto 4.6: Procesos en los nodos workers

Se puede realizar una prueba final ejecutando un algoritmo de pruebas provisto por Hadoop, que calcula el valor de PI por método de Montecarlo vía MapReduce. En el nodo master ejecutar:

```
1 hadoop jar /usr/local/hadoop/share/hadoop/
     mapreduce/hadoop-mapreduce-examples-2.5.0.jar pi
      4 100
```

Si dicho algoritmo se ejecuta adecuadamente y muestra una salida con un valor aproximado de PI, el cluster esta funcionando.

Esta instalación tiene un único inconveniente. la memoria que se utiliza por nodo es extremadamente baja, lo que hace que los recursos del cluster no se utilicen de forma conveniente. En la siguiente sección se discuten detalles de la administración de recursos en un cluster Hadoop.

### 4.2.2. Administración

La instalación por defecto de Hadoop tiene el inconveniente de sub utilizar los recursos a fin de garantizar que el sistema funcione sobre una gran cantidad de configuraciones [40]. Ejecutando algunas pruebas es sencillo identificar que cada nodo instalado dispone de CPU y RAM para que Hadoop utilice en sus tareas.

Un enfoque posible es mirar los parámetros de configuración de memoria y cores disponibles en `yarn-site.xml`[6] y `mapred-site.xml`[7]. Este

[6] `http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-common/yarn-default.xml`

[7] `http://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml`

enfoque no es recomendable si se desconocen las características de la plataforma. Existen muchos parámetros para considerar y se relacionan entre si de formas que no están documentadas (Aproximadamente 190 parámetros) [12].

La experiencia propia indica que es mejor partir de un modelo de distribución de los recursos y desde ahí realizar pequeñas modificaciones que ayuden a mejorar los cuellos de botella que puedan surgir.

El modelo elegido en este caso es el de la plataforma Hortonworks[8], que proveen un script python[9] sencillo que pide como entrada las características de los nodos disponibles y devuelve los valores de los parámetros a configurar.

El uso del script en el cluster propio permitió utilizar en promedio la mitad de los recursos, cuando la configuración por defecto limitaba dicha utilización a menos del 20 %. Hay que tener en cuenta que el modelo del script asume que un porcentaje de los recursos deben quedar para uso del Sistema Operativo. En general suelen reservarse 2 GB y 1 core completo para el sistema, lo que en este caso representó el 25 % de los recursos disponibles.

### 4.2.3. Experiencias

La decisión de construir un cluster propio desde cero, sin contar con experiencia en la plataforma trajo consigo problemas de diversa índole. Breves comentarios se vuelcan aquí como reflexiones luego de algún tiempo de trabajo con el cluster.

La gestión de memoria pude ser muy difícil de llevar a cabo si se desea realizar de forma manual. Hadoop ejecuta tareas Map y Reduce, pero también tiene diversos servicios corriendo de forma concurrente. El servidor de archivos se ejecuta en cada nodo y atiende pedidos. El master despliega servicios conocidos como Applications Managers por cada job, que también consumen memoria y deben ser considerados.

Cuando se llega una configuración estable, Hadoop funciona sin problemas. Incluso ante problemas en un nodo, los algoritmos de la plataforma de recuperación ante errores funcionan correctamente y los trabajos siguen adelante en nodos que se reporten como saludables ante fallas en otros nodos. Agregar nodos al cluster es una tarea sencilla, y no requiere ningún tipo de modificación en los algoritmos para que el nuevo nodo sea utilizado una vez que esta disponible.

A diferencia de los servicios basados en la nube, la disponibilidad puede ser un inconveniente si el datacenter tiene problemas de suministro energético. Para pruebas de cluster como las hechas para este trabajo no es un pro-

[8] `https://hortonworks.com/`

[9] `https://github.com/hortonworks/hdp-configuration-utils`

blema grande, pero para prestar un servicio a terceros debe evaluarse la posibilidad de invertir en infraestructura de alta disponibilidad. En general, salvo para la subida o descarga de archivos, la conexión entre cliente y cluster no suele ser intensiva. La mayor cantidad de operaciones se realiza dentro de la red local del cluster.

Debe tenerse en cuenta que Hadoop no cuenta con herramientas propias de monitoreo del cluster. Si cuenta con drivers que conectan al software de monitoreo Ganglia[10]. En este caso esta fue la opción elegida, porque el datacenter ya contaba con una instalación de Ganglia para el seguimiento de otros clusters.

## 4.3. Experimentos

En esta sección se explican los experimentos que se ejecutaron. Se describen la combinación de parámetros junto con la forma en que se identificaron las pruebas para el calculo posterior.

### 4.3.1. Algoritmos

Se implementaron 2 Algoritmos a los fines del presente trabajo. El primer algoritmo genera un índice en formato tradicional o Baseline y el segundo construye un índice usando Block-Max. La codificación para el Baseline es BL y para Block-Max es BM.

### 4.3.2. Formato

El formato del índice generado indica si es binario o texto. El formato de texto se utilizo para propositos de depuración de los algoritmos y ninguna prueba final fue realizada usando dicho formato. La codificación para el formato binario es B.

### 4.3.3. Compresión

Se puede optar por comprimir el índice resultante o no hacerlo. Si el índice esta comprimido, se identifica mediante el código CC. Por el contrario, si el índice no se comprime se lo identifica como SC. Cuando la opción de compresión es elegida, el índice generado se comprime utilizando el método PForDelta para los identificadores y Simple16 para las frecuencias.

[10] `http://ganglia.sourceforge.net/`

### 4.3.4. Cantidad de Nodos (Mappers / Reducers)

La cantidad de nodos varia para las pruebas de la siguiente manera: Pruebas con 1, 2 4 o 6 Nodos. Para las pruebas locales, se identifico como 0 Nodos. Se identifica como el numero de nodos seguido de una letra N.

La cantidad de reducers esta limitada por la cantidad de nodos, buscando que se ejecute 1 reducer por nodo disponible. De esta manera, para los experimentos que se realizan con 4 Nodos la cantidad de reducers varió entre 1, 2 y 4 reducers. En la codificación de los experimentos se identifica con el numero de reducers seguido de una letra R.

### 4.3.5. Codificación de los Experimentos

El código general para identificar un experimento es el siguiente:

`<Algoritmo><Compresión><Cantidad de Nodos><Cantidad de Reducers><Colección>`

Las colecciones utilizadas se identifican con los códigos C1, C2 o C3 según corresponda.

Entonces, el código de experimento BL-SC-4N-2R-C3 indica lo siguiente:

- Algoritmo Baseline
- Salida sin compresión
- 4 Nodos
- 2 Reducers
- Colección 3 (Colección de 180 GB)

Es posible que se haga referencia a un grupo de experimentos escribiendo un código parcial. En este caso, hacer referencia a los experimentos BL4N hace referencia a todos los experimentos Baseline sobre 4 Nodos. Incluye a los experimentos con o sin compresión de 1, 2 o 4 Reducers y con las tres colecciones.

A veces, para mejorar la legibilidad, se pueden incluir guiones entre cada uno de los campos del código, o quitarlos. Los códigos BLSC4N2RC3 y BL-SC-4N-2R-C3 identifican al mismo experimento.

## 4.4. Métricas

Para el proceso de construcción de índices la métrica mas importante suele ser el tiempo de construcción del índice con los recursos disponibles.

Esto tiene sentido dado que para 2 procesos que construyen el mismo índice y disponen de los mismos recursos, el algoritmo mas eficiente es el que tarda menos tiempo.

Por tratarse de un algoritmo implementado sobre un sistema distribuido resulta de interés medir y analizar características propias de este tipo de software. Dos métricas clásicas son el Speedup [21] y la Eficiencia. El Speedup es una medida de la mejora de rendimiento de la aplicación al aumentar la cantidad de procesadores (comparado con el rendimiento al utilizar un solo procesador). Formalmente se define como:

$$S_n = \frac{T_1}{T_n}$$

Donde:

- $S_n$ es el Speedup del algoritmo ejecutándose en n núcleos
- $T_1$ es el tiempo de ejecución del algoritmo en 1 núcleo o de manera secuencial
- $T_n$ es el tiempo de ejecución del algoritmo en n núcleos

Por su parte, la eficiencia cuantifica el speedup obtenido por procesador y se define como $E(n) = \frac{S_n}{n}$ donde $S_n$ es el speedup utilizando $n$ procesadores.

Además, para ser justos con la comparación se resguarda que la salida del algoritmo secuencial sea exactamente la misma que la de los algoritmos paralelos. Para este problema se asume que la máquina de búsqueda que usa los índices generados está formada por $p'$ procesadores.

Por simplicidad, se asume $p' = p$, por lo cual la cantidad de procesadores para indexar y luego realizar la búsqueda es la misma. Por ello, asumiendo un cluster de $p$ procesadores en cada configuración, el índice centralizado generado por el algoritmo es partido en $p$ partes y cada una transferida a un nodo. Este costo es adicionado al proceso secuencial.

Se define el imbalance en la ejecución de una etapa como el cociente entre el porcentaje de uso de CPU máximo sobre el porcentaje de uso promedio en los nodos que participan del procesamiento para la etapa en particular. Esta métrica es igual a 1 en el caso de un escenario de nodos perfectamente balanceados, lo que resultaría en que el tiempo máximo coincida con el promedio. A medida que la métrica progresivamente se aleja de 1 en forma creciente, resulta en una fuerte indicación que la ejecución de la etapa está dominada por un nodo en particular (no importa cuál).

## 4.5. Análisis de Resultados

La codificación de los experimentos explicada en 4.3.5 indica la manera de denominar a cada uno de los mismos y diferenciarlo de los demás (o a un grupo de ellos).

Para contar la cantidad de experimentos ejecutados, se debe tener en cuenta lo siguiente:

- Se dispone de 2 algoritmos: Block-Max y Baseline
- Cada algoritmo se ejecuta sobre cada una de las 3 colecciones construidas: C1, C2 y C3
- Cada índice generado puede ser comprimido o no: CC y SC
- Los algoritmos se pueden ejecutar de forma local o distribuida. De forma distribuida se dispuso de configuraciones de 1, 2, 4 y 6 nodos
- Para cada configuración de nodos, se dispusieron despliegues de Reducers en paralelo, siempre con el limite según la cantidad de nodos disponibles: 1, 2, 4 y 6 Reducers. El limite consiste en que no existan mas Reducers que nodos disponibles

De todos los experimentos, un subconjunto menor no pudo ser incluido en el análisis por los siguientes motivos: Algunos experimentos no fueron procesados dado que por algún motivo no pudieron finalizarse correctamente. Todos estos casos se refieren al procesamiento de C3 para 1 Reduce teniendo 1 Nodo disponible (1N-1R-C3). Esto significa que todo el peso del archivo subido a HDFS es cargado sobre el único nodo worker del cluster. Esto genera una cantidad de datos que sobrepasa la capacidad de almacenamiento del nodo (500Gb de disco). Estos experimentos no son incluidos en el análisis.

