Estudio del mecanismo de redistribución de datos shuffle en Spark

Trabajo de Fin de Grado
Grado en Matemáticas y Estadística
Curso 2022-2023

Universidad Complutense de Madrid
Facultad de Ciencias Matemáticas
Departamento de Sistemas Informáticos y Computación

Realizado por: Sara García Cabezalí
Tutelado por: Luis Fernando Llana Diaz
Madrid 2023

Índice general

Resumen

El objetivo del estudio es analizar en profundidad el funcionamiento del mecanismo de redistribución de datos *shuffle* en el framework de procesamiento distribuido Apache Spark™. Para ello, se analizaron 90 muestras aleatorias de tamaños incrementales generadas a partir del fichero de datos GSOD.txt (19 GB). El análisis se desarrolló con el lenguaje de programación Python en un clúster de 6 nodos, de 4 núcleos y 8 GB de RAM cada uno. Los experimentos diseñados fueron a) impacto general del proceso *shuffle* en el rendimiento general del sistema, b) diferencias en rendimiento en cuanto a *shuffle* para distintas transformaciones *wide* o amplias y c) el impacto del grado de paralelismo o número de particiones de los datos en el rendimiento de las aplicaciones. Las métricas utilizadas en el proceso fueron el tiempo de ejecución, uso de memoria y número de registros escritos y leídos por los ejecutores durante el proceso *shuffle*. Los resultados muestran diferencias sustanciales en el tiempo de ejecución de las aplicaciones que desencadenan el proceso *shuffle* en comparación con el resto. Dentro de las transformaciones *wide* o amplias, debería elegirse las transformaciones que ejecutan una etapa de agregación antes de distribuir los datos para el proceso *shuffle*. Considerando que el número de particiones definidas por defecto por Spark no es el óptimo, variar el número de éstas puede conducir a una mejora considerable del rendimiento de las aplicaciones Spark. Los resultados de este estudio pueden utilizarse como guía o ejemplo con el fin de entender y optimizar la utilidad de la herramienta Spark.

Abstract

The objective of the study is to analyze in-depth the behaviour of the data redistribution mechanism *shuffle* in the Apache Spark distributed processing framework. 90 random samples of incremental sizes generated from the GSOD.txt data file (19 GB) were analyzed. The analysis was carried out using the Python programming language on a cluster of 6 nodes, each with 4 cores and 8 GB of RAM. The experiments designed were a) the overall impact of the *shuffle* process on the general system performance, b) performance differences in terms of *shuffle* for various wide transformations, and c) the impact of the degree of parallelism on the performance of the applications. The metrics used in the process were execution time, memory usage, and the number of records written and read by the executors during the *shuffle* process. The results show substantial differences in the execution time of the applications that trigger the *shuffle* process compared to the rest. Among the wide or broad transformations, those that execute a stage of aggregation before distributing the data for the *shuffle* process should be chosen. Considering that the number of partitions defined by default by Spark is not optimal, varying the number of these can lead to a considerable improvement in the performance of Spark applications. The results of the study can be used as a guide or example in order to understand and optimize the utility of the Spark tool.

Capítulo 1

Introducción

1.1. Contexto.

En los últimos años, el surgimiento de grandes volúmenes de datos ha generado la necesidad de encontrar soluciones eficientes para manipular y analizar estos conjuntos de datos de manera distribuida. El Big Data abarca una amplia variedad de datos provenientes de diversas fuentes, presentando desafíos en términos de almacenamiento, transmisión, gestión, procesamiento y análisis.

Apache Spark es una herramienta de gran utilidad que permite realizar operaciones en paralelo y distribuir el procesamiento de datos en clusters, facilitando así la manipulación de grandes volúmenes de información.

El estudio del mecanismo de redistribución de datos *shuffle* en Spark es un tema relevante en el contexto del Big Data.

1.2. Planteamiento del problema

La distribución de datos en clusters conlleva un efecto secundario conocido como *shuffle*, que se produce cuando los datos deben ser redistribuidos entre los nodos del clúster para realizar ciertas operaciones. Este proceso tiene un impacto significativo en el rendimiento. Comprender y estudiar este mecanismo de redistribución de datos *shuffle* en Spark se antoja fundamental en los procesos de manipulación y análisis de Big Data.

En resumen, Apache Spark y el efecto *shuffle* son de gran utilidad y relevancia para el procesamiento de grandes volúmenes de datos, lo que conlleva también desafíos asociados.

1.3. Objetivos

El objetivo general de este trabajo es analizar y comprender en profundidad el funcionamiento del mecanismo de redistribución de datos *shuffle* en el framework de procesamiento distribuido Apache Spark. El estudio se centrará en investigar los aspectos

teóricos y prácticos de este mecanismo, así como en evaluar su impacto en el rendimiento y la eficiencia de las aplicaciones Spark.

Los objetivos específicos son:

- Investigar y comprender los conceptos fundamentales de Apache Spark y su modelo de procesamiento distribuido. Estudiar en detalle el mecanismo de redistribución de datos *shuffle* en Spark, incluyendo su funcionamiento interno y algoritmos utilizados.
- Diseñar y llevar a cabo experimentos capaces de analizar el impacto de la redistribución de datos *shuffle* en el rendimiento y la eficiencia de las aplicaciones Spark, considerando factores como el tamaño de los datos y las características de la aplicación.
- Proponer posibles mejoras o ajustes en el mecanismo de redistribución de datos *shuffle* de Spark, con el fin de optimizar su utilidad.

1.4. Estructura del TFG

Este trabajo consta principalmente, de dos bloques: un bloque teórico que presenta los concepto fundamentales de Apache Spark, abordando su arquitectura como modelo de procesamiento distribuido y el mecanismo de redistribución de datos *shuffle* (Capítulo 2), y un bloque práctico en el que se llevan a cabo diversas implementaciones de programas y aplicaciones Spark en *PySpark*, donde se ejecutan experimentos para evaluar el rendimiento y la eficiencia en diferentes escenarios (Capítulos 3 y 4).

Los resultados obtenidos son resumidos y analizados, proporcionando conclusiones relevantes para el futuro diseño y la implementación de aplicaciones Spark (Capítulo 5).

Los *scripts* diseñados y utilizados para el estudio serán recopilados en el repositorio público de GitHub junto con los j*upyter-notebooks* que incluyen los procesamientos gŕaficos: ***https://github.com/sargar19/trabajo-fin-de-grado***.

Capítulo 2

Marco teórico

Este capítulo tiene como objetivo principal realizar un análisis de la literatura existente sobre Apache Spark y la redistribución de datos *shuffle*. Su propósito es proporcionar los recursos necesarios para comprender el estudio realizado en este trabajo de fin de grado. Se introducirá el sistema Spark y se explicará en detalle el mecanismo *shuffle* en sistemas de procesamiento distribuido, abarcando la descripción general del proceso, la investigación de estrategias de optimización y el análisis del impacto en el rendimiento. A través de este análisis, se busca obtener una comprensión más profunda de esta etapa clave y su relevancia en el procesamiento eficiente de las aplicaciones Spark.

2.1. El sistema de computación Apache Spark.

2.1.1. Introducción a Apache Spark.

Apache Spark es un sistema de computación distribuida de código abierto diseñado para procesar grandes volúmenes de datos. Tiene la capacidad de realizar tareas de procesamiento en memoria, lo que es de gran utilidad para aplicaciones que requieren un alto rendimiento en el procesamiento de datos.

Una de las principales características de Apache Spark es su capacidad para procesar datos en paralelo, distribuyendo las tareas en clústeres de máquinas. Esto proporciona una escalabilidad horizontal: el procesamiento de datos se puede distribuir en múltiples nodos de un clúster, lo que dota así al procesamiento de paralelismo y mejora el rendimiento general.

En el contexto del proceso *shuffle*, Apache Spark utiliza esta operación para realizar la redistribución de datos entre los nodos del clúster. *Shuffle* es una etapa decisiva en el procesamiento distribuido, donde los datos se agrupan, se mezclan y se redistribuyen para llevar a cabo operaciones como la agregación (i.e. el *join*). Esta etapa implica una transferencia significativa de datos entre los nodos y puede tener un impacto importante en el rendimiento general del sistema.

En resumen, Apache Spark es un sistema de computación distribuida que incluye librerías para diversas tareas. Este sistema se puede ejecutar en diversos entornos, desde un ordenador hasta un cluster constituido por miles de servidores (Figura 2.1).

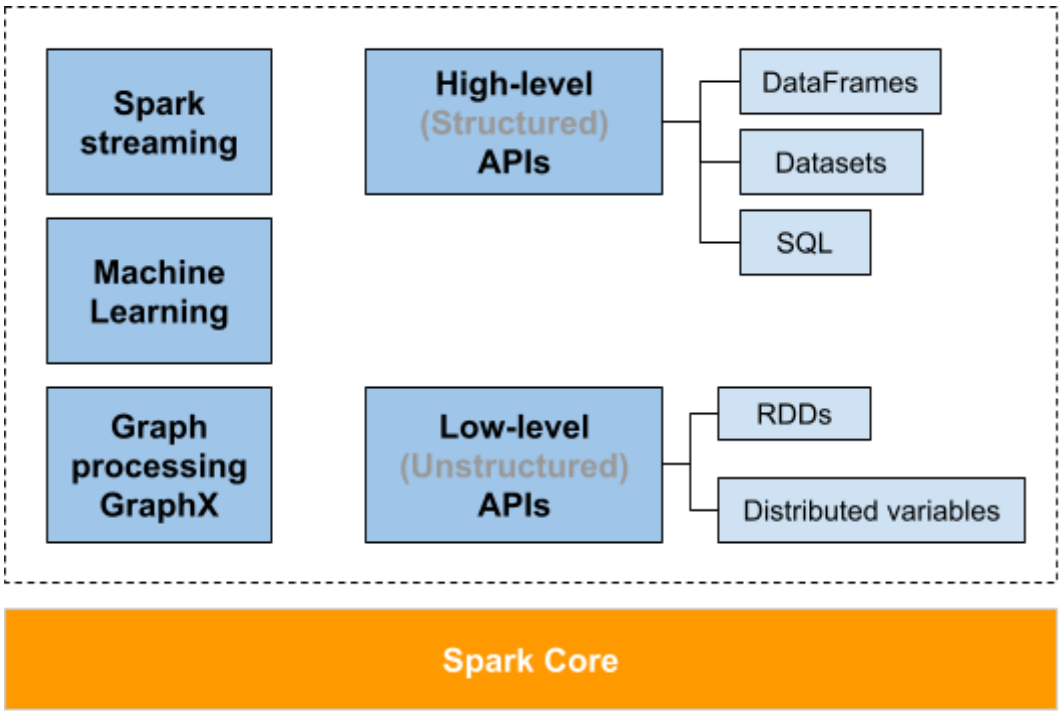

Figura 2.1: Componentes principales que conforman la infraestructura de Apache Spark.

2.1.2. Abstracciones fundamentales de Spark.

Spark proporciona varias abstracciones fundamentales para un procesamiento de datos eficiente y escalable, siendo las principales:

1. ***DataFrames***: Los *DataFrames* en Spark son una estructura de datos tabular distribuida que representa una colección organizada de datos en columnas con nombre.

2. ***Datasets***: Los *Datasets* en Spark proporcionan una abstracción fuertemente tipada de datos estructurados. Se basa en los *DataFrames*, pero agrega información adicional sobre los tipos de datos en tiempo de ejecución.

