ESTUDIO COMPARATIVO ENTRE APACHE FLINK Y APACHE SPARK

Medición de la performance en la ejecución de algoritmos tradicionales de un Datawarehouse

Trabajo Final presentado para obtener el grado de Especialista en Inteligencia de Datos orientada a BIG DATA

Febrero 2021

Autor

Ing. Rubén Alejandro Jaime

Director

Dr. Lic. Waldo Hasperué

Facultad de Informática - Universidad Nacional De La Plata

RESUMEN

El presente trabajo tiene como objetivo desarrollar un estudio comparativo entre Apache Flink y Apache Spark, partiendo de la medición de la performance en la ejecución de algoritmos tradicionales de un Datawarehouse. Para ello, la presente investigación se sustenta en el paradigma cuantitativo de tipo comparativo. Las técnicas de investigación son el análisis de documento y análisis de contenido. Los resultados develan que de tratarse de un proyecto que requiere de amplio volumen de procesamiento de datos, la opción es emplear Apache Spark, dada la facilidad para codificar y realizar numerosas tareas; adicionalmente se pudo develar que este framework dispone de abundante información y profesionales con competencias y experiencia para trabajar en este sistema. Por su parte Apache Flink se distingue por ser un framework diseñado para procesamiento de streaming, no cuenta con mucha información de fácil acceso y existe un número reducido de especialistas con dominio y experiencia en este campo; siendo dos desventajas para su selección. Además, es importante señalar que ambos framework son eficientes en la ejecución de tareas, aunque en cuanto a versatilidad se distingue Apache Spark, por cuanto que permite emplear su potencial en diversos lenguajes de programación. Asimismo, es de destacar que a través de la experimentación los mejores tiempos obtenidos para ambas herramientas se logran al modificar el storage y cuando se lo trabaja en un formato columnar.

Palabras claves: Apache Flink, Apache Spark, Big Data, KPI, Datawarehouse.

ABSTRACT

The present study aims to develop a comparative study between Apache Flink and Apache Spark, starting from the measurement of Performance in the execution of traditional algorithms of a Datawarehouse. For this, the research is based on the Comparative Quantitative Paradigm. The data collection techniques are document analysis and content analysis, designing as an instrument a data matrix in which the advantage and disadvantage of each Datawarehouse referred to during the execution of tests and the document review developed is indicated. The results reveal that being a project that requires a large volume of data processing, the option is to use Apache Spark, given the ease of coding and carrying out numerous tasks; additionally, it was revealed that this framework has abundant information and professionals with skills and experience to work on this system. For its part, Apache Flink is distinguished by being a framework designed for streaming processing, it does not have much information that is easily accessible and there is a small number of specialists with expertise and experience in this field; being two disadvantages for its selection. In addition, it is important to point out that both frameworks are fast, although Apache Spark is distinguished in terms of versatility, in that it allows to use its potential in various programming languages. Likewise, it is noteworthy that through experimentation the best times obtained for both tools are achieved by modifying the storage and it is converted into a columnar format.

Keywords: Apache Flink, Apache Spark, Big Data, KPI, Datawarehouse.

Tabla de Contenido.

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

GLOSARIO DE TÉRMINOS

+: Suma aritmética o lógica (OR) dependiendo del contexto.

A o A^{J}: Operador NOT o negación.

A, B, C, D: Variables lógicas.

API: Aplication Programming Interface.

AVRO: Marco de serialización de datos y llamadas de procedimiento remoto orientado a filas desarrollado dentro del proyecto Hadoop de Apache.

BSP: Bulk Synchronous Parallel.

Catalyst: Catalizador.

CEP: Procesamiento de Eventos Complejos.

DStreams: Flujo Discretizado.

ETL: Extrac-Transform-Load.

f, g, h: Funciones Lógicas

HDFS: Hadoop Distributed File System.

Hits: Hypertext Induced Topic Selection.

IP: Internet Protocol.

IS: Información Source Weigh Ted.

LDA: Latent Dirichlet Allocation.

RDD: Resilient Distributed Dataset.

RM-ODP: Reference Model for Open Distributed Processing.

RU: Agregación Rotation Unit

RV: Agregación Rotation Vault

SALSA: Stochastic Approach for Link-Estructure Analysis.

Spark MLlib: Librería de algoritmos de aprendizaje de máquina.

SparkContext: Contexto básico de Spark

SparKLP: Spark Link Predictor.