Por otro lado existe un conjunto de experimentos que si bien finalizaron completamente, no se dispone de datos asociados al calculo de las métricas explicadas. Esto se debe principalmente a la forma en que dichos datos son recolectados. Los datos de los experimentos son recuperados vía diversos scripts desde el software de monitoreo Ganglia. En algunos experimentos, Ganglia no registró información alguna o lo hizo de forma inconsistente.

La cantidad total de experimentos planificados es de 132. De estos, se cuenta con datos y fueron procesados 121 (91,67 %).

### 4.5.1. Tiempo de Construcción

El tiempo de construcción es una métrica medida en segundos y que indica el tiempo desde que se inicia el proceso hasta que el mismo es finalizado. El dato se calcula de las estadísticas tomadas de los registros del proceso JobHistory Server y se procesaron con el algoritmo RUMEN, que genera un archivo JSON con los registros de cada experimento.

Cada registro de experimento genera estadísticas a nivel de Job, Task y TaskAttempts. El tiempo de construcción es la diferencia entre finishTime y startTime de cada Job.

Se analizan a continuación los tiempos de construcción de índices para los diferentes algoritmos. Se separan los tiempos por algoritmo (Tipo de algoritmo y formato de salida) y se estudia la evolución de los tiempos para las diferentes combinaciones de configuración de cluster y procesos Reducers desplegados. Se discrimina además para cada una de las colecciones.

| Experimento | C1 | C2 | C3 |
|---|---|---|---|
| **0N-1R** | 26011,193 | 52950,164 | 80507,467 |
| **1N-1R** | 26838,776 | 55844,728 | S/D |
| **2N-1R** | 11207,308 | 22709,396 | 34612,986 |
| **2N-2R** | 11087,425 | 22085,329 | 39818,040 |
| **4N-1R** | 6572,095 | 13159,986 | 21100,789 |
| **4N-2R** | 5434,703 | 10694,047 | 16879,659 |
| **4N-4R** | 5333,012 | 10396,146 | 15337,024 |
| **6N-1R** | 5546,969 | 14595,622 | 23249,550 |
| **6N-2R** | 4376,344 | 7986,715 | 17520,411 |
| **6N-4R** | 3420,484 | 6926,291 | 9995,482 |
| **6N-6R** | 3209,768 | 7068,320 | 12899,832 |

Tabla 4.2: Tiempos de construcción de índices (en segundos). Algoritmo Baseline Comprimido (BL-CC).

En la Tabla 4.2 y en la Figura 4.1b se observan que en la medida que se incrementan la cantidad de nodos se observa una tendencia al descenso en los tiempos. Este comportamiento general se repite para todos los algoritmos.

A partir de la configuración de 2 Nodos los tiempos se reducen. Por ejemplo, el algoritmo BM-CC para la colección C3, registra una mejora del 59,42 % respecto a la ejecución secuencial del mismo algoritmo y colección.

Cuando se agregan 4 nodos a la configuración, se percibe una mejora mayor. En el caso de BM-CC-C3 la configuración que mejor rendimiento tiene es 4N-4R, que respecto a la corrida secuencial alcanza una reducción del tiempo de procesamiento de un 81,7 %. Este valor se mantiene similar para la misma configuración para el mismo algoritmo respecto a C1 y C2 (80,4 % y 80,7 % respectivamente). El resto de los algoritmos registra tiempos similares. BL-SC alcanza mejoras de entre 80 % y 81 % para 4N-4R para cada una de las colecciones. Para los algoritmos BL-CC y BM-SC se observan proporciones similares, dándose el peor caso en BL-CC-C1 con una disminución del 79,5 %.

| Experimento | C1 | C2 | C3 |
|---|---|---|---|
| **0N-1R** | 26137,368 | 53324,642 | 80209,510 |
| **1N-1R** | 26976,092 | 56062,717 | S/D |
| **2N-1R** | 11454,042 | 22764,499 | 34797,463 |
| **2N-2R** | 11285,692 | 22598,283 | 39899,306 |
| **4N-1R** | 6653,891 | 13283,612 | 21063,882 |
| **4N-2R** | 5448,491 | 10754,403 | 23219,713 |
| **4N-4R** | 5146,407 | 10264,901 | 15343,850 |
| **6N-1R** | 5837,719 | 11668,007 | 19210,043 |
| **6N-2R** | 4041,535 | 7864,134 | 12612,234 |
| **6N-4R** | 3418,497 | 6735,543 | 10078,965 |
| **6N-6R** | 3189,933 | 7080,187 | 12717,231 |

Tabla 4.3: Tiempos de construcción de índices (en segundos). Algoritmo Baseline sin Comprimir (BL-SC).

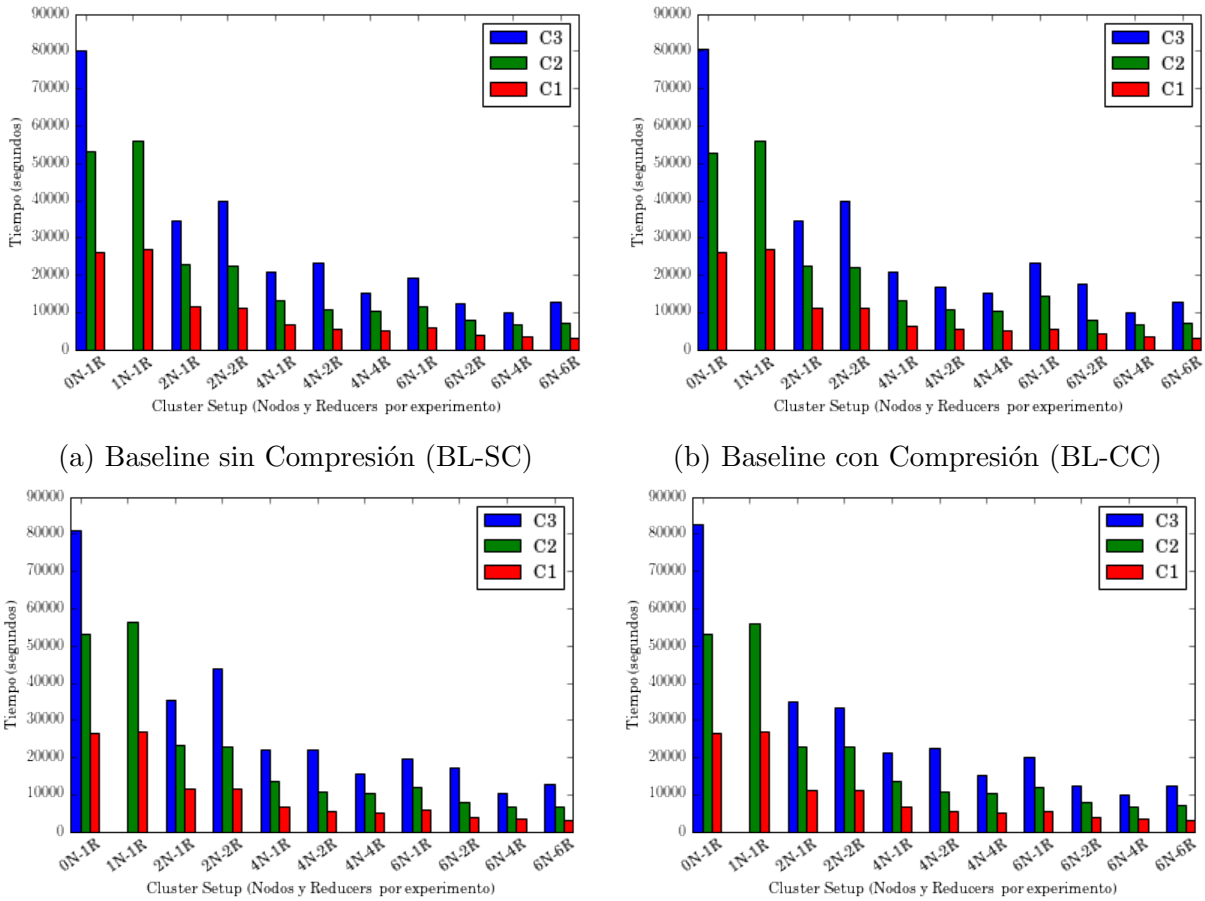
Para la configuración de 6 nodos se pueden observar los mejores tiempos de procesamiento. Dejando de lado la configuración 6N-1R (que se aborda mas adelante en esta sección), se registran mejoras de entre el 78,24 % (En el caso de BL-CC-6N-2R-C3) hasta el 88,07 % (Como se observa para el experimento BM-CC-6N-6R-C1). Estos porcentajes siempre comparando con

| Experimento | C1 | C2 | C3 |
|---|---|---|---|
| **0N-1R** | 26366,853 | 53290,361 | 81060,354 |
| **1N-1R** | 27053,689 | 56509,802 | S/D |
| **2N-1R** | 11597,016 | 23324,419 | 35429,034 |
| **2N-2R** | 11518,062 | 22802,684 | 43994,824 |
| **4N-1R** | 6762,284 | 13766,934 | 21994,637 |
| **4N-2R** | 5554,556 | 10805,609 | 21969,412 |
| **4N-4R** | 5147,026 | 10248,214 | 15745,185 |
| **6N-1R** | 6134,251 | 12195,245 | 19834,939 |
| **6N-2R** | 4159,241 | 7987,941 | 17157,718 |
| **6N-4R** | 3523,757 | 6856,038 | 10460,787 |
| **6N-6R** | 3236,844 | 6919,024 | 12981,668 |

Tabla 4.4: Tiempos de construcción de índices (en segundos). Algoritmo Block-Max sin Comprimir (BM-SC).

| Experimento | C1 | C2 | C3 |
|---|---|---|---|
| **0N-1R** | 26486,421 | 53050,557 | 82702,670 |
| **1N-1R** | 27073,788 | 56189,979 | S/D |
| **2N-1R** | 11230,281 | 22847,085 | 34881,028 |
| **2N-2R** | 11104,052 | 22751,222 | 33558,136 |
| **4N-1R** | 6628,732 | 13465,287 | 21214,401 |
| **4N-2R** | 5488,190 | 10899,734 | 22327,590 |
| **4N-4R** | 5188,618 | 10232,566 | 15129,476 |
| **6N-1R** | 5645,735 | 11860,155 | 20008,918 |
| **6N-2R** | 4086,059 | 7876,762 | 12293,151 |
| **6N-4R** | 3409,284 | 6913,552 | 10043,419 |
| **6N-6R** | 3158,890 | 7127,812 | 12359,091 |

Tabla 4.5: Tiempos de construcción de índices (en segundos). Algoritmo Block-Max con Compresión (BM-CC).