3. ***Resilient Distributed Datasets*** (**RDDs**): Los RDDs son la abstracción de datos más básica de Spark. Representan una colección inmutable y particionada de registros que se pueden operar en paralelo mediante operaciones de transformación y acción (Capítulos 2.1.5 y 2.1.7). Además, son tolerantes a fallos[1] o *fault tolerant*; al registrar en memoria las transformaciones utilizadas para construir un conjunto de datos en lugar de datos reales, es capaz de volver a calcular una partición de un RDD perdida, pues tiene suficiente información de cómo se derivó de otros RDDs (su *linaje*).

4. ***SQL Tables***: Spark permite registrar *DataFrames* y RDDs como tablas SQL, lo que permite realizar consultas utilizando el lenguaje SQL estándar.

[1] La tolerancia a fallos se refiere a la capacidad de un sistema (computadora, red o clúster) para continuar funcionando sin interrupción cuando uno o más componentes fallan.

Aunque los RDDs proporcionan un mayor control sobre la distribución física de los datos y son más flexibles en términos de tipos de objetos, son una API de nivel inferior en comparación con los *DataFrames* y los *Datasets*. Esto se debe a que los RDDs carecen de algunas optimizaciones de rendimiento y características de seguridad de tipos presentes en las APIs de alto nivel. En resumen, los RDDs brindan un mayor control pero requieren más esfuerzo y conocimiento para optimizar el procesamiento interno. Por lo tanto, solo se recomienda su uso cuando se necesita un control más fino sobre el procesamiento y la distribución de datos, como es en el caso de este trabajo.

2.1.3. Arquitectura básica de Spark.

Un clúster, o grupo de computadoras, agrupa los recursos de muchas máquinas juntas, ofreciendo la capacidad de usar todos los recursos acumulados como si fueran una sola computadora. Sin embargo, un grupo de máquinas por sí solo no es tan fuerte sin un marco donde coordinar tareas de datos a través de un grupo de computadoras. Esto es lo que hace Spark a través del programa conductor, del gerente del clúster y de los ejecutores (Figura 2.2).

El **programa conductor** o *Driver Program*, coordina la ejecución de toda la aplicación. Entre sus responsabilidades se encuentran las siguientes funciones:

1. Definir y configurar una instancia *SparkContext*.
2. Dividir la aplicación en fases y tareas (Capítulo 2.1.8).
3. Planificar y distribuir las tareas entre los ejecutores.
4. Hacer un seguimiento del progreso de ejecución, recopilar los resultados y manejar errores, siendo responsable de responder al programa del usuario.

Spark emplea también a un **administrador de clústeres**[2] o *cluster manager*. Este solicita y gestiona los recursos disponibles del clúster (CPU, memoria, etc.) en función de los requisitos del Driver. Es responsable también de la gestión de errores y su reparación. Si un ejecutor falla o no responde, el gerente de clúster toma la acción apropiada: reinicia el ejecutor o inicia uno nuevo para garantizar la tolerancia a fallos.

Por otro lado, los **ejecutores**[3] o *executors* (situados en los nodos esclavos proporcionados por el cluster manager) son responsables de ejecutar las tareas que el Driver les asigna. Esto involucra almacenar y mezclar datos en memoria o en disco. Además, deben informar al Driver del progreso de las tareas asignadas.

[2] El propio Spark, Mesos o Yarn.
[3] Los ejecutores siempre estarán ejecutando código Spark (Scala). Sin embargo, el Driver puede ser dirigido a través de múltiples lenguajes distintos.

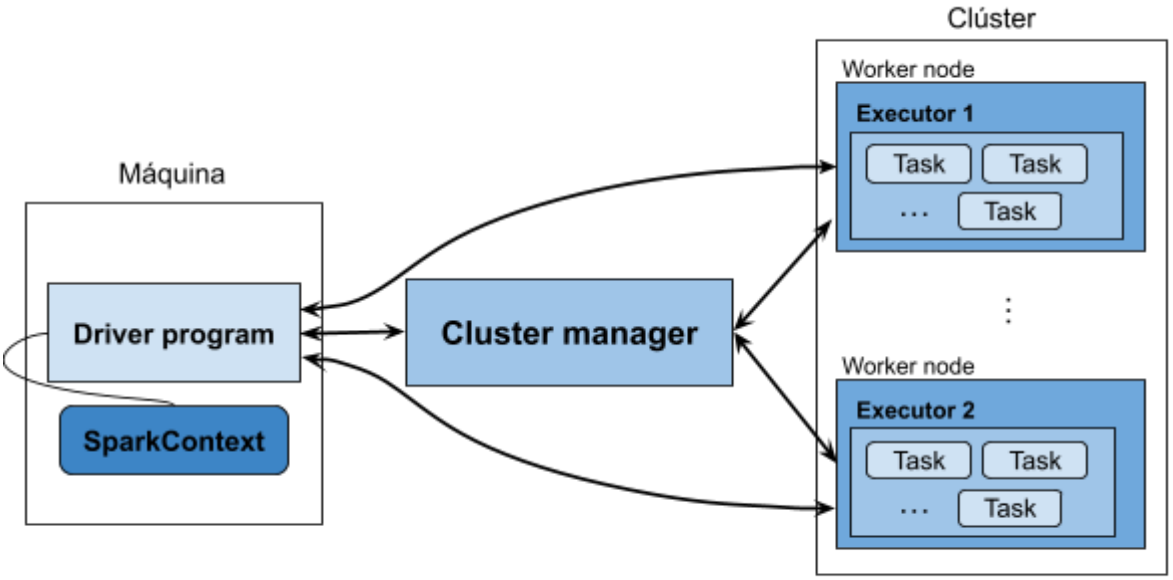

Figura 2.2: Coordinación de la arquitectura de Apache Spark.

Nota: Spark además de su modo de clúster, también tiene un modo local. El controlador y los ejecutores son simplemente procesos, en otras palabras, pueden vivir en la misma máquina o en diferentes máquinas. En modo local, se ejecutan (como subprocesos) en su computadora individual en lugar de un clúster.

2.1.4. Particiones.

Las particiones (Figura 2.3) son una de las unidades fundamentales de procesamiento en Spark. Se utilizan para dividir un conjunto de datos en fragmentos más pequeños y distribuirlos a través de un cluster de forma que cada partición es procesada en paralelo por un ejecutor en el clúster. Esto permite procesar grandes conjuntos de datos de manera más eficiente y escalable. Dicha distribución de los datos puede ser totalmente aleatoria o estar basada en cierto esquema.

Existen dos aspectos importantes sobre este esquema de particionamiento:

- El número de particiones en el que se distribuyen los datos.
- Cómo los datos se distribuyen entre las distintas particiones.

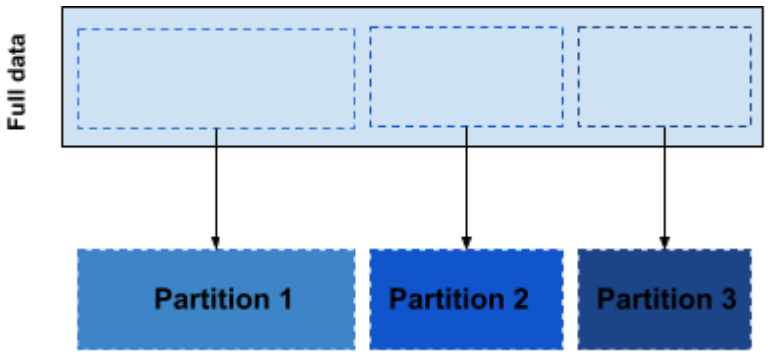

Figura 2.3: Descripción gráfica del particionamiento de datos.

En cuanto a las estrategias o técnicas de particionamiento de datos, estas pueden ser completamente aleatorias o extremadamente refinadas. Spark ofrece dos implementaciones muy utilizadas, *Hash* y *Range*, aunque también se pueden definir particionadores a medida.

Por una parte, el particionamiento *Hash* (particionador por defecto en Spark) distribuye los datos entre las distintas particiones en función del resultado de evaluar la clave sobre la función *hash*. Es decir, un dato es situado en una partición basada en el valor evaluado para ese dato, distribuyendo así los datos de manera uniforme en varias particiones sobre la base de una clave.

Por otro lado, el particionamiento de rango o *Range* determina inicialmente un rango para cada una de las particiones. Para ello, se toma una muestra aleatoria de los datos a particionar para construir una visualización del rango global de valores. Una vez esta visualización está construida, el rango global es distribuido a través de las particiones de manera uniforme. Una vez definido el rango para cada una de las particiones, se distribuyen cada una de las instancias de los datos basados en estos. Este es útil cuando se sabe que las claves tienen una distribución desigual, ya que intenta equilibrar la cantidad de registros en cada partición.

Por último, Spark permite definir un particionador customizado. Se utiliza cuando se requieren distribuciones específicas de los registros. Por ejemplo, agrupar registros basados en ciertos patrones en las claves.

En general, se recomienda tener un número adecuado de particiones para un conjunto de datos dado, ya que demasiadas particiones pueden provocar un exceso de sobrecarga en la comunicación entre los ejecutores. Más aún, demasiadas pocas particiones pueden resultar en la utilización ineficiente de los recursos del clúster. Una elección adecuada del *partitioner* puede tener un impacto significativo en el rendimiento, asegurando una utilización eficiente de los recursos.

2.1.5. Transformaciones.

Formalmente, un RDD es una colección de registros particionados, inmutables y de solo lectura. Los RDDs solo se pueden crear a través de operaciones deterministas sobre:

- Datos en almacenamiento estable u
- Otros RDDs.

Llamamos a estas operaciones **transformaciones** para diferenciarlas de otras operaciones sobre RDDs. Entre estas transformaciones se encuentran *map*, *filter* o *union*, entre otras.

Es importante entender que las transformaciones no devolverán la salida o *output*. Esto se debe a que Spark no actuará sobre las transformaciones hasta que llamemos a una acción (lo que se conoce como evaluación perezosa o *Lazy evaluation*, explicada con más detalle en el siguiente apartado).

Podemos clasificar las transformaciones en dos tipos:

1. Transformaciones *Narrow* o estrechas: los registros requeridos para procesar los datos de una partición única están disponibles en la misma partición. Así, cada partición de entrada contribuirá a una sola partición de salida. Es decir, existe una correspondencia 1 a 1 entre las particiones de entrada y salida (Figura 2.4). Un ejemplo de transformación *Narrow* sería la transformación *filter()*. En este caso, Spark realizará automáticamente una operación llamada *pipelining*. Todas las transformaciones de este tipo se realizan en memoria.

2. Transformaciones *Wide* o amplias: los registros que son necesarios para procesar los datos de una partición única pueden estar presentes en varias particiones. Así, las particiones de entrada contribuyen a múltiples particiones de salida, es decir, existe una correspondencia 1 a N entre las particiones de entrada y salida (Figura 2.4), como por ejemplo la transformación *reduceByKey()*. En este caso, a diferencia del tipo de transformaciones anterior, Spark requiere de la operación conocida como *shuffle* (motivo principal de estudio de este trabajo) que intercambia datos de las particiones a lo largo del clúster. Son más costosas en términos de rendimiento porque involucran la transferencia de datos a través de la red y operaciones de I/O con el disco.