SR: Sistema de Recomendación.

VFS: Virtual File System.

PREFACIO

Una frase bastante común en el ámbito del Big Data, adjudicada al matemático y emprendedor británico Clive Humby, es que los datos son el nuevo petróleo; y el grupo de tecnologías referenciadas como Big Data definitivamente representan la mejor estrategia para explotarlos. Lejos de ser una solución mágica como suele considerarse, por el contrario, si bien el Big Data es una solución que permite lidiar con varias limitaciones que se tenían con tecnologías anteriores como ser Datawarehouse, debe tenerse en cuenta que es necesario contar con personal capacitado, calidad de datos y una organización madura. El conocimiento sobre los frameworks de Big Data a la hora de su elección y posterior diseño son etapas cruciales para poder cumplir con las exigencias del mercado, maximizar su desempeño y obtener un rendimiento óptimo.

A partir de aquí deriva el foco de la presente investigación, cuya principal interrogante refiere a la elección de frameworks, específicamente para la tarea de ejecución de algoritmos tradicionales de un Datawarehouse. En este sentido, la inquietud que orienta el presente estudio es ¿Cuáles son las diferencias entre Apache Flink y Apache Spark, partiendo de la medición de la Performance en la ejecución de algoritmos tradicionales de un Datawarehouse?

Para esto es necesario introducir a los dos frameworks que han ganado más relevancia y que utilizan procesamiento de datos en memoria para lograr mejores tiempos en comparación del framework pionero MapReduce en el tratamiento masivo de datos, actualmente en desuso por cuestiones de eficiencia.

En este sentido, uno de los frameworks presentados como se ha enunciado es Apache Flink, el cual comienza su historia en 2010 como un proyecto de investigación de universidades alemanas, y llegando a ser aceptado como un proyecto top-level en la Fundación de Software Apache en diciembre de 2014. Apache Flink está desarrollado en Java, realiza procesamiento de flujo de datos utilizando un modelo de ventanas de tiempo y puntos de control, procesa de manera indiferente un flujo finito e infinito y además permite el procesamiento batch como un caso particular del procesamiento de flujo.

Su contraparte es Apache Spark, creado por Matei Zaharia en la Universidad de Berkeley e implementado en Scala. Este framework ha sido donado en 2013 a la Fundación de Software Apache, logrando su estado de proyecto top-level en enero de 2014. Apache Spark procesa datos casi en tiempo real, y está basado en un modelo de micro batching. Asimismo, soporta procesamiento de flujos de datos (utilizando una estructura denominada DStreams) y procesamiento por lotes con sus estructuras de datos denominadas RDD (Resilient distributed dataset).

De esta manera se evidencian las diferencias conceptuales existentes entre ambos frameworks, cuyas características determinan sus comportamientos frente a diferentes escenarios y a su antecesor en la resolución de algoritmos tradicionales el Datawarehouse.

CAPÍTULO I. INTRODUCCIÓN

Objetivos de Investigación.

Objetivo General.

Desarrollar un estudio comparativo entre Apache Flink y Apache Spark, partiendo de la medición de la Performance en la ejecución de algoritmos tradicionales de un Datawarehouse.

Objetivos Específicos.

a) Indagar desde la perspectiva teórica la caracterización de Apache Flink y Apache Spark como frameworks de procesamiento distribuido.
b) Establecer criterios para la valoración de las ventajas comparativas de Apache Flink y Apache Spark, partiendo de la medición de la performance en la ejecución de algoritmos tradicionales de un Datawarehouse.
c) Ejecutar pruebas en base a criterios de comparación para la valoración de Apache Flink y Apache Spark, partiendo de la medición de la performance en la ejecución de algoritmos tradicionales de un Datawarehouse.

Metodología

A fin de alcanzar los objetivos enunciados, metodológicamente la investigación se orienta bajo el paradigma cuantitativo, a través de un tipo de estudio comparativo, por cuanto que de acuerdo con Hurtado (2015), permite "…destacar la forma diferencial en la cual un fenómeno se manifiesta en ciertos contextos o grupos diferentes, sin establecer relaciones de causalidad." (p. 116). En este sentido, según Hurtado (2015), para establecer el proceso de comparación es indispensable desarrollar previamente un análisis y descripción del fenómeno en estudio. De igual forma, se precisa acotar también que las técnicas de recolección de datos son el análisis de documentos y análisis de contenido, diseñando para ello como instrumento una matriz de datos en la

que se indica la ventaja y desventaja de cada Datawarehouse referidos durante la ejecución de pruebas y la revisión documental desarrollada.