(a) Baseline sin Compresión (BL-SC)

(b) Baseline con Compresión (BL-CC)

(c) Block-Max sin Compresión (BM-SC)

(d) Block-Max con Compresión (BM-CC)

Figura 4.1: Tiempos de construcción por experimento para las colecciones C1, C2 y C3

el respectivo algoritmo secuencial al igual que los párrafos anteriores.

Se puede ver además que para una cantidad de nodos fija, el incremento de los procesos Reducers introduce mejoras en todos los tiempos solo para el caso de C1. Llegando a las ultimas dos configuraciones (6N-4R y 6N-6R), estas mejoras rondan el 1 % a favor de 6N-6R respecto de 6N-4R. Sin embargo, para C2 y C3 se observa que en todos los algoritmos la configuración 6N-4R de nodos/Reducers mejora en tiempos a 6N-6R. Para C2, los algoritmos corridos sobre 6N-4R mejoran entre un 0,12 % los tiempos de aquellos que se ejecutaron en 6N-6R (Para el caso de BM-SC) llegando hasta un porcentaje de mejora del 0,65 % de la primer configuración respecto de la segunda ejecutando el algoritmo BL-SC. Para la ejecución de los algoritmos sobre las mismas 2 configuración C3 se tienen resultados similares a C2, pero

en este caso los porcentajes de correr los indexadores sobre 6N-4R mejoran mucho mejor el rendimiento respecto a hacerlo sobre 6N-6R (Los porcentajes de mejora varían entre el 2,8 % para el algoritmo BM-CC hasta 3,61 % para BL-CC). Todos estos datos fueron ordenados en la tabla 4.6 y se pueden ver gráficamente en la Figura 4.2.

En resumen de lo anterior, para las configuraciones 6N-4R y 6N-6R se parte de una situación donde la primera rinde menos que la segunda para C1, pasando a rendir ligeramente mejor la primera en detrimento de la segunda para C2 y acentuándose aun mas la tendencia de C2 para el caso de C3. Esta situación donde agregar Reducers no siempre implica mejora es consistente con las recomendaciones de la plataforma[11], donde se explica un modelo para aproximar la cantidad de reducers en base a multiplicar un factor (0,95) por la cantidad de nodos y las tareas a ejecutar por nodo. El modelo indica que para 6 nodos el valor de Reducers debe ser 5, por lo cual en el caso de 6N-6R exceden el valor en cuestión. Una hipótesis es que la sobrecarga de datos de control de la plataforma esta generando el incremento de tiempos para la cantidad de Reducers en cuestión sobre los nodos. Mientras la cantidad de nodos crece, la plataforma necesita generar mas datos de control que perjudica el tiempo. Esto a su vez empeora en la medida que la colección es mayor, mostrando las diferencias de C1 (donde al ser una colección pequeña la sobrecarga no alcanza a perjudicar la ejecución) con C2 y C3.

Para el resto de las configuraciones se cumple que agregar Reducers para una misma cantidad de nodos mejora el tiempo de ejecución, salvo excepciones puntuales.

Para C1 y C2, se puede observar que la ejecución del algoritmo secuencial (0N) es siempre mejor que la ejecución distribuida con un único nodo (1N). Para el caso de C1, el algoritmo BL con 1N decae en proporción 3,18 % y 3,21 % (Con y sin compresión respectivamente) en relación a la ejecución local. Para esta misma colección, el algoritmo BM se ve afectado en menor medida, ya que BM-CC registro un incremento de tiempo del 2,22 % y la versión sin comprimir lo hizo en el orden del 2,6 %.

Para el caso de la colección 2 esta diferencia en favor de la ejecución secuencial es aun mas marcada, encontrando diferencias de entre 5,13 % para BL-SC y 5,47 % para BL-CC. A diferencia de lo que sucedió con C1, en este caso el algoritmo BM incremento aun mas su tiempo en relación a BL. El experimento BM-SC-C2 para 1 nodo tardo un 6,04 % mas que su versión 0N, y BM-CC con la misma colección lo hizo en un 5,92 %.

[11] Ver notas bajo el subtitulo "How Many Reduces?" en el sitio `https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`

Una hipótesis que puede explicar parte de esto es que el cuello de botella en la transferencia de red en la configuración 1N puede estar causando la degradación en el rendimiento general. Esta hipótesis se refuerza cuando los algoritmos corriendo sobre C2 registran una diferencia tan consistente en relación a idénticos algoritmos ejecutándose sobre C1.

Respecto a las diferencias entre algoritmos, no existe evidencia que surja de las pruebas realizadas que permita explicar el motivo por el cual el algoritmo BL tiene un mejor rendimiento que BM para C1 pero en la segunda colección se da el comportamiento inverso. Se considera que la cantidad de pruebas realizadas es baja para asegurar que se esta frente a una tendencia (8 Pruebas en total, 2 para BL y 2 para BM para ambas colecciones).

| Reducers | C1 | C2 | C3 |
|---|---|---|---|
| **1R** | 5546,969 | 11668,007 | 19210,043 |
| **2R** | 4041,535 | 7864,134 | 12293,151 |
| **4R** | 3409,284 | 6735,543 | 9995,482 |
| **6R** | 3158,89 | 6919,024 | 12359,091 |

Tabla 4.6: Tiempos de construcción (en segundos) por cantidad de Reducers para una configuración de cluster de 6 Nodos (6N).

Otro comportamiento que se presenta de forma consistente es que se encontró un mejor rendimiento en la configuración 4N-4R que con 6N-1R. Esto se puede ver en la Tabla 4.7. Se puede notar que no importa algoritmo ni colección, en todos los casos la configuración de 6 Nodos y 1 Reduce es peor. De dicha Tabla se puede observar que en la medida que la colección se hace mayor, esta proporción se hace, en promedio, cada vez mas grande. Por ejemplo, la proporción mínima para C1 es de 4 % (Para BL-CC). Para C2 se corresponde con 13 % (En el caso de BL-SC). Mientras que para C3 se puede observar, en el experimento BL-SC una proporción del 25 %. Si se toma en cuenta el valor máximo, se observa una tendencia similar, con 19 % para C1, la segunda colección con 40 % y la tercera colección tiene un máximo en 51 %.

En la Figura 4.3 se puede observar el efecto del incremento de los nodos en el cluster tomando para cada uno de ellos el mejor tiempo obtenido para alguna configuración de Reducers. Las sub-figuras fueron construidas en base a los datos de las tablas 4.3, 4.2, 4.4 y 4.5.

Se puede observar que de manera consistente, las 4 figuras en 4.3 muestran una tendencia descendiente conforme el número de nodos aumenta. Esta

| | | 4N-4R | 6N-1R | Incremento |
|---|---|---|---|---|
| **BL-SC** | **C1** | 5146,407 | 5837,719 | 13,43 % |
| | **C2** | 10264,901 | 11668,007 | 13,67 % |
| | **C3** | 15343,85 | 19210,043 | 25,20 % |
| **BL-CC** | **C1** | 5333,012 | 5546,969 | 4,01 % |
| | **C2** | 10396,146 | 14595,622 | 40,39 % |
| | **C3** | 15337,024 | 23249,550 | 51,59 % |
| **BM-SC** | **C1** | 5147,026 | 6134,251 | 19,18 % |
| | **C2** | 10248,214 | 12195,245 | 19,00 % |
| | **C3** | 15745,185 | 19834,939 | 25,97 % |
| **BM-CC** | **C1** | 5188,618 | 5645,735 | 8,81 % |
| | **C2** | 10232,566 | 11860,155 | 15,91 % |
| | **C3** | 15129,476 | 20008,918 | 32,25 % |

Tabla 4.7: Incremento de tiempo de procesamiento (en segundos) de experimento 6N-1R respecto de 4N-4R

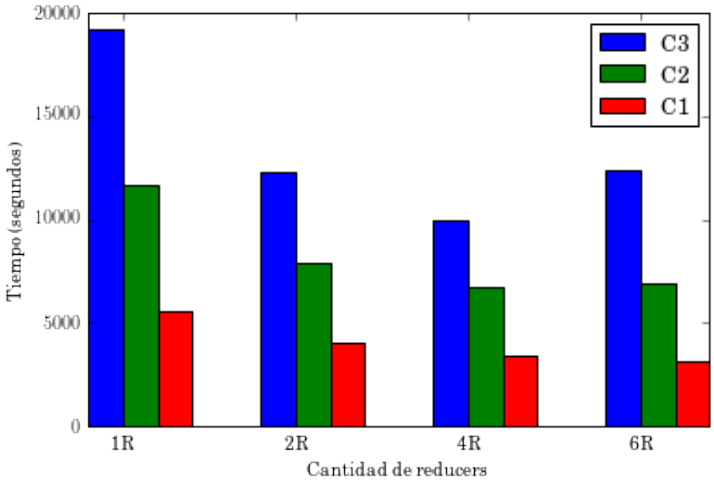



Figura 4.2: Tiempos de construcción de índices en un cluster de 6 nodos tomando el mejor tiempo para cada experimento.

observación es análoga a la comentada para la Figura 4.1, solo que ahora se aísla de efectos de las configuraciones que rindieron menos.

También se puede observar que las figuras incluyen las lineas de tendencia ajustadas con una curva logarítmica. Por claridad no se incluye en las mismas las ecuaciones de las curvas de tendencia ni el coeficiente de correlación $R^2$, los cuales se pueden observar en la Tabla 4.8.

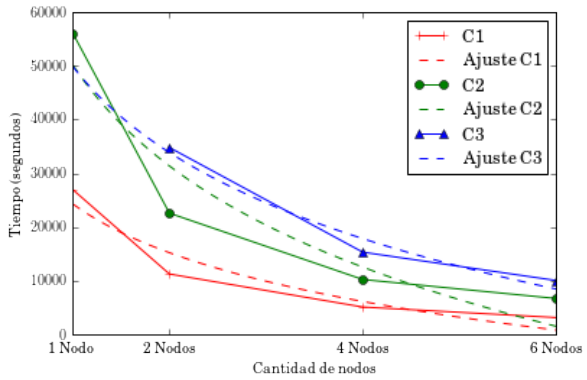
La primer observación que se puede realizar es que el coeficiente de correlación para casi todos los casos es mayor a 0,9 (Se registra un valor de 0,899 en el caso de BL-CC-C2). En particular para el caso de C3, se puede ver que en todas las curvas de ajustes es mayor que para las otras 2 colecciones. Esto se explica principalmente por la falta de datos para C3 (Faltan todas las pruebas en relación a 1N-C3).