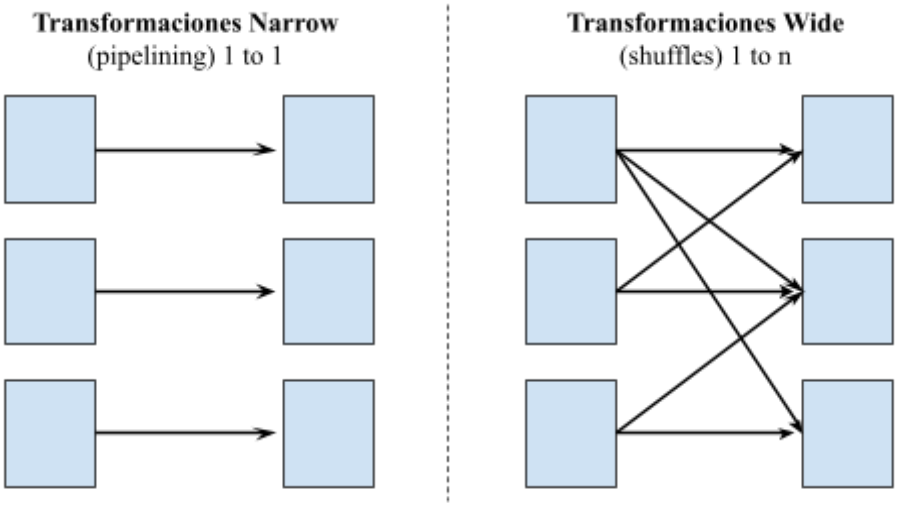

Figura 2.4: Diagrama caracterizante de las transformaciones *Narrow* (izquierda) *y transformaciones Wide* (derecha).

2.1.6. Evaluación perezosa o *Lazy evaluation*.

Como se ha mencionado en el apartado anterior, no es necesario que los RDDs se materialicen en todo momento. En cambio, un RDD tiene suficiente información sobre cómo se derivó de otros conjuntos de datos (su linaje) para calcular sus particiones a partir de datos en almacenamiento estable. También se pueden hacer control de persistencia y particionado.

En lugar de modificar los datos inmediatamente después de expresar cierta operación sobre los objetos, Spark almacenará cada una de estas transformaciones sobre el objeto inicial y construirá un plan o gráfico lógico de cálculo. Este no será compilado hasta que una acción sea llamada sobre el objeto final (Figura 2.5). Esto es lo que se conoce como evaluación perezosa o *Lazy evaluation*.

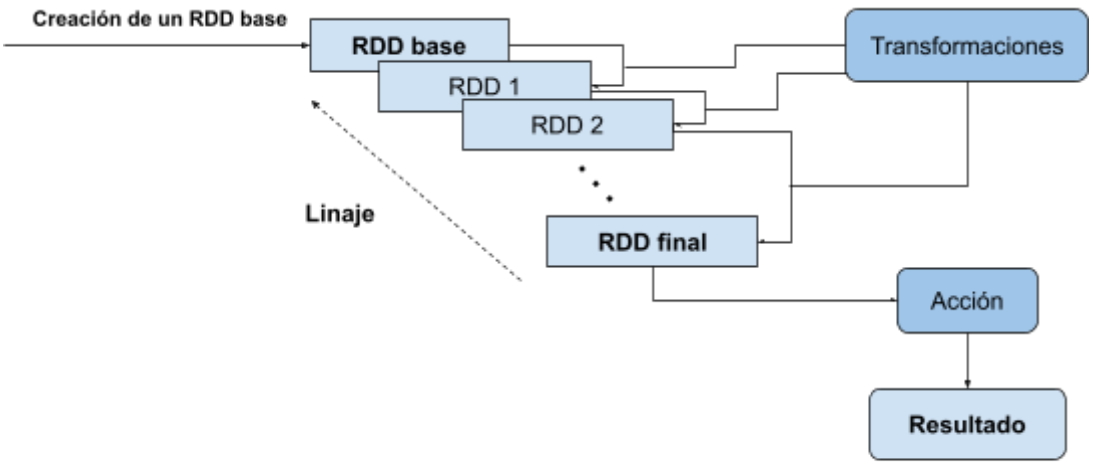

Figura 2.5: Explicación gráfica de la evaluación perezosa o *Lazy evaluation* sobre los RDDs.

Al esperar hasta el último minuto para ejecutar el código, Spark convierte el plan lógico de transformaciones en un plan físico y real, que se ejecutará de la manera más

eficiente posible en todo el clúster. Esto ofrece grandes beneficios sobre el desempeño del procesamiento de datos, ya que permite a Spark optimizar el flujo de datos por completo, de principio a fin.

2.1.7. Acciones.

Como se ha visto en las secciones anteriores, las transformaciones nos permiten construir un plan lógico de operaciones. Sin embargo, para originar la computación de este plan, necesitamos ejecutar una *acción* sobre el objeto final. Las acciones ordenan a Spark a computar el resultado de una serie de transformaciones y devuelven el valor resultante al *Driver*.

Podemos clasificar las acciones en tres tipos[4]:

1. Acciones que permiten ver los datos en consola (ej. *collect, take, sample, count*.)
2. Acciones para recoger los datos en objetos nativos del lenguaje de programación respectivo.
3. Acciones que nos permiten escribir a fuentes de datos (ej. *saveAsTextFile*).

2.1.8. DAGScheduler: trabajos, etapas y tareas.

Como se ha indicado anteriormente, el linaje de dependencias entre RDDs está definido por el encadenamiento de operaciones y constituye un plan lógico de transformaciones que, más tarde, se transforma en un plan físico de ejecuciones, de tipo etapa-orientado. Esto es responsabilidad de lo que se conoce como *DAGScheduler* o *Directed Acyclic Graph Scheduler* (Figura 2.6).

[4] Para conocer todas las acciones existentes, consultar la documentación oficial de Spark.

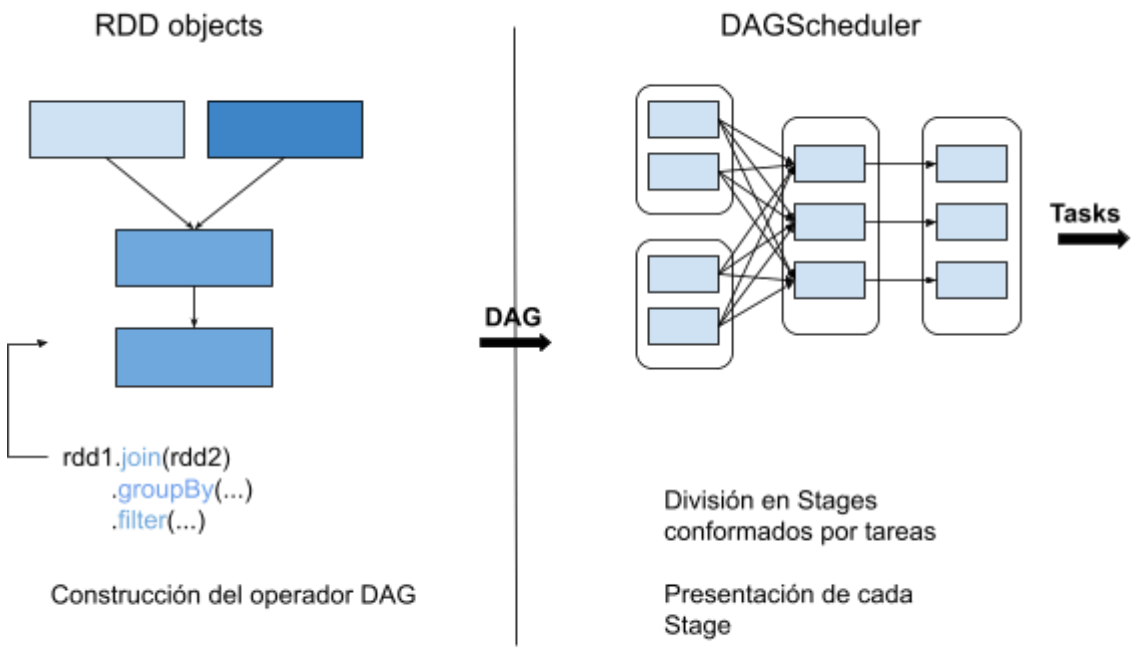

Figura 2.6: Transformación del plan lógico al plan físico de ejecuciones.

La llamada de una acción junto con la ejecución de su plan lógico generan lo que se denomina como trabajo o *Job*. El *DAGScheduler* divide este trabajo en una colección de etapas o *Stages*. Cada una de estas etapas está compuesta por un conjunto de tareas o *Tasks* paralelizables, que se distribuyen entre los distintos ejecutores (Figura 2.7). Nótese que en Spark, **una tarea es la unidad de ejecución individual más pequeña** ya que solo opera sobre una única partición del RDD. Esto es lo que permite una ejecución paralela de las tareas de Spark (es necesario que todas las tareas de una etapa se completen antes de que comiencen las etapas siguientes).

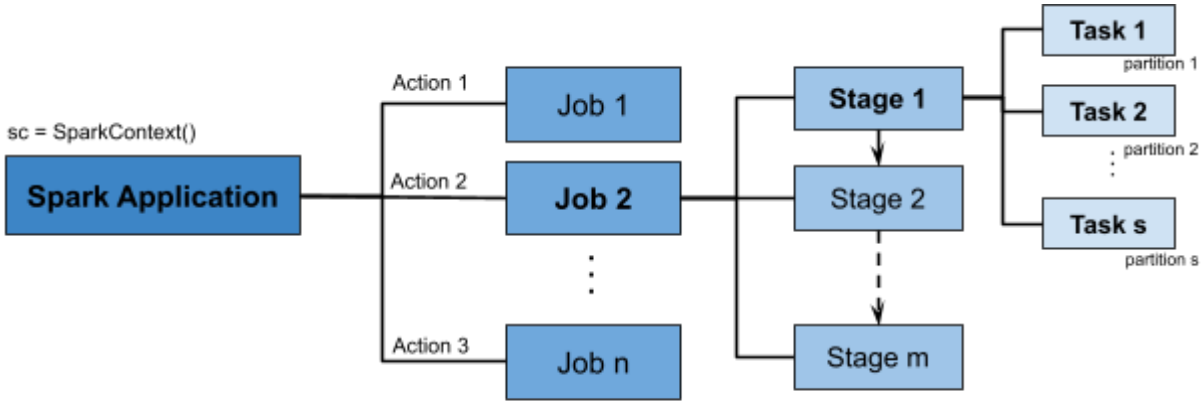

Figura 2.7: División de las aplicaciones en trabajos, estados y tareas.

Cada una de estas etapas contiene una secuencia de transformaciones *narrow* que pueden ser completadas sin realizar mezcla de datos (*shuffle*) en el *Dataset*. Estas operaciones son conectadas internamente (*pipelined*) en un único conjunto de tareas en

cada etapa, pero las operaciones con dependencia *wide* (*shuffle*), requerirán más de una etapa.

En resumen, una aplicación Spark consta de tantos trabajos como acciones se ejecuten y cada uno de estos trabajos se descompondrá en una o varias etapas. A su vez, estas etapas se dividen en tareas individuales, que son enviadas a los ejecutores de Spark que se encuentran en los nodos del clúster. A menudo varias tareas se ejecutarán en paralelo en el mismo ejecutor, cada una procesando una partición de los datos.

Spark ofrece una interfaz gráfica, Spark UI[5], que permite al usuario consultar el estado y el uso de los recursos de un clúster Spark, así como el gráfico DAG y el estado de la ejecución del plan físico de ejecuciones.