Contribución.

En este trabajo se realiza un estudio comparativo entre los frameworks Apache Flink y Apache Spark, que durante el último lustro han ganado popularidad por encima de otros. El análisis comparativo tiene por objeto valorar las diferencias de estos frameworks, cómo funcionan y sobre qué tecnologías se apoyan para realizar un análisis que conduzca a la elección del que resulte más adecuado para un problema determinado.

La cantidad de posibilidades en Big Data a la hora de elegir una solución es tan amplia, que cometer un error en su elección implica indefectiblemente un aumento de tiempo y de costos; de esta manera, un proyecto que depende de una tecnología o framework inadecuado, debe enfrentarse a todo tipo de problemas a futuro (mantenimiento y tiempos de procesamiento no satisfactorios, entre otros).

La ejecución de consultas tradicionales en entornos Datawarehouse, como, por ejemplo: el cálculo de KPI de rendimiento de un producto por sucursal puede consumir muchos recursos, tanto en tiempo de ejecución como de uso de almacenamiento, a diferencia de los frameworks para Big Data que logran realizar consultas de forma rápida y efectiva.

En Datawarehouse, la mayor parte de su procesamiento es por lotes, los cuales demoran mucho tiempo en su ejecución ya que por lo general no contemplan todos los cálculos requeridos, debiendo realizar agregaciones complejas. Este escenario conlleva a largas discusiones en el ámbito corporativo sobre qué herramientas y tecnologías deben ser implementadas para satisfacer al negocio en términos de *time to market* y toma de decisión.

En función a lo expuesto, el presente estudio presenta como aporte principal el desarrollo de un estudio comparativo entre Apache Flink y Apache Spark, partiendo de la medición de la Performance en la ejecución de algoritmos tradicionales de un

Datawarehouse; a fin de que se pueda ofrecer una mejor comprensión de la funcionalidad de cada uno de estos frameworks en el manejo de grandes datas.

Marco Referencial

El Marco Referencial según explica Hurtado (2015) comprende una revisión de los trabajos previos realizados sobre el problema en estudio y de la realidad contextual en la que se ubica.

El marco teórico es una de las fases más importantes de un trabajo de investigación, consiste en desarrollar la teoría que va a fundamentar el proyecto. Una vez que se ha seleccionado el tema objeto de estudio y se han formulado las preguntas que guíen la investigación, el siguiente paso consiste en realizar una revisión de la literatura sobre el tema. Esto consiste en buscar las fuentes documentales que permitan detectar, extraer y recopilar la información de interés para construir el marco teórico pertinente al problema de investigación planteado.

1. Big Data

Según Hernández et al., (2017), se denomina Big Data al conjunto de tecnologías de hardware y software que permiten el tratamiento de grandes volúmenes de datos. En este sentido, es preciso considerar lo siguiente:

> A pesar de que el término Big Data se asocia principalmente con cantidades de datos exorbitantes, se debe dejar de lado esta percepción, pues Big Data no va dirigido solo a gran tamaño, sino que abarca tanto volumen como variedad de datos y velocidad de acceso y procesamiento. En la actualidad se ha pasado de la transacción a la interacción, con el propósito de obtener el mejor provecho de la información que se genera minuto a minuto. (Hernández et al., 2017, p. 3).

Por lo general estos datos están almacenados en diferentes computadoras (formando lo que se conoce como clúster) las cuales tienen instalados diferentes aplicaciones y frameworks que permiten a los analistas de datos consultar ese enorme volumen de datos distribuidos de manera ágil y transparente como si todo este cuerpo de datos analizados estuviera almacenado en una única computadora.

En las siguientes secciones se estudian algunos conceptos sobre las tecnologías que están involucradas en un ambiente de Big Data, desde cómo se almacena la información, hasta cómo se recupera y qué lenguajes pueden ser utilizados para esta tarea.

1.1 Formato y Tipos de Archivo Utilizados en Big Data

En ambientes de Big Data, en particular en aquellos donde se utilizan Apache Flink y Apache Spark, existen diferentes formatos de archivos distribuidos. Cada uno de ellos tiene sus particularidades etiquetando según su estructura y clasificándose según su tipo de formato; existiendo tres: estructurados, semi estructurados y sin estructura. (Villazón et al., 2019).