Por otro lado, para C1 y C2, el coeficiente de la curva de ajuste de BL-CC es mayor al del resto de los experimentos. También se puede observar que para todos los algoritmos y la misma colección, la versión sin compresión tiene un coeficiente menor que la versión con compresión. Un ejemplo de esto es los coeficientes entre BM-SC (-23337,7118149268) y BM-CC (-21962,1250413717) para la Colección 3. Se puede observar estas diferencias en las ecuaciones de la Tabla 4.8.

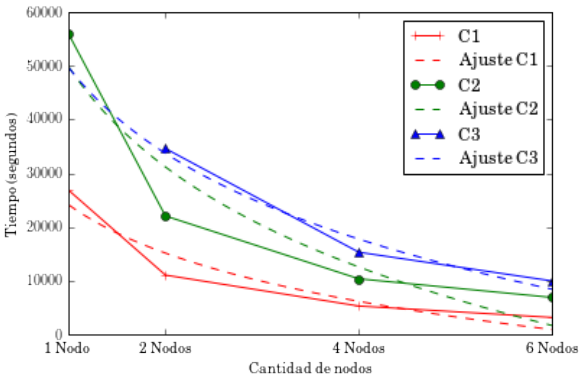
Se representaron curvas de ajuste logarítmicas. Esto se debe a que con la incorporación de los primeros nodos el tiempo disminuye de forma acelerada, pero en determinado momento la mejora obtenida al agregar mas nodos es

| | **Ajuste (Formula)** | $R^2$ |
|---|---|---|
| | **C1** | |
| **BL-SC** | $-13088,6718968037 * \ln x + 24316,7509696745$ | 0,916 |
| **BL-CC** | $-12937,6710157737 * \ln x + 24138,3265286992$ | 0,913 |
| **BM-SC** | $-13142,2632267535 * \ln x + 24457,9909223115$ | 0,920 |
| **BM-CC** | $-13116,8455426022 * \ln x + 24325,8234311186$ | 0,912 |
| | **C2** | |
| **BL-SC** | $-27125,0762447019 * \ln x + 50167,0166448059$ | 0,905 |
| **BL-CC** | $-26822,8723596723 * \ln x + 49772,3061485867$ | 0,899 |
| **BM-SC** | $-27337,5648638873 * \ln x + 50561,4866841876$ | 0,905 |
| **BM-CC** | $-27142,2397985841 * \ln x + 50290,0962866083$ | 0,905 |
| | **C3** | |
| **BL-SC** | $-23099,0922784781 * \ln x + 49880,5025497996$ | 0,971 |
| **BL-CC** | $-22989,4641488247 * \ln x + 49647,4429509869$ | 0,973 |
| **BM-SC** | $-23337,7118149268 * \ln x + 50659,9931899738$ | 0,971 |
| **BM-CC** | $-21962,1250413717 * \ln x + 47916,9438872479$ | 0,973 |

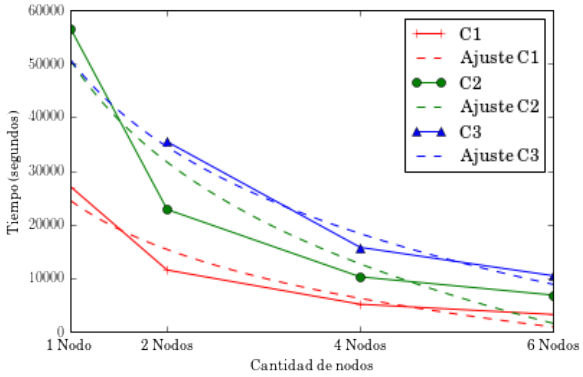
Tabla 4.8: Función de las curvas de ajuste y coeficientes de correlación para Figura 4.3



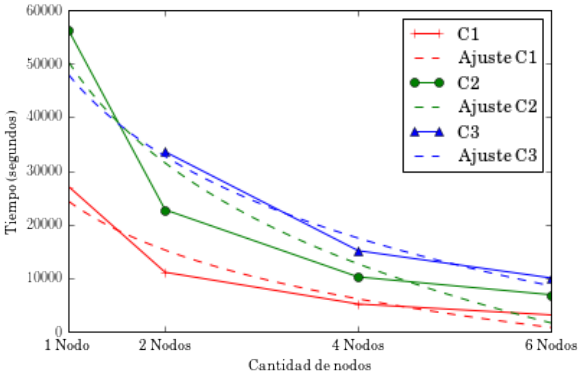

(a) Baseline sin Compresión (BL-SC)




(b) Baseline con Compresión (BL-CC)




(c) Block-Max sin Compresión (BM-SC)




(d) Block-Max con Compresión (BM-CC)

Figura 4.3: Tiempos de construcción por experimento y nodos para las diferentes colecciones, tomando como valor el mejor resultado entre las diferentes configuraciones de Reducers.

cada vez menor. Un ajuste logarítmico se considera lo mas adecuado para datos que se comportan de este modo.

### 4.5.2. Speedup y Eficiencia

A partir de los tiempos analizados, se puede construir una curva de Speedup para determinar cuanto se mejora el rendimiento de los algoritmos con la incorporación de nodos.

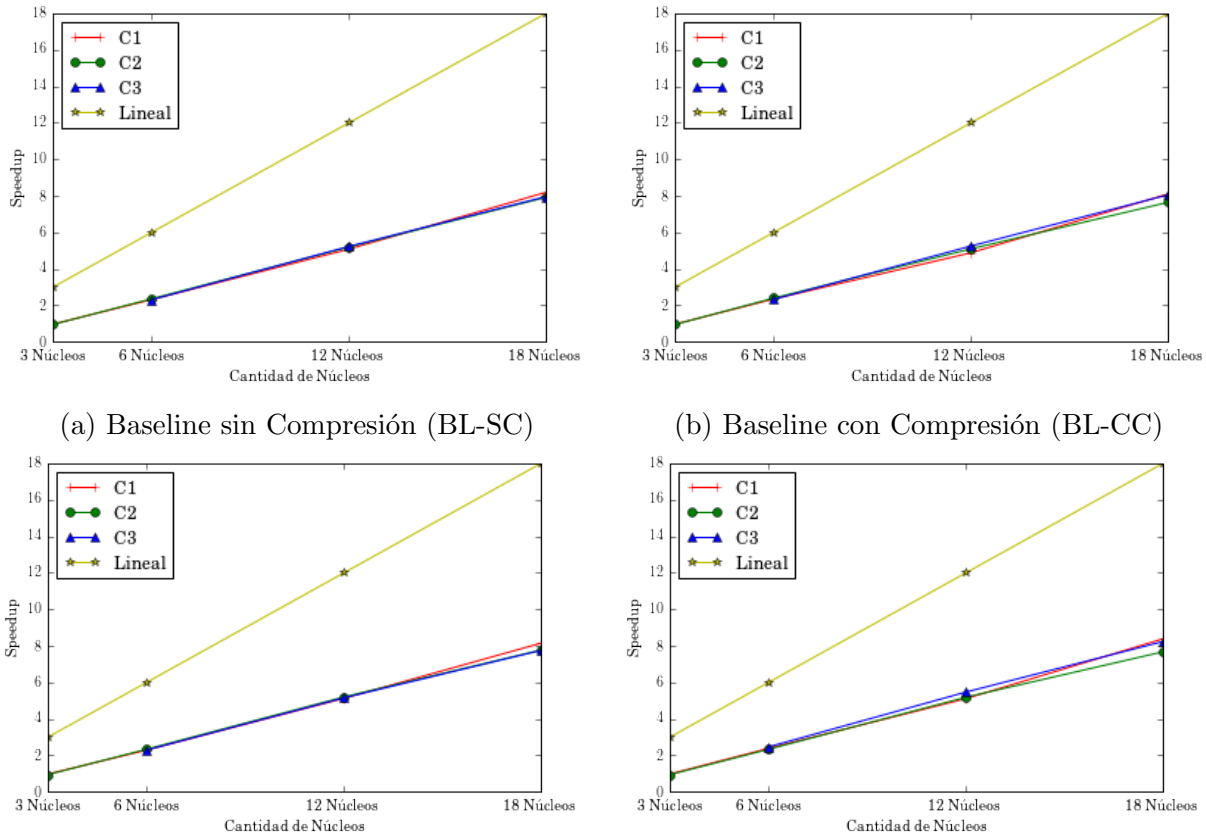
Se decide que se calcula el Speedup en base a los núcleos disponibles para Hadoop en lugar de los nodos disponibles, a diferencia de lo realizado por Mccreadie et. al. [35]. Esto tiene varios motivos:

- Calcular Speedup en base a los nodos sobrestimaba el crecimiento y

daba a los resultados un comportamiento superlineal, es decir, que superaban a la curva teórica de rendimiento. Este comportamiento no describe adecuadamente los resultados.

- Calcular las curvas en base a los núcleos permite un análisis mas detallado del comportamiento de los algoritmos. Por eso se eligen 3 núcleos y no 4, dado que por configuración se deja un núcleo y parte de la memoria principal para el sistema operativo.

En la Figura 4.4 se pueden apreciar las curvas para los 2 algoritmos en sus versiones con y sin compresión.




(a) Baseline sin Compresión (BL-SC)

(b) Baseline con Compresión (BL-CC)

(c) Block-Max sin Compresión (BM-SC)

(d) Block-Max con Compresión (BM-CC)

Figura 4.4: Speedup para los algoritmos Baseline y Block-Max sobre las diferentes colecciones.