2.2. Descripción general del proceso shuffle.

Como se ha mencionado anteriormente, ciertas operaciones desencadenan un evento conocido como *shuffle*. El proceso *shuffle* en Spark se da cada vez que existe la necesidad de redistribuir una colección distribuida de datos entre nuevas particiones, ya sea representada por un RDD, un *DataFrame* o un *Dataset*. Esta necesidad puede deberse a cualquiera de las dos siguientes situaciones:

1. Que se quiera **incrementar o reducir el número de particiones de datos**. Como se ha explicado en la sección anterior, puesto que una partición de datos en Spark representa la cantidad de datos procesados por una única tarea de Spark, y por tanto el grado de paralelismo, podemos encontrarnos con cualquiera de las siguientes situaciones:

    (a) El número existente de particiones es demasiado pequeño como para optimizar el uso de los recursos disponibles y cubrir el número de ejecutores disponibles.

    (b) Las particiones existentes son demasiado pesadas, generando desbordamientos de memoria.

    (c) El número existente de particiones es demasiado grande, haciendo que la planificación de tareas se vuelva el cuello de botella del tiempo total de procesamiento.

    (d) Los datos no están distribuidos a partes iguales en las distintas particiones, lo que causa lo que se conoce como *Data Skew* (Figura 2.8). En muchas ocasiones los datos se dividen en particiones basadas en una clave, como por ejemplo la primera letra de un nombre. Si los valores de esta clave no están distribuidos de forma equitativa en los datos, entonces más datos serán situados en una partición que en otra.

[5] Para una explicación detallada, consultar la documentación oficial de la *Spark UI* en el enlace https://spark.apache.org/docs/3.0.0-preview2/web-ui.html.

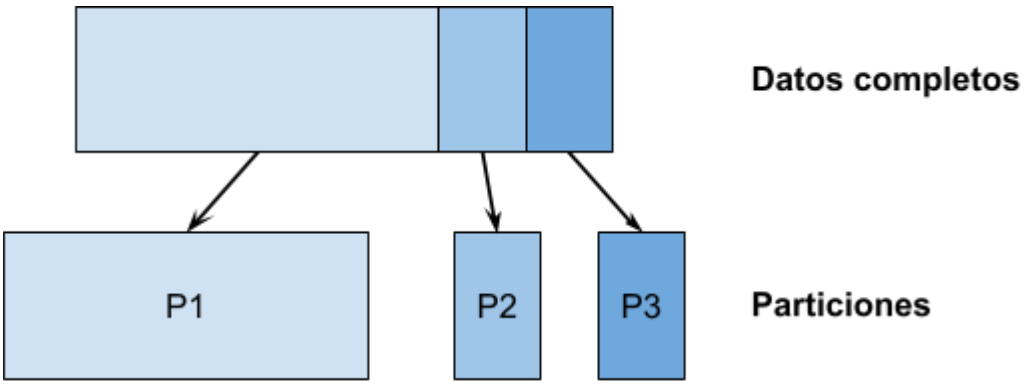

Figura 2.8: Ejemplo visual de particiones desbalanceadas.

En cualquiera de los casos anteriores, se recomienda redistribuir los datos para mejorar el rendimiento de nuestra aplicación Spark, lo que activará de manera secundaria el efecto *shuffle*.

2. Aplicar **transformaciones *Wide*** a nuestro RDD. Para visualizar con más claridad este escenario, consideremos el siguiente ejemplo:

   Ejemplo 1: Imaginemos que nuestro RDD contiene los datos sobre las edades de los delegados de distintos grados de la Facultad de Ciencias Matemáticas de Madrid, almacenados en tuplas clave, valor, donde la clave es el nombre del Grado que el estudiante está cursando y el valor es la edad de este. Figuremos que los datos están distribuidos en 3 particiones como se muestra en la Figura 2.9 y que se quiere calcular la edad media de los delegados en cada uno de los grados.

   Entonces, para cada uno de los grados, pongamos el Grado de “Matemáticas - Física”, se tiene que acceder a cada una de las particiones para agrupar las instancias de esta clave, o los delegados de este Grado y así poder calcular la media de todas las edades. Y de forma idéntica para todos los demás grados o claves.

   Es decir, las transformaciones *Wide* involucran el acceso a datos que habitan en muchas particiones, lo que implica el movimiento de estos entre las distintas particiones.

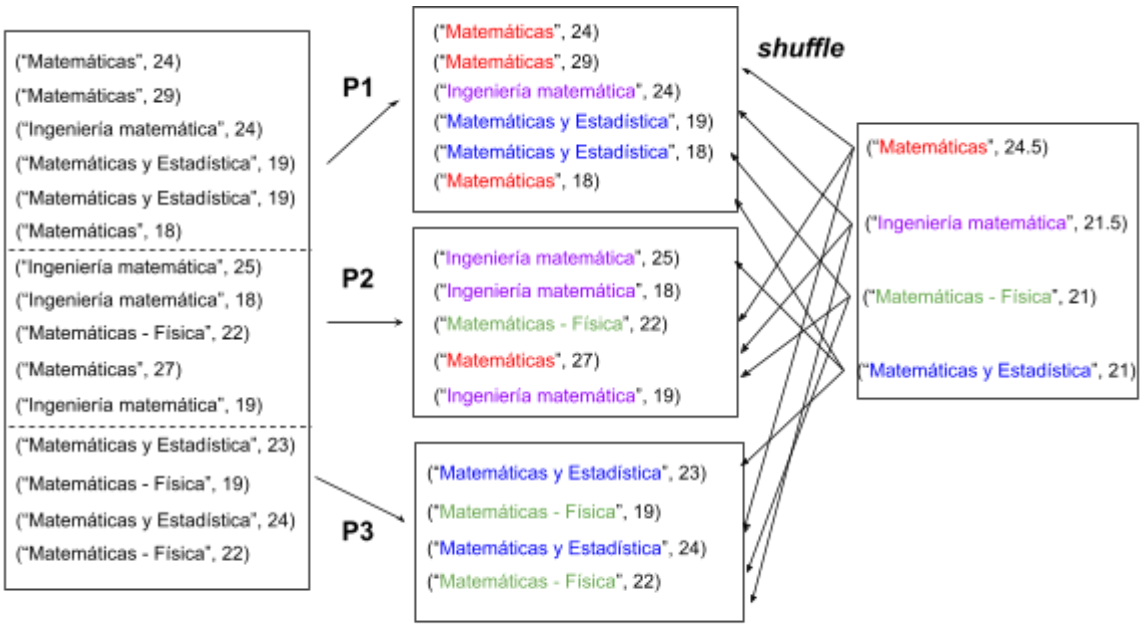

Figura 2.9: Datos del Ejemplo 1 distribuidos en 3 particiones.

Por tanto, las operaciones que causan la activación del proceso *shuffle*, incluyen:

1. Operaciones de repartición como *repartition()* o *coascele()*,
2. Operaciones por clave, como *groupByKey()* y *reduceByKey()*,
3. Operaciones *join*, como *cogroup()* y *join()*.

Es importante tener en cuenta que Spark **aumenta automáticamente a 200** el número de particiones cuando se desencadena el proceso *shuffle* sobre nuestros datos. Por lo tanto, encontrar el número indicado de particiones *shuffle* tiene un impacto directo en el rendimiento de tu aplicación Spark. Una recomendación común es definir 4 particiones por CPU, aunque el desempeño de Spark depende de cada caso particular.

2.2.1. Shuffle Read y Shuffle Write.

Cuando la operación *shuffle* se lleva a cabo, se introducen 2 etapas en la aplicación Spark, que se conocen como *Shuffle Read* y *Shuffle Write*.

1. *Shuffle Write*.

Spark redistribuye los datos para que todos los registros con la misma clave terminen localizados en la misma partición. Como se ha mencionado en el Capítulo 2.1.4, Spark utiliza como particionador por defecto *Hash Partitioner*. De este modo, las salidas de las tareas anteriores se dividen en distintas particiones. Cada una de estas particiones estará destinada a una tarea específica en la etapa siguiente.

Los registros se almacenan inicialmente en un buffer en memoria. Cuando este buffer se llena, su contenido se clasifica por partición y se escribe en el disco en un archivo intermedio.

Una vez que una tarea ha escrito sus datos de *shuffle*, registra los metadatos de estos datos (en qué archivo están, en qué desplazamiento y cuántos bytes ocupa, etc) en un mapa *shuffle*. Este mapa es crucial para la siguiente etapa, ya que informa de cómo recuperar los datos de entrada. Los datos escritos en el proceso de *Shuffle Write* se almacenan localmente en el nodo donde se ejecuta la tarea. No se transfieren a otros nodos hasta que se necesiten durante el *Shuffle Read*.

2. *Shuffle Read*.

El proceso *Shuffle Read* comienza cuando una tarea en una etapa posterior necesita datos que fueron escritos por tareas en una etapa anterior. Antes de que pueda comenzar la lectura, la tarea necesita saber de dónde obtener sus datos. La tarea consulta el mapa de *shuffle* mencionado en el apartado anterior, para determinar la localización de los datos de cada partición:

- Si los datos requeridos están en el mismo nodo, la tarea lee esos datos desde el almacenamiento local.
- Si los datos están en un nodo diferente, se produce una transferencia de red. La tarea envía una solicitud al nodo remoto y recupera los datos a través de la red.

Una vez que los datos están disponibles localmente, si la operación es una agregación, la tarea combina los valores de todos los registros con la misma clave utilizando la función proporcionada. Es importante notar que Spark intenta, en la medida de lo posible, programar tareas en los nodos donde se encuentran sus datos, para reducir la necesidad de transferencias a través de la red.

2.2.2. Impacto del shuffle en el rendimiento y estrategias de optimización.

Aunque *shuffle* es esencial para muchos tipos de transformaciones, puede ser costoso en términos de tiempo y recursos. Los principales impactos en el rendimiento son los siguientes:

- Latencia de Red: si los datos se transfieren entre nodos en un clúster, puede haber una latencia significativa debido a la transferencia de datos a través de la red.
- I/O en Disco: durante el proceso *shuffle*, los datos se escriben en el disco y luego se leen nuevamente, lo que puede generar una sobrecarga significativa de I/O.
- Desbalance de Datos: si los datos no están uniformemente distribuidos entre las particiones, puede llevar a desbalances, donde algunos nodos terminan procesando más datos que otros.

Para ayudar a mitigar estos efectos negativos pueden llevarse a cabo estrategias de optimización como:

1. **Ajustar el número de particiones**: usando la configuración de Spark, se pueden definir el número de particiones generadas *post-shuffle*. Ajustarlo adecuadamente puede reducir la cantidad de datos transferidos y mejorar la paralelización.

2. **Uso de operaciones eficientes**: elegir operaciones como *reduceByKey* en lugar de *groupByKey*, ya que la primera realiza operaciones de reducción antes del *shuffle*, reduciendo la cantidad de datos transmitidos.

3. **Persistencia y caché**: a la hora de realizar múltiples operaciones en un conjunto de datos que ha sido "shuffleado", considerar cachearlo o persistirlo para evitar *shuffles* repetidos.

4. **Monitorización**: utilizar la Spark UI para monitorizar las operaciones de *shuffle*.

En resumen, este proceso conlleva la escritura de datos en disco, y, por lo general, implica transferir datos entre ejecutores y máquinas, lo que hace de esta etapa una **operación compleja y costosa**. Sin embargo, aunque *shuffle* es una operación costosa, es una etapa esencial y prevalente en el procesamiento de datos redistribuidos en Spark.

El proceso *shuffle* es de gran interés e importancia en las aplicaciones Spark, ya que tiene un gran impacto en el desempeño de la aplicación.