Los archivos con formato estructurado son aquellos que contienen un esquema que los describe, por ejemplo, se pueden nombrar los datos que residen en las bases de datos relacionales. Algunos de los formatos más utilizados son los conocidos como Parquet y ORC. (Hernández et al., 2017).

Los archivos con formato semiestructurado son de esquema variable: archivos de texto, CSV (archivos separados por comas), JSON (JavaScript Object Notation). Mientras que los archivos sin estructura son archivos con datos que no tienen un esquema: imágenes, videos, texto libre o archivos binarios. (Hernández et al., 2017).

Habitualmente es más simple trabajar con datos estructurados ya que hay muchas herramientas para hacerlo, por ejemplo, el lenguaje de consulta estructurado SQL. No es así para los semiestructurados y los no estructurados, que se han transformado en un gran desafío al momento de extraer información a partir de ellos. (Monleón, 2015). Con el constante crecimiento de las necesidades de extraer información de estos formatos nacen las bases de datos NoSQL, las cuales constituyen un conjunto de bases de datos no relacionales creadas para cubrir un conjunto finito de casos de uso que no cubren el ACID.

Por otro lado, existen formatos de archivos que mantienen los datos en columnas contiguas tanto en el disco como en la memoria, los denominados formatos columnares. De esta manera, es posible acceder a todos los valores de una columna de manera óptima, minimizando la lectura en disco.

Al igual que con los archivos de formato columnar existen las bases de datos columnares, las cuales almacenan los datos de una sola columna de forma contigua en el disco, en la memoria o en ambos. De esta manera puede procesarse rápidamente todos los valores de una columna cargándolos en memoria con pocos accesos al disco. Este tipo de formato es ideal para las tareas de agregaciones. (Hernández et al., 2017). Dado que cada columna se almacena de forma contigua como una unidad separada, se puede utilizar la técnica de compresión que sea más adecuada para la columna en particular: dependiendo de la cardinalidad de la columna y el tipo de datos que almacene.

El formato de almacenamiento en columnas permite una mejor utilización de la memoria caché de la CPU, ya que la caché se llena de valores relacionados (misma columna), los cuales son necesarios para que el ejecutor de la consulta ejecute las operaciones pertinentes (evaluación, cálculo, etc.). (Villazón et al., 2019). En otras palabras, solo se introducen en la caché los datos que realmente necesitan ser examinados. Por lo tanto, el procesamiento de valores de una misma columna es mucho más rápido que las técnicas de procesamiento convencionales.

Cualquier consulta eventualmente necesita enviar el conjunto de resultados (tuplas) al usuario final. Esto debe ocurrir independientemente del tipo de almacén (de columnas o de filas) de datos que se esté utilizando. Otra ventaja de los formatos de datos columnares es que solo se accede al subconjunto de columnas necesarias para resolver una consulta (por lo general pocas en las consultas analíticas). (Villazón et al., 2019).

En las siguientes secciones se desarrollan tres formatos de archivo columnar que fueron estudiados durante esta investigación.

1.1.1 Parquet

Apache Parquet es un formato de tipo columnar de código abierto que está optimizado para trabajar con datos complejos y estructurados dando buen tiempo de respuesta. Incluye mecanismos para optimizar el espacio de disco usando compresión y codificación de tipos. Este formato, al ser orientado a columnas, usa menos almacenamiento y tiene mayor rendimiento en las consultas. (Inoubli et al., 2018). A

diferencia de otros formatos columnares que necesitan aplanar las estructuras complejas para poder guardarlas, Parquet las guarda sin aplanar.

En este sentido, vale destacar que el formato Parquet está constituido por una cabecera formada por cuatro bytes y un identificador “PAR” que indica el formato del archivo parquet; un grupo de filas que consisten en una división horizontal lógica de los datos en filas, un grupo de filas consiste en un bloque de columna para cada conjunto de datos; una sección de columnas, la cual contempla un bloque de datos para una columna particular; las páginas en la que los bloques de columnas se dividen; y por último el pie de página que contiene los metadatos que incluye formato del archivo, esquema, par de clave / valor y la localización de todas las columnas y ubicaciones de inicio de los metadatos de la columna.