Como resulta esperable debido al carácter de prototipo de los algoritmos construidos, las curvas de Speedup se alejan del lineal teórico, con leves diferencias entre las distintas colecciones recién visibles en los 18 núcleos en

| Speedup | Lineal | C1 | C2 | C3 |
|---|---|---|---|---|
| **3** | 3 | 0,969 | 0,951 | S/D |
| **6** | 6 | 2,316 | 2,360 | 2,305 |
| **12** | 12 | 5,079 | 5,195 | 5,227 |
| **18** | 18 | 8,194 | 7,917 | 7,958 |

Tabla 4.9: Speedup para Baseline sin Compresión (BL-SC)

| Speedup | Lineal | C1 | C2 | C3 |
|---|---|---|---|---|
| **3** | 3 | 0,969 | 0,948 | S/D |
| **6** | 6 | 2,346 | 2,398 | 2,326 |
| **12** | 12 | 4,877 | 5,093 | 5,249 |
| **18** | 18 | 8,104 | 7,645 | 8,054 |

Tabla 4.10: Speedup para Baseline con Compresión (BL-CC)

| Speedup | Lineal | C1 | C2 | C3 |
|---|---|---|---|---|
| **3** | 3 | 0,975 | 0,943 | S/D |
| **6** | 6 | 2,289 | 2,337 | 2,288 |
| **12** | 12 | 5,123 | 5,200 | 5,148 |
| **18** | 18 | 8,146 | 7,773 | 7,749 |

Tabla 4.11: Speedup para Block-Max sin Compresión (BM-SC)

| Speedup | Lineal | C1 | C2 | C3 |
|---|---|---|---|---|
| **3** | 3 | 0,978 | 0,944 | S/D |
| **6** | 6 | 2,385 | 2,332 | 2,464 |
| **12** | 12 | 5,105 | 5,184 | 5,466 |
| **18** | 18 | 8,385 | 7,673 | 8,235 |

Tabla 4.12: Speedup para Block-Max con Compresión (BM-CC)

algunos casos (Ver figuras 4.4d y 4.4b por ejemplo). Este comportamiento del Speedup es consistente con los resultados de otras investigaciones de Recuperación de Información sobre Hadoop [36].

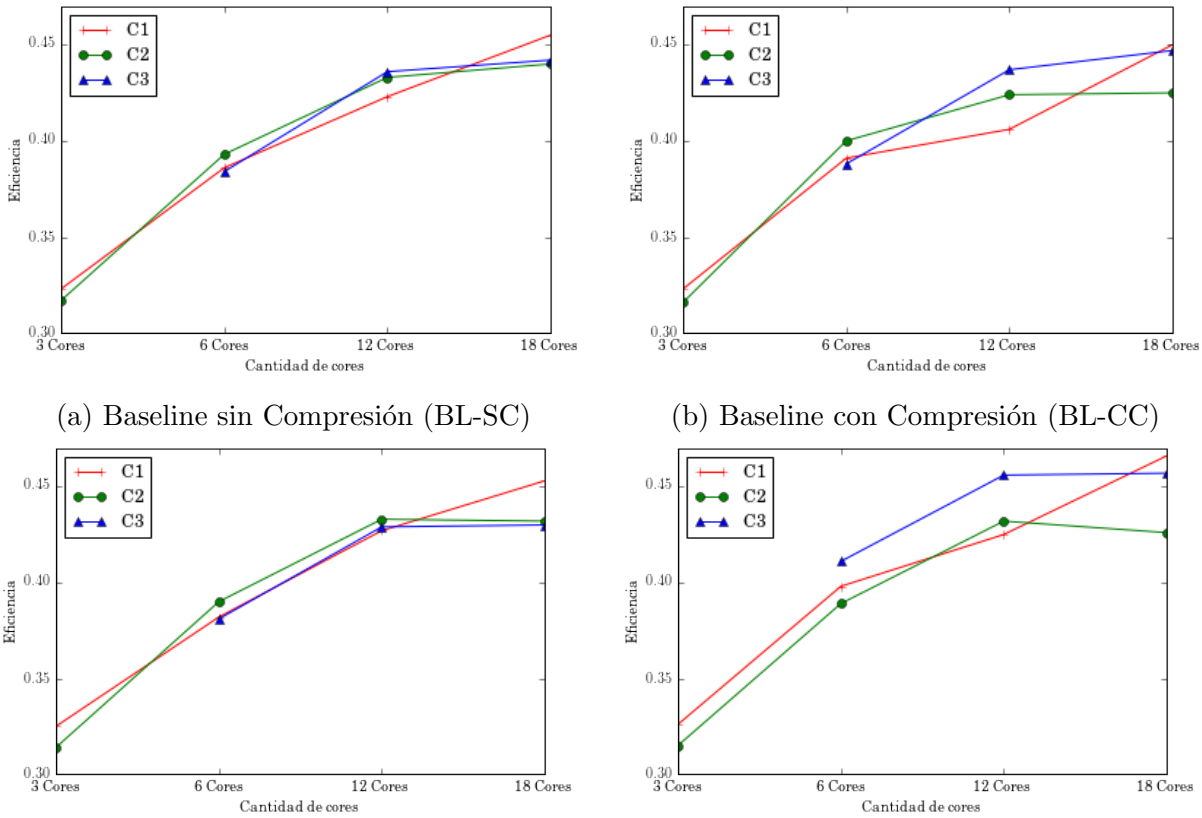
Por un lado, esto indica que el tamaño de las colecciones no alcanza a afectar el desempeño de los algoritmos. Por otro lado, las diferencias entre las curvas de Speedup respecto de la curva lineal indica que siempre es posible realizar mejoras al algoritmo, aunque debe tenerse en cuenta la sobrecarga de procesamiento propia de la plataforma (En este caso Hadoop).

La información disponible de las pruebas no permite afirmar nada respecto a la mejora que el proceso de compresión introduce dado que los tiempos y el Speedup de los algoritmos con y sin comprimir no presenta diferencias prácticamente. Se recuerda que los algoritmos se ejecutan en un contexto de hardware de bajo costo o "commodity" y son equipos con prestaciones orientados a usuario final. Además, resulta claro que los dos algoritmos escalan de forma similar, dado que si se analizan los valores de las diferentes tablas (Ver tablas 4.9, 4.10, 4.11 y 4.12) se encuentra valores de Speedup prácticamente iguales.

La eficiencia permite medir de forma sencilla como se están utilizando los recursos. La eficiencia puede ayudar a comprender que aunque se inviertan recursos en cantidad y se logre mejorar el Speedup, los mismos pueden ser excesivos y se esten desperdiciando en relación a los resultados obtenidos.

En la tabla 4.13 se puede observar que la eficiencia para todos los casos varia de entre 31 % y 45/46 %. Para cada algoritmo / colección, resulta claro que el incremento de cores mejora la eficiencia, salvo para 18 cores en casos determinados. Por ejemplo, para BM-C2 en los casos que se pasa de 12 a 18 cores (Es decir, en el paso de 4 a 6 nodos) se observa que la eficiencia pasa de 43,3 % a 43,2 % para el caso SC, mientras que para CC desciende desde 43,2 % hasta 42,6 %.

En la Figura 4.5 se puede observar claramente que para C2 y C3, el




(a) Baseline sin Compresión (BL-SC)

(b) Baseline con Compresión (BL-CC)

(c) Block-Max sin Compresión (BM-SC)

(d) Block-Max con Compresión (BM-CC)

Figura 4.5: Eficiencia para C1, C2 y C3 con los diferentes algoritmos

incremento de la eficiencia es menos pronunciado en la medida que se agregan núcleos (incluso para C2 hay 2 casos donde disminuye, explicados en el párrafo anterior). Para C1, se puede observar que la curva de eficiencia sigue creciendo aun en los 18 cores y lo hace con mejor pendiente que en el caso de las otras 2 colecciones.

Para finalizar el análisis de eficiencia, se considera de interés mostrar en una misma Figura las gráficas de Eficiencia para todos los algoritmos con una misma colección. En este caso todas las curvas para idéntica colección mostraron comportamiento similares variando en alguno puntos porcentuales. Se puede observar para el caso de C3 en la Figura 4.6 (que refleja los datos de 4.14). Mientras que la eficiencia para Baseline se mantiene casi idéntica para ambas versiones del mismo, en el caso de Block-Max se plantea una diferencia (muy pequeña) de aproximadamente 0,03 % que se mantiene para las diferentes configuraciones de nodos.

Además se puede observar que la eficiencia promedio crece en la medida

| Eficiencia | | | | | | | |
|---|---|---|---|---|---|---|---|
| **BL-SC** | **C1** | **C2** | **C3** | | **BL-CC** | **C1** | **C2** | **C3** |
| **3** | 0,323 | 0,317 | S/D | | **3** | 0,323 | 0,316 | S/D |
| **6** | 0,386 | 0,393 | 0,384 | | **6** | 0,391 | 0,400 | 0,388 |
| **12** | 0,423 | 0,433 | 0,436 | | **12** | 0,406 | 0,424 | 0,437 |
| **18** | 0,455 | 0,440 | 0,442 | | **18** | 0,450 | 0,425 | 0,447 |
| **BM-SC** | **C1** | **C2** | **C3** | | **BM-CC** | **C1** | **C2** | **C3** |
| **3** | 0,325 | 0,314 | S/D | | **3** | 0,326 | 0,315 | S/D |
| **6** | 0,382 | 0,390 | 0,381 | | **6** | 0,398 | 0,389 | 0,411 |
| **12** | 0,427 | 0,433 | 0,429 | | **12** | 0,425 | 0,432 | 0,456 |
| **18** | 0,453 | 0,432 | 0,430 | | **18** | 0,466 | 0,426 | 0,457 |

Tabla 4.13: Eficiencia para los diferentes algoritmos por colección

| Eficiencia para Colección 3 (C3) | | | | |
|---|---|---|---|---|
| **Cores** | **BL-SC** | **BL-CC** | **BM-SC** | **BM-CC** |
| **6** | 0,384 | 0,388 | 0,381 | 0,411 |
| **12** | 0,436 | 0,437 | 0,429 | 0,456 |
| **18** | 0,442 | 0,447 | 0,430 | 0,457 |

Tabla 4.14: Eficiencia de los algoritmos durante la construcción del índice para la Colección 3 (C3)

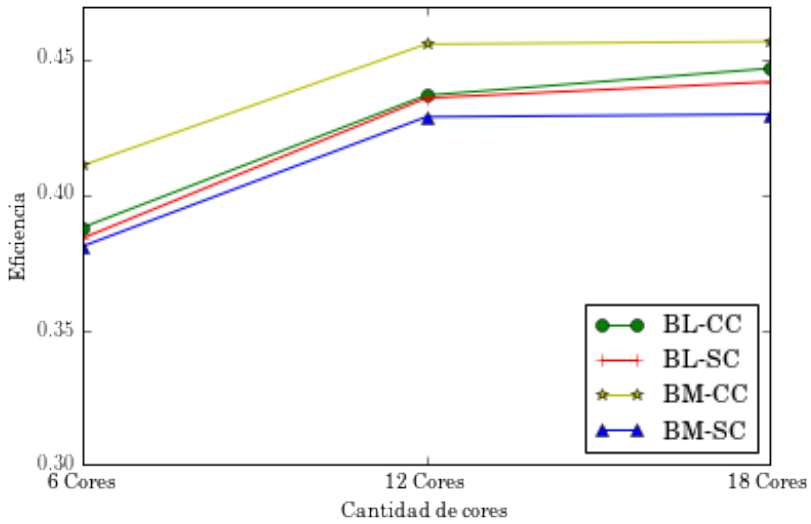



Figura 4.6: Eficiencia de los algoritmos durante la construcción del índice para la Colección 3 (C3)

que también lo hace la cantidad de núcleos. Para 6 núcleos la eficiencia promedio es de 39,1 %, mientras que para 12 esta proporción media asciende a 43,9 % y para 18 núcleos es de 44,4 %.