Capítulo 3

Metodología

3.1. Configuración experimental.

Para cumplir con los objetivos experimentales propuestos en el Capítulo 1.3, se han realizado distintos experimentos, utilizando la base de datos GSOD.txt. Su tamaño es de 19GB, y recopila datos climáticos diarios a nivel global, incluyendo temperatura, humedad, velocidad del viento y precipitación desde el año 1929. Estos datos se registran en estaciones meteorológicas de todo el mundo y se gestionan por el Centro Nacional de Datos Climáticos (NCDC). Se utilizan para el análisis climático y el desarrollo de modelos y pronósticos meteorológicos a nivel mundial. Esta base de datos puede descargarse directamente desde la página oficial de la organización, o a través de la dirección FTP: ftp://ftp.ncdc.noaa.gov/pub/data/gsod.

Las distintas aplicaciones y *scripts* han sido diseñados utilizando el lenguaje de programación Python. Estas aplicaciones se ejecutarán en el clúster proporcionado por el tutor del trabajo de fin de Grado (*Departamento de Sistemas Informáticos y Computación*). Este clúster se llama Dana, consta de un conjunto de 6 CPUs (nodos) con entorno Linux y 8GB de RAM y 4 núcleos por nodo. Consta de las siguientes versiones de las herramientas utilizadas: Spark (versión 3.3.1) , Hadoop (versión 3.3.4) y Python (versión 3.10.8). Nótese que los resultados obtenidos no son directamente trasladables a clústers con recursos diferentes. Hay que tener en cuenta los límites que se presentan según las características del entorno de trabajo del que dispongamos.

3.2. Métricas del rendimiento y diseño experimental.

Para probar el impacto del proceso *shuffle* en el rendimiento de nuestra aplicación, se diseñaron varios experimentos. En ellos, se tomaron como métricas principales el tiempo de ejecución de las aplicaciones, el número de operaciones I/O y el uso de memoria y CPU.

Estos experimentos se ejecutaron sobre distintas muestras aleatorias del fichero original GSOD.txt, generadas a través del *script buils_sample_hdfs.py*. Cada uno de estos ficheros, *gsod_sample_x_1.txt*, contiene un x% de los registros del fichero original, contando

con un total de 90 ficheros de tamaños incrementales. En nuestro caso, se ha utilizado HDFS o *Hadoop Distributed File System*[6] para almacenar estas muestras.

3.2.1. Experimento 1: Impacto del shuffle en el rendimiento general del sistema.

A diferencia de las transformaciones *narrow*, las transformaciones *wide* activan el proceso *shuffle* y es por esto que el rendimiento de nuestra aplicación se puede ver afectado. Para analizar este impacto, se diseñaron 2 *scripts* distintos, *narrow_transformations.py* y *wide_transformations.py*. Estos *scripts* aplican sobre las distintas muestras del fichero original las transformaciones *narrow*, *map*, *filter* y *union*, y las transformaciones *wide*, *distinct*, *reduceByKey* y *GroupByKey*.

Las transformaciones *narrow* generalmente deberían ser más rápidas y usar menos recursos debido a la ausencia de *shuffle*, mientras que las transformaciones *wide* probablemente mostrarán más lentitud, mayores operaciones de I/O, y mayor uso de memoria y CPU.

Es importante destacar que, aunque las transformaciones *wide* tienden a ser más costosas en términos de rendimiento, son esenciales para muchos tipos de operaciones de procesamiento de datos. El desafío no es evitarlas, sino optimizarlas y entender cómo minimizar su impacto en el rendimiento general de una aplicación Spark.

3.2.2. Experimento 2: GroupByKey vs ReduceByKey.

No todas las distintas transformaciones *narrow* tienen el mismo desempeño en cuanto a *shuffle*. El procesamiento interno de Spark es distinto para la transformación *groupByKey* y *reduceByKey*, por lo que la eficiencia con la que se realiza esta operación puede variar según la transformación utilizada.

Por un lado, cuando se invoca *groupByKey*, Spark lanza un proceso *shuffle* que reúne todos los valores de una misma clave en una sola lista. Esta operación puede ser ineficiente, especialmente cuando hay una gran cantidad de datos para una clave en particular (como en el ejemplo de la Figura 3.1), ya que todos estos datos deben ser transferidos y agrupados en un nodo específico.

[6] *Hadoop Distributed File System* es un sistema de archivos distribuido, que permite el almacenamiento de grandes volúmenes de datos de manera segura y accesible. Es una componente clave del ecosistema *Hadoop*, por lo que puede integrarse con la herramienta Apache Spark.

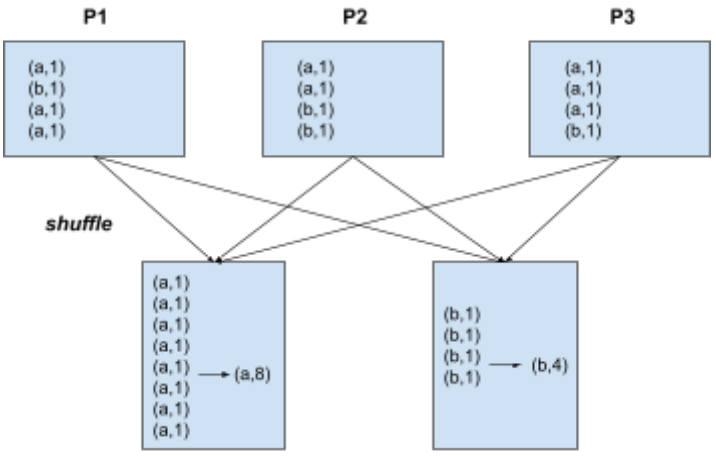

Figura 3.1: Etapa shuffle en la transformación *groupByKey*.

Por otro lado, antes de realizar el *shuffle*, *reduceByKey* combina registros con la misma clave en la misma partición (Figura 3.2). Esta operación de combinación local reduce la cantidad de datos que necesitan ser transferidos durante el *shuffle*. Esto significa que se envían menos datos a través de la red y se necesita menos I/O para escribir y leer datos temporales.

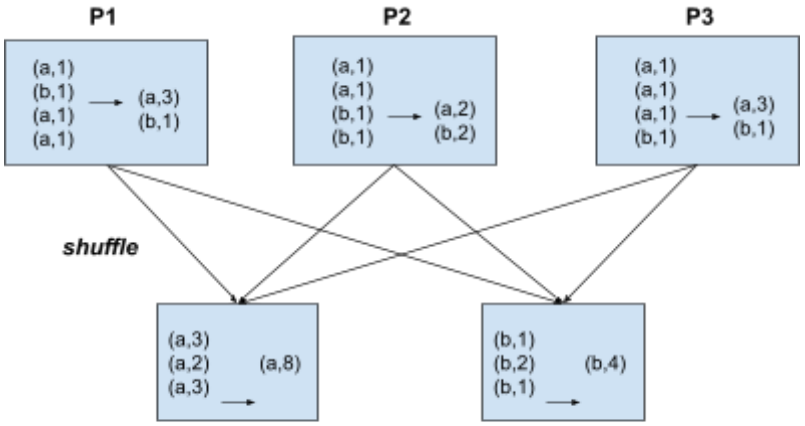

Figura 3.2: Etapa shuffle en la transformación *reduceByKey*.

Por lo tanto, *reduceByKey* generalmente es más eficiente que *groupByKey* cuando se trata de operaciones que requieren *shuffle*. En general, *reduceByKey, aggregateByKey*, *foldByKey* y *combineByKey* aplican un paso de combinación antes de aplicar *shuffle*, por lo que siempre y cuando sea posible, deberían elegirse por encima de *groupByKey*.

Es importante notar que no todos los problemas que pueden resolverse con la transformación *groupByKey* pueden calcularse con la transformación *reduceByKey*. Por ejemplo, *reduceByKey* requiere combinar todos los valores en un nuevo valor del mismo tipo.

En el *script groupByKeyvsreduceByKey.py* se han diseñado las siguientes 2 aplicaciones utilizando *groupByKey* y *reduceByKey* en cada una de ellas:

1. Aplicación 1: cálculo del número de registros de cada estación.
2. Aplicación 2: temperatura mínima histórica por estación.

Así, el experimento debería mostrar una diferencia sustancial en términos de tiempo de ejecución y uso de recursos (menos datos transferidos, menos I/O, y potencialmente menos uso de memoria) entre las dos transformaciones.

3.2.3. Experimento 3: Optimización del número de particiones.

Como ya se ha mencionado en capítulos anteriores, el número de particiones o grado de paralelismo de nuestra aplicación tiene un impacto directo en el rendimiento de esta. Es vital estudiar el número óptimo de particiones con el fin de garantizar el rendimiento eficiente de una aplicación, ya que una cantidad excesiva puede causar sobrecargas significativas en la comunicación entre ejecutores. Esto puede causar que la planificación de tareas se vuelva el cuello de botella del tiempo total de procesamiento y genere desbordamientos de memoria. Por otro lado, un número demasiado pequeño de particiones puede llevar a una utilización subóptima de los recursos del clúster. Por lo tanto, no solo es importante controlar el número de particiones definidas, sino también cómo se distribuyen los datos entre estas.

Como se ha mencionado anteriormente, los ficheros utilizados para los experimentos están almacenados en el *HDFS*. Este sistema fragmenta los archivos en varios bloques o particiones, distribuyéndolos en diferentes nodos del clúster, y sostiene múltiples replicaciones de cada segmento para asegurar una alta disponibilidad y tolerancia a fallos. Es importante destacar que Apache Spark procesa de forma distinta estos archivos. Para los archivos almacenados en *HDFS*, la cantidad de particiones durante una operación *shuffle* se determina por el número de bloques en los que se divide el archivo. En el resto de casos, cuando se desencadena la operación *shuffle*, Spark incrementa automáticamente el número de particiones del RDD a 200, independientemente del tamaño de este.

Sin embargo, existen distintos métodos para alterar el número de particiones. Por una parte, mediante la configuración de Spark, es posible determinar el número de particiones que se generarán tras el proceso *shuffle*, utilizando para ello el parámetro ***spark.default.parallelism***. Este parámetro establece el número predeterminado de particiones que se crearán. Un ajuste óptimo del mismo puede ayudar a reducir el volumen de datos transferidos a lo largo de la red, favoreciendo así una mejor paralelización del proceso y, en consecuencia, una ejecución más eficiente del programa. Además de este parámetro, también se puede controlar el número de particiones mediante el uso de las funciones ***repartition()*** y ***coalesce()***. La primera se puede utilizar para aumentar o disminuir el número de particiones, redistribuyendo los datos de manera más equilibrada entre las particiones y posibilitando una utilización más óptima de los recursos disponibles. Mientras tanto, la segunda solo permite reducir el número de particiones.

Para analizar cómo el grado de paralelismo (dictado por el número de particiones) influye en el rendimiento de nuestra aplicación, se ha desarrollado el *script shuffle_partitions.py.* En este *script*, configuramos una aplicación que emplea la transformación groupByKey, integrando una variabilidad adicional: el parámetro *default.shuffle.parallelism*, que puede adoptar valores en el rango de 1 a 1500, modulando así el número de particiones en cada RDD.

El objetivo principal de este experimento es observar y entender cómo fluctúan los tiempos de ejecución al variar el número de particiones establecidas. Esto nos permitirá evaluar si el número predeterminado de particiones que Spark asigna es el más óptimo en términos de eficiencia de tiempo, o si existiría un valor alternativo que podría ofrecer resultados más favorables. Además, se anticipa que habrá un punto óptimo en el número de particiones donde el rendimiento será máximo y tras sobrepasar este umbral, el rendimiento puede empezar a degradarse a causa del overhead adicional generado por el manejo de un mayor número de particiones.