En la figura 1 se muestra la estructura descrita, en la que se devela un ejemplo donde hay *N* columnas divididas en *M* grupos de fila. En la figura 2 se puede evidenciar que los metadatos contienen las ubicaciones de comienzo de las columnas. Estos metadatos se graban en archivos *Thrift*[1] y se escriben después de los datos para permitir la escritura en una sola pasada. Se espera que quien los lea, revise primero los metadatos para encontrar todos los bloques de las columnas de interés. Estos bloques son leídos de manera secuencial.

[1] https://thrift.apache.org/

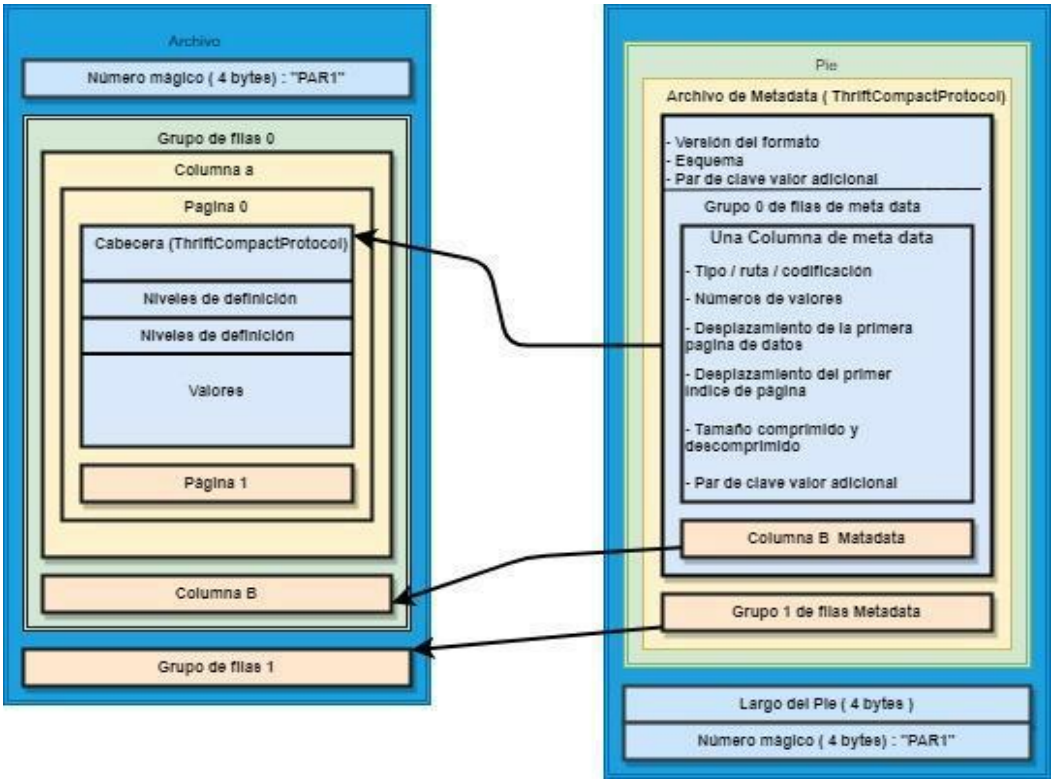

Figura 1. Formato de los archivos Parquet. **Fuente:** Tomado de parquet.apache.org.

```
4-byte magic number "PAR1"
<Column 1 Chunk 1 + Column Metadata>
<Column 2 Chunk 1 + Column Metadata>
...
<Column N Chunk 1 + Column Metadata>
<Column 1 Chunk 2 + Column Metadata>
<Column 2 Chunk 2 + Column Metadata>
...
<Column N Chunk 2 + Column Metadata>
...
<Column 1 Chunk M + Column Metadata>
<Column 2 Chunk M + Column Metadata>
...
<Column N Chunk M + Column Metadata>
File Metadata
4-byte length in bytes of file metadata
4-byte magic number "PAR1"
```

Figura 2. Ejemplo de estructura del formato Parquet. **Fuente:** Tomado de parquet.apache.org.

1.1.2 ORC

Apache ORC es un formato de archivo que fue diseñado para cargar trabajos en Hadoop[2] y almacenar colecciones de filas en un único archivo. Las filas se graban en formato columnar, lo que permite que sean procesadas de forma paralela a través del clúster. Cada archivo se comprime, optimizando espacio. En la figura 3 se puede observar la estructura del formato ORC. (https://blog.cloudera.com/orcfile-in-hdp-2-better-compression-better-performance/).