### 4.5.3. Imbalance de Carga por Fase

Como se desarrolla anteriormente en este mismo capitulo, el imbalance de carga es una medida que indica cuando un nodo ocupa mayor cantidad de CPU respecto a los demás durante la ejecución de un proceso distribuido.

Se calculó el imbalance de carga para la mayoría de los experimentos, diferenciando entre las fases Map y Reduce. Algunos valores de imbalance no están disponibles. Dichos valores fueron indicados con "S/D". El principal motivo por el cual un experimento no tiene imbalance es porque el servicio Ganglia no registro valores para el uso de CPU durante el tiempo de ese experimento, o lo hizo con valores inconsistentes.

#### Fase Map

En la Tabla 4.15 se observa el imbalance para los diferentes experimentos en la Fase Map para una configuración de 2 Nodos. Se divide por tipo de algoritmo y colección.

| | BL-SC | BL-CC | BM-SC | BM-CC |
|---|---|---|---|---|
| **C1** | 11,915 | 36,026 | 23,300 | 24,069 |
| | 28,395 | S/D | 21,238 | 21,650 |
| **C2** | 13,699 | 34,045 | 17,673 | 15,340 |
| | 40,124 | S/D | S/D | 12,510 |
| **C3** | 40,290 | S/D | 40,191 | 39,140 |
| | 41,327 | 43,302 | 39,980 | 41,507 |

Tabla 4.15: Imbalance por colección y experimento para configuraciones de 2 Nodos durante la fase Map

Tomando los valores por colección, se puede observar que para C1 se tiene un promedio de imbalance de entre 23 % y 24 %. Esto significa que 1 de los 2 nodos tuvo un uso de CPU un 24 % por encima del promedio de uso.

Se observa que para el caso del algoritmo BL se obtienen los máximos y mínimos, 11,9 % y 36 % respectivamente. Esto no tiene relación con el tipo de algoritmo, dado que la fase Map es idéntica para ambos. Una hipótesis a estudiar es si este imbalance es propio de la plataforma, siendo el planificador de tareas de Hadoop el que introduce el mismo.

En el caso de C2, se observa un imbalance promedio del 22 %. Para todas las colecciones, se observa que en la Tabla se tienen 4.15 2 filas por colección. Se corresponde con las configuraciones de Reducers (La primer fila refiere a 1R y la segunda a 2R). Separando por fila, el promedio de para 1R (primer fila de C2) es de 20 % y el de 2R es 26 %.

Para la colección 3 se observa que el imbalance se incrementa notablemente. El promedio es de aproximadamente 41 %. El imbalance máximo se observa en BL-CC-2R-C3 y es de 43 %. El mínimo es de 39 % y se observa para BM-CC-1R-C3.

Los experimentos realizados no son suficientes para afirmar si el incremento de los datos de entrada tienen correlación con el aumento del imbalance. Para el caso de C2, el promedio general es menor que para C1 (22 % contra 23-24 % respectivamente). Para C3, este promedio asciende a casi el doble (41 %).

Diferenciando por tipo de algoritmo, el imbalance promedio es aproximadamente un 29 % para BL-SC, para el caso de BL-CC el promedio es de 37,791 %. Para los algoritmos Block-Max, se tiene un promedio de imbalance de 28 % para SC y un 25 % para CC. De nuevo, estas diferencias deben

estudiarse sin tomar en cuenta las diferencias entre algoritmos (BL y BM) y entre formatos (SC y CC) porque las mismas se ejecutan en la fase Reduce.

| | BL-SC | BL-CC | BM-CC | BM-SC |
|---|---|---|---|---|
| **C1** | 13,953 | 20,082 | 14,939 | 15,418 |
| | 18,093 | S/D | 38,525 | 24,914 |
| | 13,159 | 50,170 | 15,622 | S/D |
| **C2** | 16,585 | 17,762 | 12,346 | S/D |
| | 27,582 | S/D | 29,881 | 16,648 |
| | 21,446 | 46,554 | 13,932 | 22,990 |
| **C3** | 16,781 | 13,480 | 13,572 | 16,529 |
| | 24,347 | 17,340 | 26,285 | 14,532 |
| | 25,085 | 18,931 | 13,838 | 49,002 |

Tabla 4.16: Imbalance por colección y experimento para configuraciones de 4 Nodos durante la fase Map

Un análisis similar al anterior se puede realizar para la Tabla 4.16, que muestra los valores de imbalance para los experimentos bajo una configuración de 4 nodos. En este caso, se observa que cada colección tiene imbalance en 3 renglones, que representan experimentos para 1R, 2R y 4R.

La colección 1 tiene un promedio de imbalance aproximado de 22 %, con un imbalance máximo de 50 % (para BL-CC-4R) y uno mínimo de 13 % (en el caso de BL-SC-4R). Para C2 el promedio es de 22,5 %. El experimento BL-CC-4R registra el imbalance máximo de la serie (con un porcentaje 46,5 %) y el mínimo se registra para BM-CC-1R siendo el porcentaje de imbalance de 12 %. En el caso de C3, se tiene un promedio general de 21 %. El imbalance máximo es de 49 % para el experimento BM-SC-4R y el mínimo es de 13 % para BL-CC-1R.

En este caso se observa que en promedio, todos los imbalances por colección son similares. Estos resultados contrastan con los encontrados para 2 nodos.

Observando el imbalance por experimento (Es decir, tomando los datos de la Tabla 4.16 por sus columnas), se observan los siguientes promedios: 19,6 % (BL-SC), 26,3 % (BL-CC), 22,8 % (BM-SC) y 19,8 % (BM-CC). En todos los casos, estos promedios son inferiores a sus equivalentes para 2 nodos

(Analizados para la Tabla 4.15).

| | BL-SC | BL-CC | BM-CC | BM-SC |
|---|---|---|---|---|
| **C1** | 27,053 | 37,197 | 21,684 | 24,044 |
| | S/D | S/D | 43,378 | 12,073 |
| | S/D | 20,580 | 18,305 | 18,291 |
| | 41,443 | 48,259 | 19,968 | 24,978 |
| **C2** | 21,560 | 10,142 | 20,904 | 26,530 |
| | 27,863 | S/D | 18,081 | 13,185 |
| | 14,505 | 14,751 | 14,491 | 16,266 |
| | 40,578 | 56,127 | 48,052 | 38,831 |
| **C3** | 20,900 | 15,444 | 28,463 | 15,153 |
| | 13,946 | 10,142 | 12,977 | S/D |
| | 17,968 | 20,996 | 14,444 | 22,519 |
| | 35,548 | 46,253 | 20,905 | 37,962 |

Tabla 4.17: Imbalance por colección y experimento para configuraciones de 6 Nodos durante la fase Map

La ultima configuración de nodos para la fase Map es de 6 nodos. Los resultados de imbalance asociados a estos experimentos se encuentran el la Tabla 4.17. Los 4 renglones representan 1 Reducer, 2 Reducers, 4 Reducers y 6 Reducers.

Para la colección 1, el promedio es de 27,5 % de imbalance. Para la colección 2 este promedio es de 25,5 %. La media de imbalance para C3 es 22,2 %. Los máximos para cada colección son 48,3 % para C1, 56,1 % en el caso de C2 y 46,3 % para C3. Lo destacable en el caso de los máximos que en todos los casos se dieron para BL-CC-6R. Para este mismo experimento, se tienen los valores mínimos para C2 y C3 (10,1 %).

Otro dato a observar es que separando por Reducer, la ultima fila de cada colección presenta el imbalance promedio mayor. Esta ultima fila se corresponde con la configuración 6N-6R. Para C1 este promedio es de 33,7 %, 45,9 % para C2 y en el caso de C3 es de 35,2 %.

Dividiendo por experimento (o sea, tomando las columnas de la Tabla 4.17), el imbalance promedio va entre 22 % (Para BM-CC), y 27,8 % (BL-

CC). En este ultimo experimento, para la configuración de 6N, se da la particularidad de imbalance máximos y mínimos y esto genera que el promedio sea el mayor pero se mantenga dentro de los valores similares al resto de los experimentos.

### Fase Reduce

Para el análisis de imbalance de la fase Reduce, todos los experimentos que tenían 1 Reduce deben ser excluidos, debido a que 1 único proceso reduce es un único nodo procesando, y por ende no se puede medir imbalance en tal escenario.

| | BL-SC | BL-CC | BM-CC | BM-SC |
|---|---|---|---|---|
| **C1** | 21,758 | S/D | 20,788 | 28,523 |
| **C2** | 27,369 | S/D | S/D | S/D |
| **C3** | S/D | 1,051 | 33,241 | 14,390 |

Tabla 4.18: Imbalance por colección y experimento para configuraciones de 2 Nodos durante la fase Reduce

Se pueden observar el imbalance para 2 nodos en la Tabla 4.18. El promedio de imbalance para C1 es de 23,7 %. En el caso de C3 es de 16,2 %. Este ultimo contiene los valores máximos y mínimo de toda la Tabla. Destaca el valor de imbalance 1,1 % que significa que ambos nodos de la configuración durante la fase Reduce utilizaron la misma cantidad de CPU respectivamente. En ningún caso se observan valores superiores a 33,2 %, como sí sucede para la misma configuración de nodos en la fase Map.

Para 4 nodos se observan los valores de imbalance mostrados en la Tabla 4.19. En el caso de la colección 1, se tiene un promedio de imbalance de 17,1 %. Se puede observar que el menor valor de imbalance es 4,4 %, y el mayor valor es 31,9 %. En el caso de C2 se observa un promedio de 16,8 %. En este caso el mínimo es 10,7 % y el máximo observado es 24,7 %. Para C3 se tiene un promedio de 14,5 %, observando un valor mínimo de 4 % y un máximo de 23,2 %.