Capítulo 4

Resultados experimentales y discusión

4.1. Experimento 1: Impacto del shuffle en el rendimiento general del sistema.

Tras ejecutar las distintas transformaciones *narrow* y *wide* sobre los distintos ficheros, se han registrado los resultados en los gráficos que se muestran a continuación (Figura 4.1 y Figura 4.2). El eje de abscisas corresponde con el porcentaje del tamaño del fichero original y, el eje de ordenadas, el tiempo de ejecución de la aplicación. Como puede observarse, existe una clara diferencia entre la distribución de los tiempos de ejecución de las aplicaciones *narrow* (Figura 4.1 y Figura 4.2) y las aplicaciones *wide* (Figura 4.3).

Por una parte, para las transformaciones *map*, *filter* y *union* (Figura 4.1), no se observa ninguna dependencia clara entre el tamaño del fichero y el tiempo de ejecución de la aplicación. Sin tener en cuenta los outliers, si observamos el diagrama de caja y bigotes de la Figura 4.2, los tiempos de ejecución tienen una variabilidad de máximo 2 segundos. En resumen, los tiempos de ejecución se distribuyen uniformemente, con rangos muy similares y poca variabilidad en los tiempos de ejecución. Esto demuestra que el rendimiento de las aplicaciones que solo comprenden transformaciones *narrow* es bastante predecible y estable, siendo estas poco costosas y rápidas.

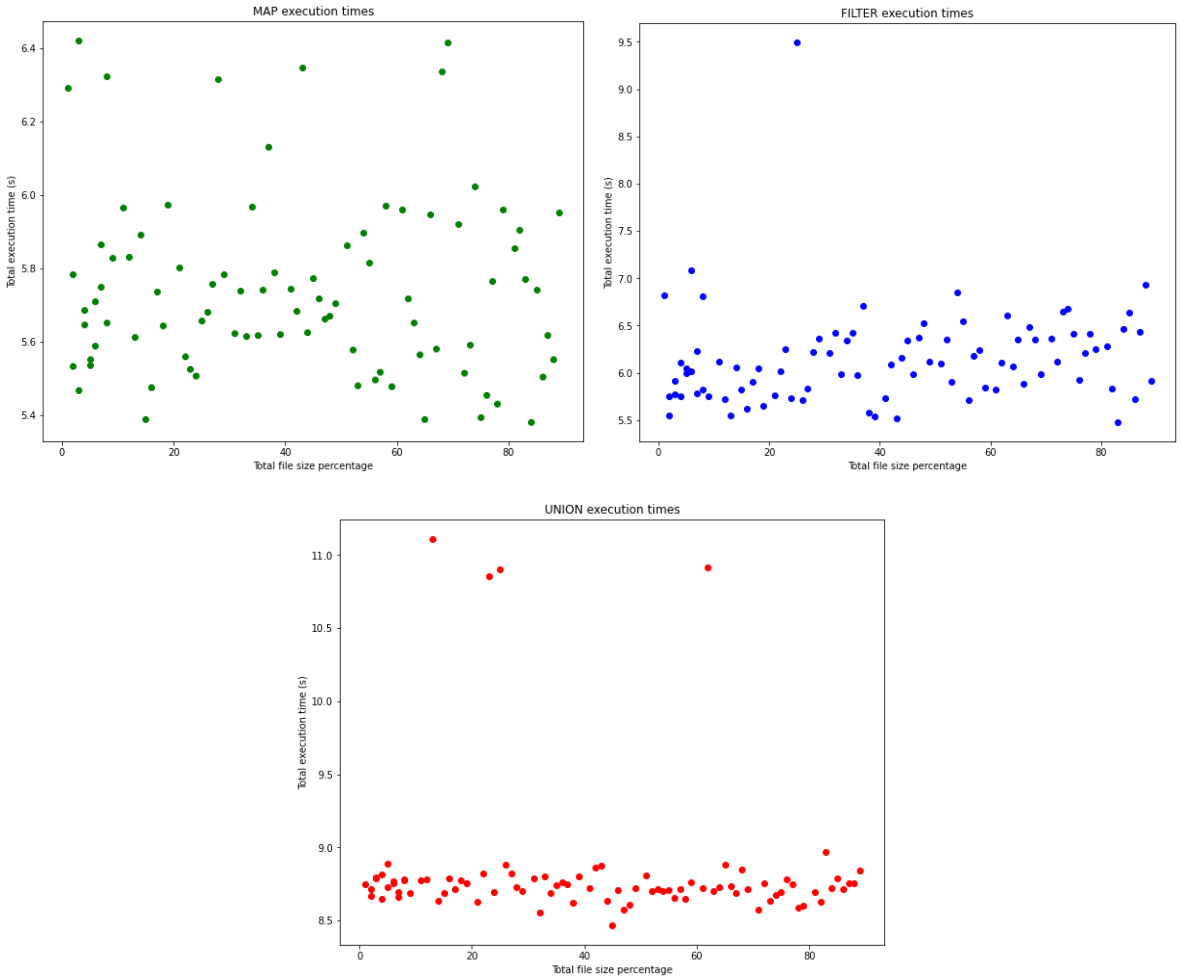

Figura 4.1: Dispersión de los tiempos de ejecución de las transformaciones narrow.

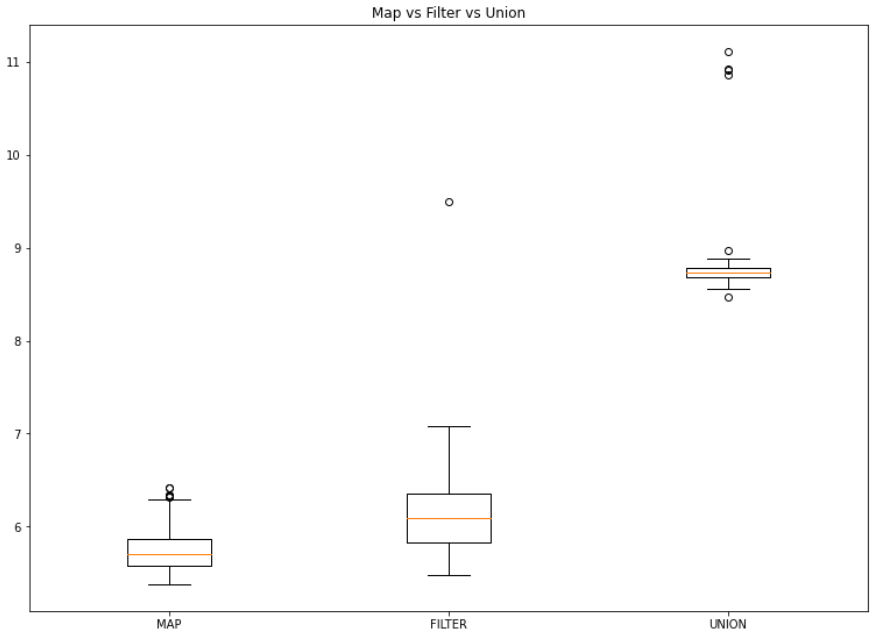

Figura 4.2: Diagrama de caja y bigotes de los tiempos de ejecución de las transformaciones narrow.

Por otra parte, las transformaciones *reduceByKey*, *groupByKey* y *distinct* (Figura 4.3), muestran una distribución completamente distinta de los datos. Puede observarse una clara correlación entre el tamaño del fichero y el tiempo de ejecución. Cabe destacar un sustancial incremento de los tiempos de ejecución para los ficheros de tamaño mayor o igual que el 82% del tamaño del fichero original. Como se verá en el Capítulo 4.3, esto se debe al número de particiones utilizadas en el proceso *shuffle*.

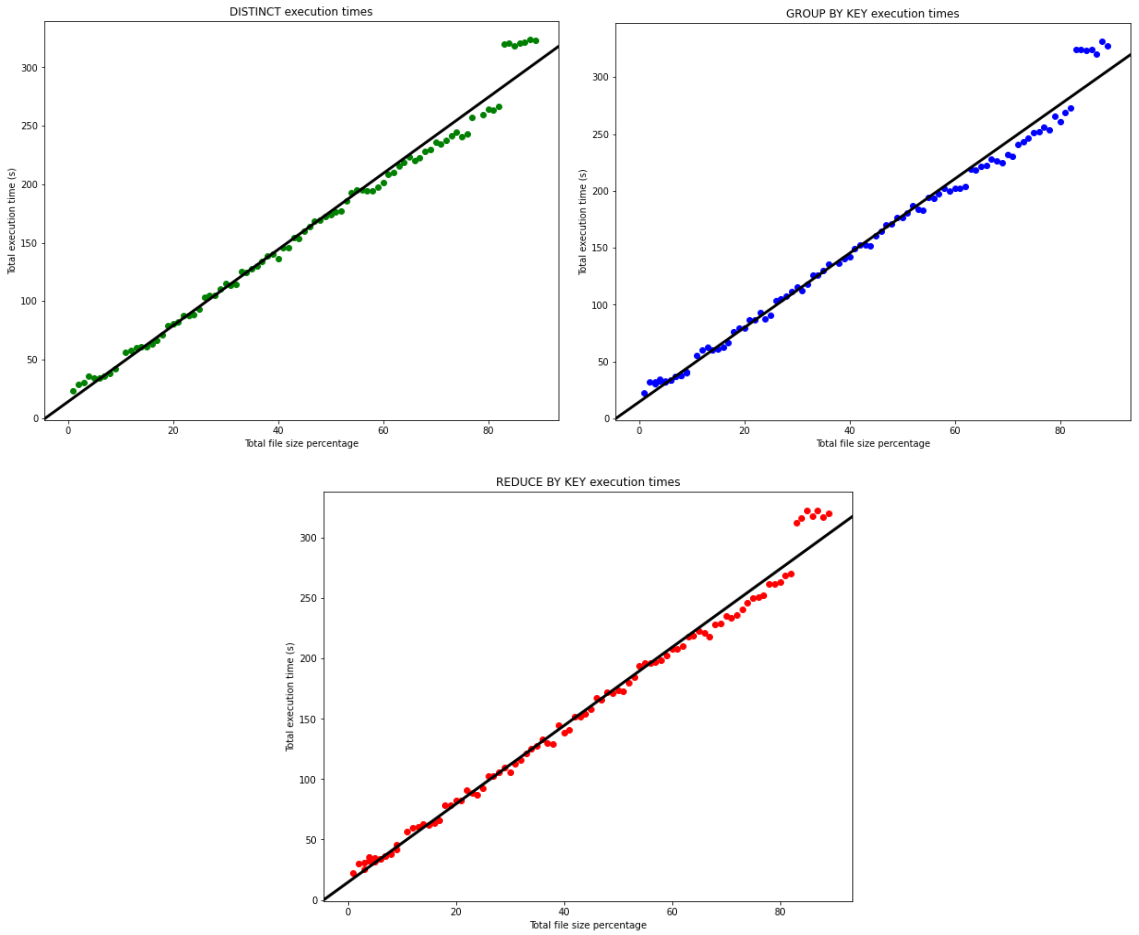

Figura 4.3: Dispersión de los tiempos de ejecución de las transformaciones wide.

La siguiente tabla muestra los modelos de regresión lineal simple a los que se ajustan los pares de valores de cada una de las transformaciones. Como puede observarse, el valor de todos los coeficientes muestra que el tiempo de ejecución y el porcentaje total del tamaño del archivo se ajustan positivamente a una regresión lineal simple.

	distinct	**groupByKey**	**reduceByKey**
modelo	$y = 3.26 * x + 13.78$	$y = 3.26 * x + 14.53$	$y = 3.24 * x + 14.46$
p-valor	$1.5473e - 80$	$5.6715e - 88$	$1.0954e - 94$
pearson	0.9929	0.9932	0.9949
R²	0.98596	0.9866	0.9899

En resumen, los resultados del experimento 1 constatan que el proceso *shuffle* tiene un gran impacto en los tiempos de ejecución de nuestra aplicación y, como se verá en el experimento 4.3, que el número de particiones puede tener un efecto directo en el rendimiento de la tarea.