La metadata de ORC es grabada usando el protocolo de *protocol buffers* que proporciona la capacidad de agregar nuevos campos sin generar problemas de consistencia para los procesos lectores. La lectura del archivo ORC se lee desde atrás: en lugar de hacer muchas lecturas cortas, se leen los primeros 16k bytes del archivo tratando de leer las secciones de pie de página. El último byte del archivo contiene la longitud serializada del PostScript que debe ser menor a 256 bytes.

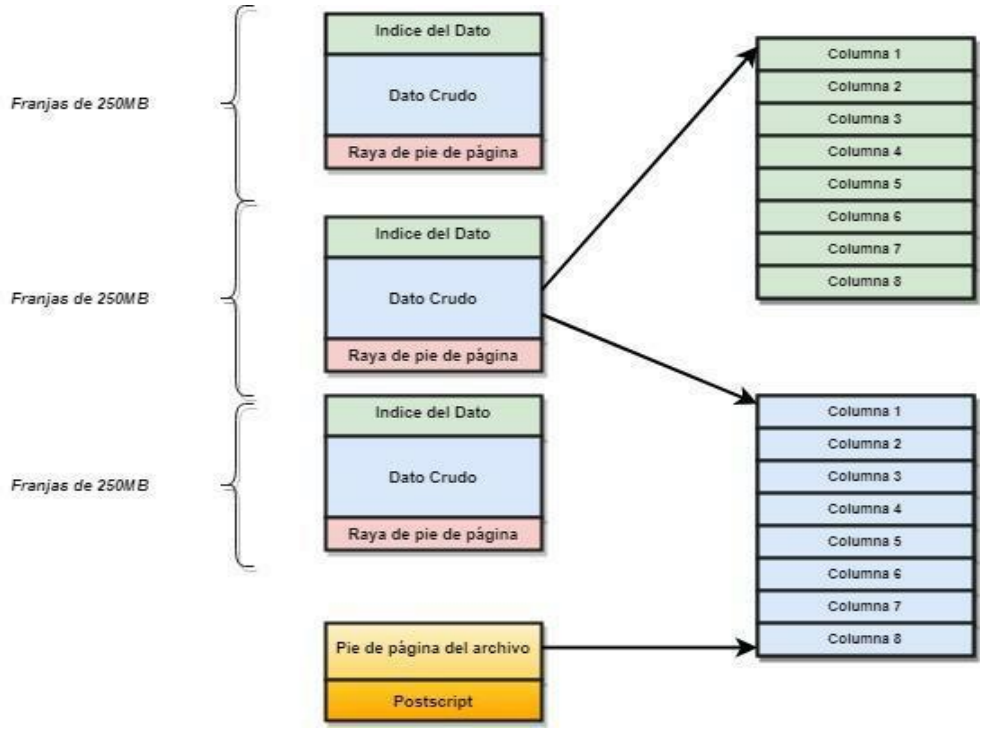

Figura 3. Estructura de ORC. **Fuente:** Tomado de parquet.apache.org.

[2] https://hadoop.apache.org/

Una vez conocida la longitud comprimida, el pie de página se puede descomprimir y analizar. (Veija et al., 2016). La estructura del formato tiene las siguientes divisiones: PostScript, la cual proporciona la información necesaria para interpretar el resto del archivo que incluye la longitud de las secciones de pie de página y metadato del archivo, versión del mismo, tipo de compresión (por ejemplo, zlib, snappy). PostScript no se comprime y termina con un byte antes del final del archivo.

Asimismo, cuenta con un pie de página el cual contiene la estructura del cuerpo del archivo en función al tipo de esquema, número de filas y estadísticas sobre cada una de las columnas. (Veija et al., 2016). Además, posee información de la franja, dividiéndose en tres secciones a saber: conjunto de índices para las filas dentro de la franja, los datos y el pie de página de la franja.

Todas las filas de ORC deben tener el mismo esquema. El esquema se presenta como un árbol, donde los tipos compuestos tienen subcolumnas (Figura 4).