Al igual que para 2 nodos, esta fase para 4 nodos presenta valores en promedio mucho mas bajos de imbalance que la fase Map correspondiente.

Para la configuración de 6 nodos, cuyos valores de imbalance se presentan en la Tabla 4.20, se omite el análisis detallado de C1 porque se tienen pocos valores. El promedio para dicha colección es de 14,6 %.

| | BL-SC | BL-CC | BM-CC | BM-SC |
|---|---|---|---|---|
| **C1** | 4,374 | S/D | 26,082 | 18,959 |
| | 13,709 | 31,879 | 15,577 | 9,005 |
| **C2** | 19,186 | S/D | S/D | 16,758 |
| | 13,619 | 24,723 | 10,703 | 15,513 |
| **C3** | 4,023 | S/D | 15,159 | 18,546 |
| | 21,975 | 12,143 | 6,627 | 23,200 |

Tabla 4.19: Imbalance por colección y experimento para configuraciones de 4 Nodos durante la fase Reduce

| | BL-SC | BL-CC | BM-CC | BM-SC |
|---|---|---|---|---|
| **C1** | 3,039 | S/D | S/D | 2,848 |
| | S/D | S/D | S/D | 11,410 |
| | 30,996 | S/D | 24,709 | S/D |
| **C2** | 15,068 | S/D | 3,843 | 0,259 |
| | 6,309 | 1,998 | 16,447 | 21,077 |
| | 16,223 | 26,888 | 24,938 | 7,681 |
| **C3** | 0,487 | 30,378 | 27,653 | S/D |
| | 11,472 | 12,092 | 4,300 | 7,367 |
| | 21,951 | 28,462 | 39,750 | 19,990 |

Tabla 4.20: Imbalance por colección y experimento para configuraciones de 6 Nodos durante la fase Reduce

Para la colección 2, el valor medio de imbalance es 12,8 %, teniendo un imbalance de 0,3 % (BM-SC-2R) como valor mínimo y el máximo es 26,9 %, que se observa para BL-CC-6R. La colección 3 presenta un promedio de imbalance de 18,5 %. Se observa para BL-SC-2R un valor mínimo de 0,5 %. El máximo se encuentra para BM-CC-6R y es de 39,8 %.

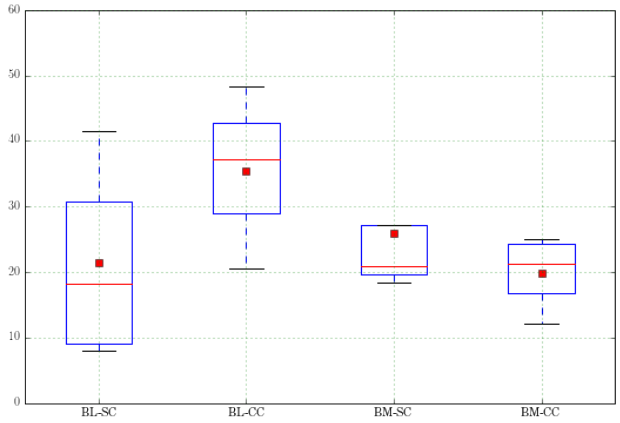
En estos casos también se observa que la fase Reduce presenta valores de imbalance menores si se los compara con los mismos en la fase Map.

Incluso si se analizan por columna esta observación sigue siendo valida. El experimento BL-SC-6N presenta un imbalance promedio de 13,2 % en la fase reduce, contra 26 % para el mismo en la fase Map. BL-CC-6N muestra para la fase Reduce un promedio de 20 % contra 27,8 % del mismo experimento en la fase Map. Para el caso de BM-SC-6N a su vez muestra 10,1 % de imbalance promedio para la fase Reduce, y su valor para la fase Map es de 24,4 %. El experimento BM-CC-6N tiene los valores de imbalance mas parecidos entre fases. Mientras que dicho experimento presenta un valor de 20,2 % para la fase Reduce, su valor durante la fase Map es de 22 %.

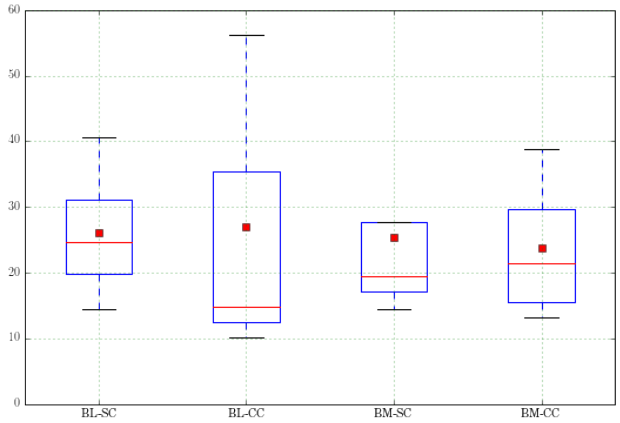
Las Figuras 4.7 y 4.8 muestra diagramas de caja del imbalance de cada uno de los algoritmos para una configuración de 6 Nodos en diferentes fases y para las distintas colecciones.

Una primer observación muestra que la Fase Map tiene una media de imbalance mayor que la Fase Reduce. En general, durante la Fase Map, el algoritmo Baseline tuvo una tendencia a tener una mayor dispersión de imbalance que Block-Max. Esta observación no es intuitiva, ya que el proceso ejecutado durante esta fase para ambos algoritmos es idéntico. Una posible explicación es que este imbalance puede estar siendo introducido por la plataforma, dado que tanto el planificador de tareas de Hadoop como HDFS tienen conocidos problemas de rendimiento en su configuración genérica [45].

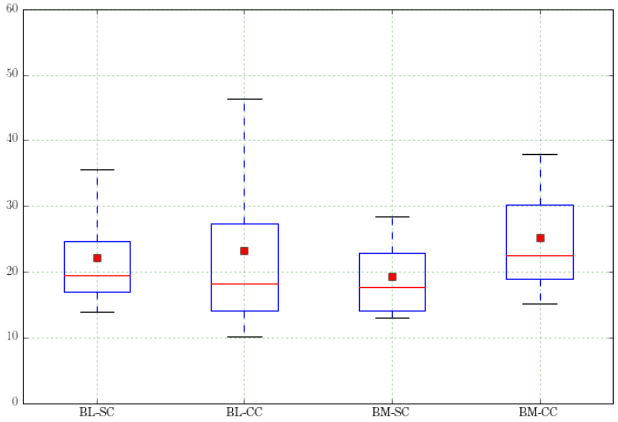
Durante la Fase Reduce, se puede observar que la mediana de imbalance subió para todos los experimentos, menos para BL-SC, donde dicha métrica tuvo un ligero descenso. En esta fase se presento una menor dispersión de los datos en general. Sin embargo, para BM-SC-C3 en Reduce se observa una dispersión mayor, aun si se la compara con el mismo experimento para la Fase Map, sin importar la colección con la cual se lo compare.




(a) Fase Map - C1

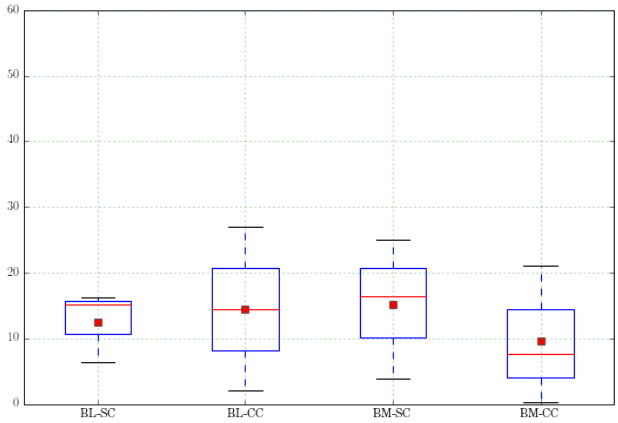



(b) Fase Map - C2

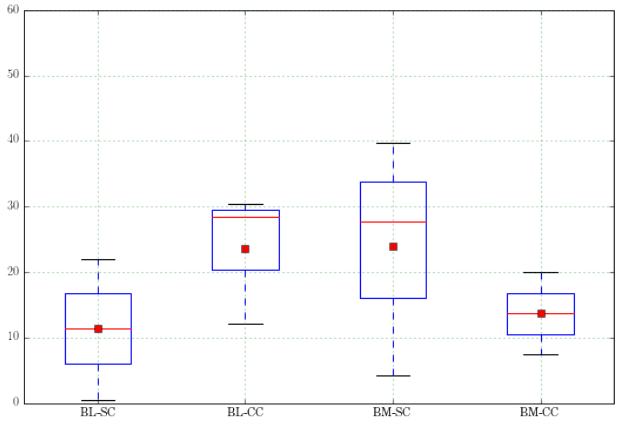



(c) Fase Map - C3

Figura 4.7: Imbalance por experimento y colección para la fase Map con una configuración de 6 Nodos




(a) Fase Reduce - C2




(b) Fase Reduce - C3

Figura 4.8: Imbalance por experimento y colección para fase Reduce en una configuración de 6 Nodos

# Capítulo 5

# Conclusiones del trabajo

El presente trabajo es fruto del esfuerzo de combinar técnicas de Recuperación de Información con tecnologías de Datos Masivos. La motivación principal se centró en construir y estudiar algoritmos que permitan la escalabilidad de estrategias para la construcción de índices sobre colecciones de documentos.

Se desarrollaron dos algoritmos MapReduce para construir índices de colecciones de datos no estructurados. Un índice básico fue utilizado como Baseline de los experimentos y otro de formato avanzado conocido como Block-Max fue construido para ser comparado con el primero.

También se construyo un cluster con hardware económico y se pudo constatar la viabilidad de ejecutar la plataforma Hadoop sobre el mismo. Se comprobó que las definiciones de "commodity hardware" difieren entre la literatura y el contexto de la Universidad. Se probó la robustez de Hadoop como plataforma en el uso intensivo de la misma, la cual escaló de forma apropiada hasta donde el limite de hardware se lo permitió.

Otras conclusiones extraídas de la realización del trabajo fueron:

- La métrica principal es el tiempo de construcción del índice. Es posible establecer que paralelizar la tarea de indexado vía Hadoop para los algoritmos construidos mejora los tiempos de construcción. La paralelización se realiza mediante el incremento de nodos y de reducers. Sin embargo, se comprobó que aumentar nodos y reducers no siempre conlleva una mejora. Bajo ciertas condiciones una menor cantidad de nodos o reducers mejoran el tiempo de construcción.