4.2. Experimento 2: GroupByKey vs ReduceByKey.

Tras ejecutar en el cluster las dos aplicaciones mencionadas en el Capítulo 3.2, utilizando las dos transformaciones *reduceByKey* y *groupByKey*, se ha ajustado, de la misma forma que en el experimento anterior, un modelo de regresión lineal simple para predecir el tiempo de ejecución de la aplicación a partir del tamaño del fichero. Los resultados de este ajuste se resumen en la siguiente tabla:

	Aplicación 1 (sum)		**Aplicación 2 (distinct)**	
	groupByKey	**reduceByKey**	**groupByKey**	**reduceByKey**
modelo	$y = 3.20 * x + 14.71$	$y = 3.07 * x + 12.6$	$y = 2.42 * x + 18.44$	$y = 2.17 * x + 16.11$
p-valor	$3.9799e - 81$	$1.4495e - 79$	$1.8278e - 75$	$4.2103e - 78$
pearson	0.9928	0.9922	0.9902	0.9915
R^2	0.9857	0.9844	0.9806	0.9832

Como puede apreciarse, numéricamente en la tabla y visualmente en la Figura 4.4, existe una diferencia entre las pendientes de las cuatro rectas. En cualquiera de los casos, los tiempos de ejecución de *reduceByKey* están por debajo de los de *groupByKey*. Por tanto, únicamente en términos de tiempos de ejecución el uso de la transformación *reduceByKey* parece más idóneo para ambas aplicaciones.

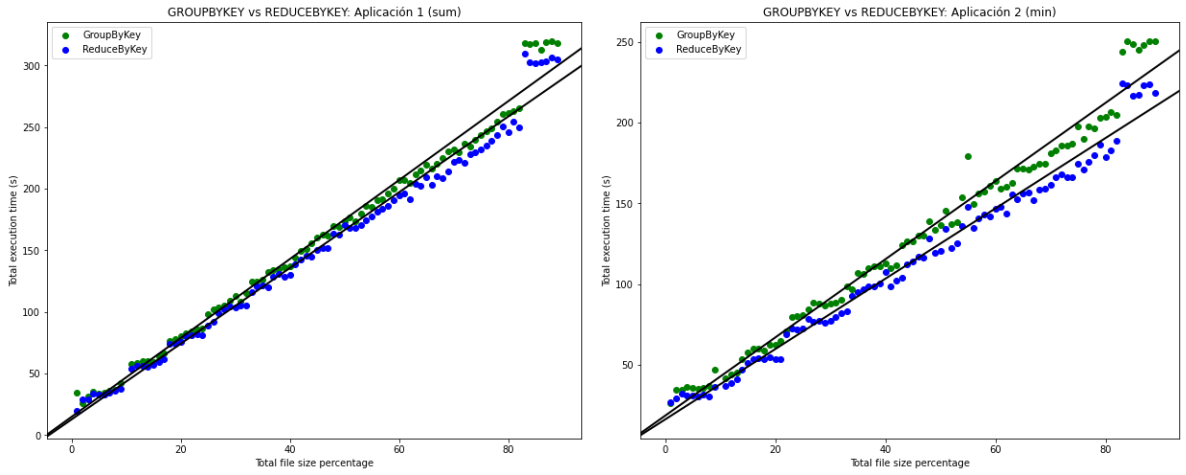

Figura 4.4: Dispersión del tiempo de ejecución para las aplicaciones 1 (izq.) y 2 (dch.) con reduceByKey y groupByKey.

Sin embargo, también buscábamos probar numéricamente la diferencia en el procesamiento de ambas transformaciones. Para ello, se han estudiado con más detalle las métricas asociadas al proceso *shuffle*, disponibles para cada una de las etapas que conllevan *shuffle* de cada aplicación. Para cada aplicación, se han agrupado el número de registros escritos (*Shuffle Write*) y leídos (*Shuffle Read*) durante el proceso *shuffle* bajo las métricas *Total Shuffle Write Records* y *Total Shuffle Read Records*.

En la Figura 4.5, se representa el tamaño del fichero frente al número de registros escritos durante las etapa *shuffle* o *Total Shuffle Write Records*[7]. Puede observarse un comportamiento cuadrático para ambas transformaciones, pero específicamente *reduceByKey*. Al compararla con la transformación *groupByKey*, se observa que escribe menos registros en total por aplicación Esto evidencia la etapa de agregación o reducción aplicada por esta antes de iniciar el proceso *shuffle* donde los datos son distribuídos por clave en las distintas particiones.

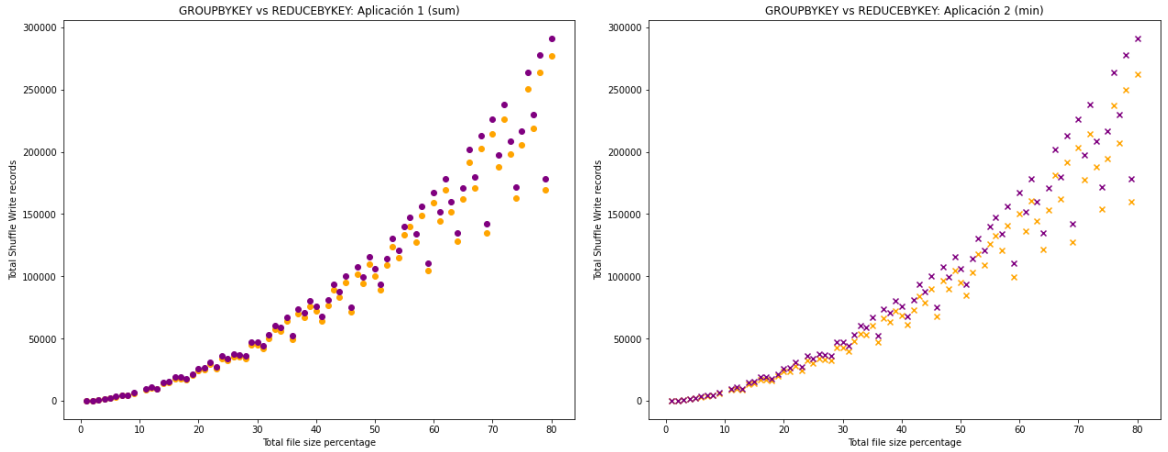

Figura 4.5: Dispersión del número total de registros *shuffle* escritos 1 (izq.) y 2 (dch.) con reduceByKey y groupByKey.

Mientras que los primeros gráficos exponen una diferencia no muy significativa entre los tiempos de ejecución de ambas transformaciones, los segundos prueban una diferencia sustancial en el número de registros escritos durante el proceso *shuffle*. Esto implica una menor transferencia de datos a través de la red. A partir de los dos puntos expuestos, podemos concluir que el rendimiento de *reduceByKey* es mejor que *groupByKey*, y es preferible su uso siempre y cuando sea posible.

[7] Se ha omitido la métrica *Total Shuffle Read Records* ya que siempre coincidirá con la métrica *Total Write Shuffle Records.*

4.3. Experimento 3: Optimización del número de particiones.

En el Capítulo 4.1, identificamos en los resultados algunos valores atípicos en los archivos que superan el 82% del tamaño del fichero original. Estos sobrepasaron la recta de regresión ajustada, con un salto sustancial en el tiempo de ejecución. Se sospechaba que esta característica estaba correlacionada con el número de particiones utilizado.

Para comprender mejor este fenómeno, examinamos el número predeterminado de particiones en las que se distribuyen los datos en la etapa *shuffle* para las transformaciones *wide* definidas en el primer experimento (*distinct*, *reduceByKey* y *groupByKey*). Como ya se explicó, el número de particiones para ficheros almacenados en HDFS viene dado por el número de bloques en los que se almacenan los ficheros. Por tanto, el número de particiones será el mismo para todas estas.

En la Figura 4.6, se muestran el número de particiones y, por tanto, el grado de paralelismo de los distintos ficheros, para las 3 transformaciones: *distinct*, *reduceByKey* y *groupByKey*.

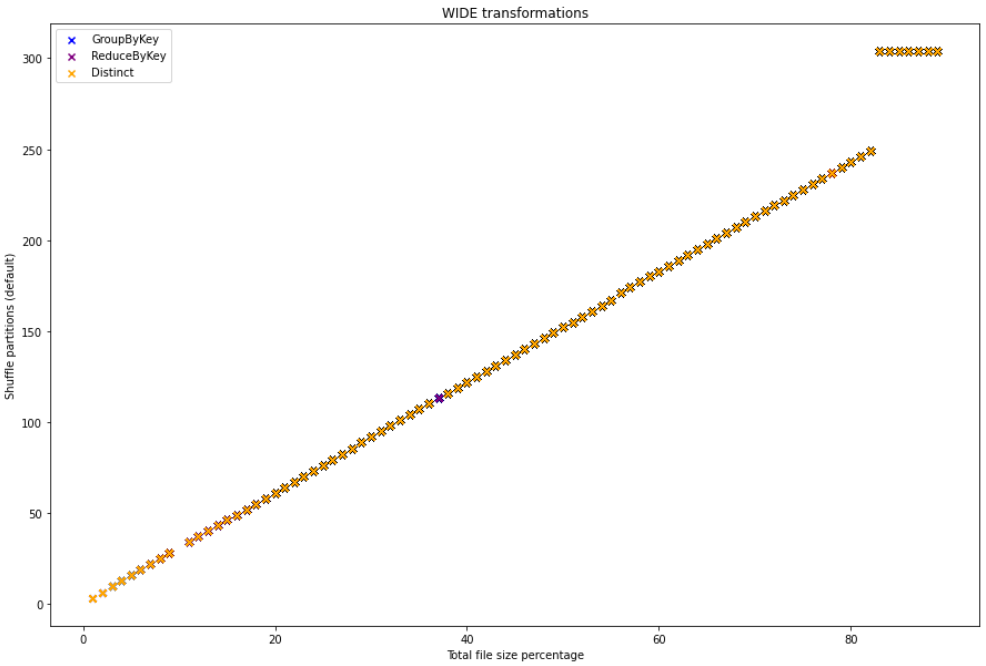

Figura 4.6: Número de particiones para los distintos ficheros en las transformaciones: *distinct*, *reduceByKey* y *groupByKey*.

Durante la evaluación, se evidenció una relación lineal creciente con respecto al número de particiones hasta alcanzar el tamaño del 82%. Pasado este punto, la cantidad de particiones se mantuvo constante (304 particiones por defecto), con un salto evidente en el número de particiones para tamaños mayores o iguales al 82%.

Estos hallazgos motivaron a estudiar si el número predeterminado de particiones es el más idóneo o si, por el contrario, ajustar el número de particiones durante el *shuffle* podría potenciar el rendimiento de nuestra aplicación. Para llevar esto a cabo, se procedió a

ejecutar pruebas con un rango variado de particiones, ajustándose según los tamaños de los archivos en cuestión. La estrategia consistía en comparar los tiempos de ejecución en diferentes conjuntos de archivos con un número variable de particiones frente al tiempo de ejecución resultante del particionamiento predeterminado que establece Spark. A través de este método, aspiramos a distinguir si existe un margen para optimizar el número de particiones y, de ser así, evaluar el impacto que tendría esta optimización en el rendimiento general.

Durante las pruebas preliminares, se observó que un aumento excesivo en el número de particiones resulta en un incremento significativo en el tiempo de ejecución. Este efecto adverso se debe a la sobrecarga asociada con la coordinación entre tareas y ejecutores, un factor que se amplifica con un mayor número de particiones. En la Figura 4.7 se enfrentan el número de particiones y el tiempo de ejecución de la aplicación para los ficheros del 4, 5 y 6 por ciento del tamaño original.

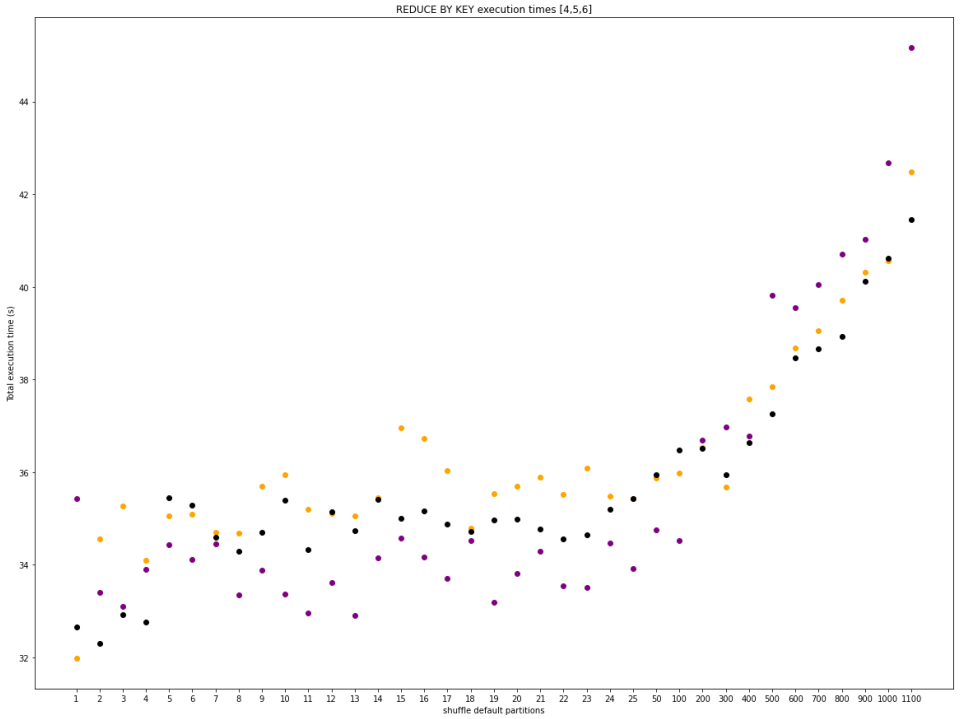

Figura 4.7: Tiempos de ejecución en función del número de particiones para los ficheros de tamaño 4%, 5% y 6%.

Se advierte un patrón creciente a partir de 25 particiones. Sin embargo, para un número de particiones inferior, no se induce ningún patrón claro. Es por esto que se decidió ejecutar repetidas veces el mismo experimento, para contar con la variabilidad del sistema en los resultados. En la Figura 4.8, se estudian ahora los ficheros del 9, 11 y 12 por ciento del tamaño original.

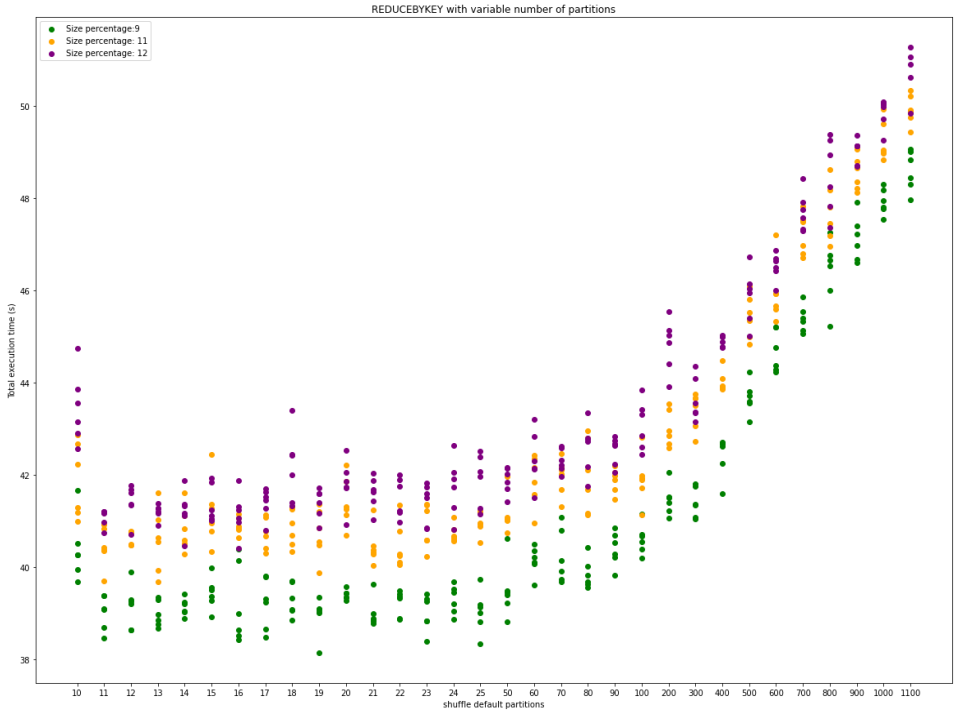

Figura 4.8: Tiempos de ejecución en función del número de particiones para los ficheros de tamaño 9%, 11% y 12%.

Se observó que para los distintos ficheros se puede elegir un número particular de particiones, distinto al definido por defecto por Spark, que ofrece mejores resultados. Las diferencias y mejoras en el rendimiento eran más notables para los ficheros de tamaño mayor o igual que 81. Por ejemplo, para la el fichero del 82%, el tiempo de ejecución pasa de 272 segundos de media a 205 segundos, variando el número de particiones de 249 a 300. Los resultados de experimentos sobre estos se resumen en la Figura 4.9.

	Transformation	Size percentage	Default Spark Partitions	Default Spark execution time (s)	Set up partitions	Variable partitions execution time (s)	Improvement (s)	Improvement %
0	groupByKey	81	246	268.835	300	203.100500	65.734500	32.365504
1	groupByKey	82	249	272.919	300	205.225167	67.693833	32.985152
2	groupByKey	83	304	323.846	600	240.248333	83.597667	34.796357
3	groupByKey	84	304	324.100	400	239.124000	84.976000	35.536374
4	groupByKey	85	304	323.085	400	240.094000	82.991000	34.566045
5	groupByKey	86	304	324.328	400	239.197000	85.131000	35.590329
6	groupByKey	87	304	320.452	600	240.158667	80.293333	33.433452
7	groupByKey	88	304	331.692	400	245.315833	86.376167	35.210188

Figura 4.9: . Tiempos de ejecución para distintos ficheros y número de particiones.

En general, para los ficheros de mayor tamaño, existe una mejora de alrededor del 33% (columna ‘Improvement %’) en términos de tiempo de ejecución al utilizar ‘Set up partitions’ como número de particiones en vez del definido por Spark.

En conclusión existe un gran margen de mejora en cuanto al número de particiones shuffle predefinido por Spark. Es importante notar que el valor óptimo de particiones a definir no solo dependerá del tamaño de los ficheros o del tipo de datos a manejar, sino también de los recursos de nuestro clúster.

Capítulo 5

Conclusiones y recomendaciones

5.1. Resumen de los hallazgos.

A lo largo del trabajo de fin de grado, se diseñaron y ejecutaron tres experimentos críticos para entender el comportamiento del proceso *shuffle* en aplicaciones Spark. Uno de los hallazgos más significativos fue la notable diferencia en los tiempos de ejecución entre las aplicaciones que instigaron el proceso de *shuffle* y las que no. Esto subraya la considerable demanda de recursos que el *shuffle* puede imponer a una aplicación.

Otro descubrimiento crucial fue la eficiencia que se puede lograr utilizando transformaciones *wide* que incorporan una etapa de agregación antes del procedimiento de *shuffle*. Esta estrategia parece minimizar la cantidad de datos que necesitan ser transferidos, optimizando así el proceso.

Adicionalmente, se identificó que el número de particiones predeterminado por Spark no siempre garantiza el rendimiento óptimo. Una exploración cuidadosa de diferentes configuraciones de partición reveló que ajustar el número de estas puede llevar a mejoras significativas en la eficiencia de las aplicaciones Spark.

5.2. Recomendaciones para mejorar el mecanismo de shuffle.

Basándonos en los hallazgos derivados de los experimentos realizados, es evidente que hay margen para mejorar el rendimiento del mecanismo de *shuffle* en Spark. Una de las recomendaciones más preeminentes sería fomentar el uso de transformaciones *wide* que ejecutan una etapa de agregación antes de entrar en la fase de *shuffle*, reduciendo así la carga de datos y, por ende, optimizando los tiempos de ejecución.

Asimismo, es aconsejable no conformarse con la configuración predeterminada de Spark con respecto al número de particiones. Los desarrolladores deberían estar dispuestos a experimentar con diferentes números de particiones para encontrar el equilibrio perfecto que maximice la eficiencia de su aplicación.

En conclusión, el trabajo realizado demuestra que una comprensión profunda del proceso *shuffle* y una elección cuidadosa de las estrategias y configuraciones pueden llevar a mejoras sustanciales en el rendimiento de las aplicaciones Spark. Se insta a los futuros desarrolladores a que consideren estos hallazgos y recomendaciones para potenciar la eficiencia de sus aplicaciones Spark.

Bibliografía

[1] Apache Spark official documentation: https://spark.apache.org/docs
[2] Matei Zaharia, Mosharaf Chowhury, … , *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, University of California, Berkeley, 2012.
[3] Bill Chambers & Matei Zaharia, *Spark: the Definite Guide, Big Data Processing Made Simple*, O'Reilly, 2018.
[4] Nirali Rana, *Shuffle Performance in Apache Spark*, International Journal of Engineering Research & Technology (IJERT), 2015.
[5] Ajay Gupta & Naushad Ahamad, *Guide to Spark Partitioning: Spark Partitioning Explained in Depth*, 2020.
[6] Haoyu Zhang, Brian Cho, …, *Riffle: Optimized Shuffle Service for Large-Scale Data Analytics*, Princeton University (Facebook Inc), 2018.
[7] Brunenberg, P. 2021. *Understanding Apache Spark Shuffle*, blog.kholdus.com.
[8] Aaron Davidson and Andre Or, *Optimizing Shuffle Performance in Spark*, UC Berkeley, 2015.
[9] Gupta, A. 2020. *Revealing Apache Spark Shuffling Magic*, medium.com.
[10] Nazir, M.2023. *Know Apache Spark Shuffle Service*, ksolves.com
[11] de Beneducci, M. 2018. *Apache Spark - Performance*, blog.scotlogic.com.
[12] Viriyavaree, S. 2022. *Shuffle Partition Size Matters and How AQE Help Us Finding Reasoning Partition Size*, medium.com.
[13] Mehanna, N. 2020. *Efficiently working with Spark partitions*, naifmehanna.com.
[14] IBM, 2023. *Tuning Spark application tasks*, ibm.com/docs.
[15] Asif Abbasi, M, *Learning Apache Spark 2*, Packt Publishing Ltd, 2017.