- Se calculo asimismo el Speedup de los algoritmos frente a las diferentes configuraciones del cluster. Se encontró que si se tomaban en cuenta los nodos se estaba frente a un falso caso de speedup superlineal. Se tomaron en cuenta los núcleos efectivamente disponibles para Hadoop.

Se encontró que el Speedup de los algoritmos no se ve afectado en la medida que los datos de entrada se incrementan. Tampoco se encontró impacto entre el Speedup y los diferentes tipos de algoritmos (BL y BM) ni en sus salidas (comprimidas y sin comprimir).

- Respecto a la métrica eficiencia, se pudo constatar que hasta un numero de núcleos la misma aumentaba. Sin embargo, a partir de cierto momento se ve que la misma se mantiene, y en algunos casos desciende. Esto aun cuando en todos los casos el Speedup sigue mejorando. Existe un valor experimental a partir del cual no es eficiente agregar mas recursos. Sin embargo, esta afirmación es condicionada por los datos de entrada, ya que la colección mas pequeña si mejoró la eficiencia para todos los casos.

- También se realizó un análisis del imbalance de carga por fase del proceso. Se observaron que en general el imbalance tiene una variación muy alta aun en la fase Map, donde la ejecución es idéntica para todos los algoritmos, lo que contradice la hipótesis de que esta fase seria similar para todos los experimentos. En la fase Reduce también se encuentra una alta variabilidad, aunque menor aun que en el caso de la fase Map. Por otro lado, con los experimentos realizados no se puede afirmar una relación entre el tamaño de las colecciones de entrada y el comportamiento de la métrica imbalance.

- Se estableció el uso de un modelo para la configuración inicial de recursos en un cluster Hadoop que permita ir ajustando, de forma experimental y continua, la forma en que el cluster es utilizado.

## 5.1. Trabajos Futuros

Todos los experimentos ejecutados para el presente trabajo mostraron un conjunto de resultados analizados anteriormente, pero también abrieron nuevos interrogantes:

- La cantidad de nodos y reducers disponibles probaron tener un efecto determinante en la ejecución de los algoritmos. Se considera que una linea de trabajo posible es estudiar como la mejora de los tiempos de procesamiento esta limitada por estos parámetros, siempre en un contexto de hardware de bajo costo. Además, es necesario estudiar como impacta el cambio de tamaño de colección en cada una de las combinaciones de nodos y reducers en una escala superior a la estudiada aquí.

- El uso de Ganglia como sistema de monitorización muestra ser adecuado. Sin embargo, como sistema de recolección de métricas presenta limitaciones claras. Es necesario diseñar un esquema de recolección de métricas distribuido que permita la gestión de los valores de hardware de los nodos de manera similar a como el JobHistory maneja las métricas de los procesos Hadoop.

- La métrica de imbalance muestra que aun siendo el mismo proceso en todos los experimentos, la fase map presenta una variabilidad alta en lo que respecta a la manera que se distribuyó la carga de trabajo. Es necesario estudiar si el planificador de tareas puede tener impacto en las métricas tomadas y estudiar como los diferentes parámetros que el planificador acepta impactan en las tareas ejecutadas.

- Otro estudio que puede ser interesante de realizar es el de aumentar la cantidad de discos disponibles por nodo y ver como impacta en el desempeño global de los algoritmos.

- Una cuestión que quedó sin respuesta es porque se degrada el rendimiento en la configuración 1N respecto de la 0N. La diferencia entre ambas configuraciones es la transferencia por red de los datos. Una hipótesis es que la velocidad de transferencia esta actuando como cuello de botella del proceso.

- En algunos experimentos se mostraron sutiles diferencias entre los algoritmos Baseline y Block-Max. La cantidad de pruebas realizadas no permite determinar cual de los 2 algoritmos tiene un rendimiento mayor. Una linea de trabajo es ejecutar mayor cantidad de pruebas que permitan una mejor comparación entre ambos. Algo similar ocurre con el formato de salida (con y sin compresión), aunque en este caso nunca se vio diferencias entre uno y otro.

- La combinación de cantidad de nodos reducers no responde a un modelo lineal donde aumentar ambos mejora el rendimiento. De hecho, no solo en algunos casos no mejora el Speedup ni la eficiencia, sino que se ve afectado directamente el tiempo de procesamiento.

# Bibliografía


[1] ITU-T Study Group 13. Big data - cloud computing based requirements and capabilities. recommendation y.3600. 2015.

[2] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[3] Claudine Santos Badue, Ricardo Baeza-Yates, Berthier Ribeiro-Neto, and Nivio Ziviani. Distributed query processing using partitioned inverted files. In *spire*, volume 1, pages 10–20, 2001.

[4] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval: the concepts and technology behind search*. Addison-Wesley Publishing Company, 2nd edition, 2011.

[5] David Bollier and Charles M Firestone. *The promise and peril of big data*. Aspen Institute, Communications and Society Program Washington, DC, 2010.

[6] Andrei Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. *Proceedings of the twelfth international conference on Information and knowledge management - CIKM '03*, pages 426–434, 2003.

[7] Stefan Büttcher and Charles LA Clarke. Index compression is good, especially for random access. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 761–770. ACM, 2007.

[8] Matteo Catena, Craig MacDonald, and Iadh Ounis. On inverted index compression for search engine efficiency. In *ECIR*, volume 8416 LNCS, pages 359–371. Springer, 2014.

[9] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of the 8th conference on*

*Visualization'97*, number July, pages 235–244. IEEE Computer Society Press, 1997.

[10] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: Information retrieval in practice*. Addison-Wesley Reading, 2010.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *OSDI*, 2004.

[12] Diego Desani, Veronica Gil-Costa, Cesar AC Marcondes, and Hermes Senger. Black-box optimization of hadoop parameters using derivative-free optimization. In *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*, pages 43–50. IEEE, 2016.

[13] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM Press, 2011.

[14] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:1–114, 2007.

[15] Esteban Feuerstein, Mauricio Marin, Michel Mizrahi, Veronica Gil-Costa, and Ricardo Baeza-Yates. Two-dimensional distributed inverted files. In *Lecture Notes in Computer Science*, volume 5721 LNCS, pages 206–213. Springer, 2009.

[16] William Frakes and Ricardo Baeza-Yates. *Information retrieval: data structures and algorithms*. 1992.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[18] Gregory Grefenstette and Pasi Tapanainen. What is a word, what is a sentence?: problems of tokenisation. 1994.

[19] Neil Gunther, Paul Puglia, and Kristofer Tomasette. Hadoop superlinear scalability. *Queue*, 13(5), may 2015.

[20] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, 2009.

[21] John Hennessy and David Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[22] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science (New York, N.Y.)*, 332(6025):60–65, 2011.

[23] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. Webbase: A repository of web pages. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks*. North-Holland Publishing Co., 2000.

[24] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment*, 3(1-2):472–483, 2010.

[25] Avita Katal, Mohammad Wazid, and R. H. Goudar. Big data: Issues, challenges, tools and good practices. In *2013 6th International Conference on Contemporary Computing, IC3 2013*, pages 404–409. IEEE, 2013.

[26] Woo-cheol Kim, Changryong Baek, and Dongwon Lee. Measuring the optimality of hadoop optimization. 2013.

[27] Aaron Kimball, Sierra Michels-Slettvet, and Christophe Bisciglia. Cluster computing for web-scale data processing. *ACM SIGCSE Bulletin*, 40(1):116–120, 2008.

[28] Chuck Lam. *Hadoop in action*. Manning Publications Co., 2011.

[29] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software - Practice and Experience*, 45(1):1–29, 2015.

[30] Jimmy Lin. Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail! *Big Data*, 1(1):28–37, 2013.

[31] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, jan 2010.

[32] Jimmy Lin, Dmitriy Ryaboy, and Kevin Weil. Full-text indexing for optimizing selection operations in large-scale data analytics. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 59–66. ACM, ACM Press, 2011.

[33] Christopher Manning, Prabhakar Raghavan, Hinrich Schütze, et al. Introduction to information retrieval. *An Introduction To Information Retrieval*, 151, 2008.

[34] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. *McKinsey Global Institute*, (June), 2011.

[35] Richard McCreadie, Craig Macdonald, and Iadh Ounis. Comparing distributed indexing: To mapreduce or not? *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, i(July):41–48, 2009.

[36] Richard McCreadie, Craig Macdonald, and Iadh Ounis. Mapreduce indexing strategies: Studying scalability and efficiency. *Information Processing & Management*, 48(5):873–888, 2012.

[37] Camilo Melani, Juan Echagüe, Joaquín Torre Zaffaroni, and Daniel Yankelevich. Un caso de big data punta a punta: análisis de datos de transporte y su uso en el negocio. *Proceedings de AGRANDA 2015 Simposio Argentino de GRANdes DAtos*, pages 11–13, 2016.

[38] Martin Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[39] Anand Rajaraman and Jeffrey Ullman. *Mining of massive datasets*, volume 67. 2011.

[40] Kai Ren, Garth Gibson, Yongchul Kwon, Magdalena Balazinska, and Bill Howe. *Hadoop's adolescence; a comparative workloads analysis from three research clusters*, pages 1452–1453. 2012.

[41] Berthier Ribeiro-Neto, Joao Paulo Kitajima, Gonzalo Navarro, Cláudio Sant'Ana, and Nivio Ziviani. Parallel generation of inverted files for distributed text collections. In *Computer Science, 1998. SCCC'98. XVIII International Conference of the Chilean Society of*, pages 149–157. IEEE, 1998.

[42] Berthier Ribeiro-Neto, Edleno Moura, Marden Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 105–112. ACM, 1999.

[43] Berthier A Ribeiro-Neto and Ramurti A Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM conference on Digital libraries*, pages 182–190. ACM, 1998.

[44] Viktor Mayer Schönberger and Kenneth Cukier. *Big data: la revolución de los datos masivos*. Turner, 2013.

[45] Jeffrey Shafer, Scott Rixner, and Alan L Cox. The hadoop distributed filesystem: Balancing portability and performance. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 122–133. IEEE, 2010.

[46] Konstantin Shvachko. Hdfs scalability: The limits to growth. *login*, 35(2):6–16, 2010.

[47] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. IEEE, 2010.

[48] Jason Venner. *Pro Hadoop*. Apress, 2009.

[49] Tom White. *Hadoop: The definitive guide*, volume 54. O'Reilly Media, Inc., 4th edition, 2015.

[50] Ian Witten, Alistair Moffat, and Timothy Bell. *Managing gigabytes: compressing and indexing documents and images*, volume 41. Morgan Kaufmann, 2nd edition, 1999.

[51] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2), 2006.