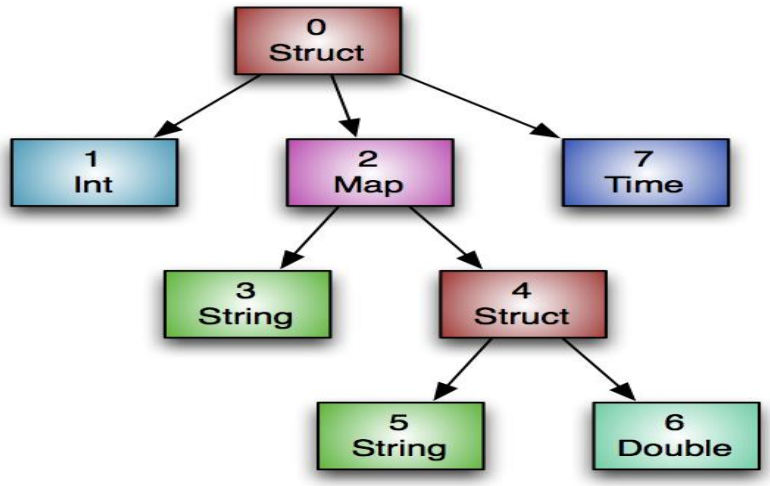

Figura 4: Árbol ORC. **Fuente:** Tomado de Apache Hive ORC.

1.1.3 AVRO

AVRO es un formato JSON para definir tipos de datos y protocolos. Además, serializa y compacta datos en formato binario. Este formato se basa en esquemas: al leerse un archivo, se usa el esquema para obtener el dato y se graba el mismo permitiendo que los datos puedan ser procesados más tarde por cualquier programa que soporte librerías JSON, ya que los esquemas están definidos en esa

especificación. Este formato soporta deserialización parcial, deserializa sólo la columna que va a procesar y soporta compresión *snappy*[3].

Dentro de un clúster, los archivos JSON y XML no pueden dividirse, lo que genera una gran limitación para ser procesados de forma paralela. AVRO permite que los archivos se distribuyan a lo largo del clúster, ofreciendo alta escalabilidad y procesamiento paralelo. En la figura 5 se detalla el formato interno de un archivo AVRO.

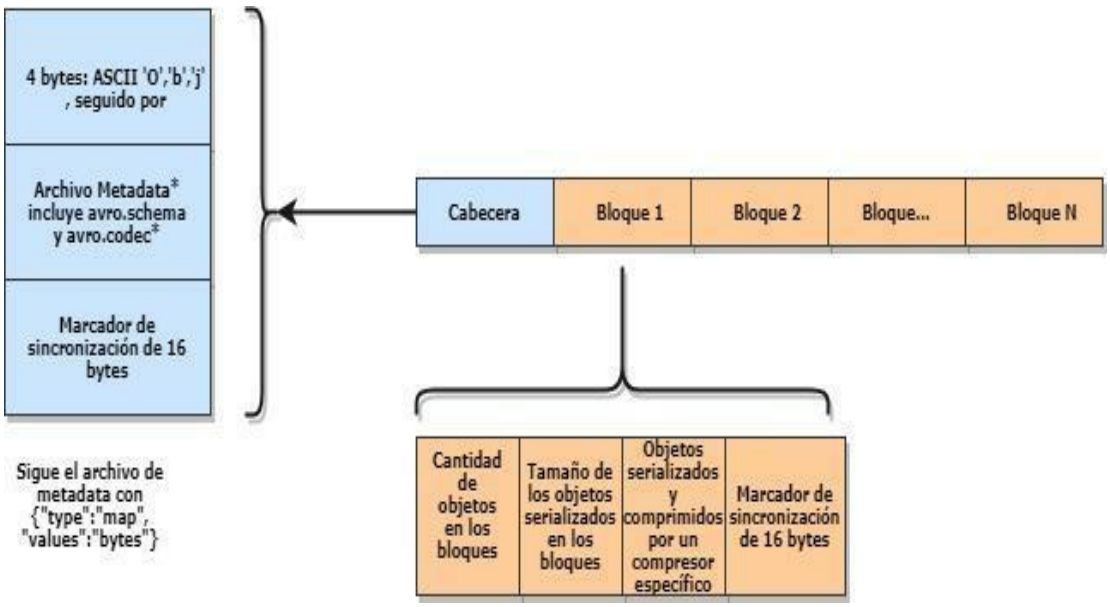

Figura 5: Estructura interna de un archivo AVRO**. Fuente:** Tomado de Apache Hive ORC.

2. Antecedentes de la Investigación

En esta sección se mencionan y detallan algunas investigaciones de otros autores que preceden al presente trabajo y que guardan relación con el tema objeto de estudio.

En el trabajo desarrollado por Borja (2017) se describen los conceptos teóricos y metodológicos que hacen referencia a Apache Flink y Apache Spark. En este trabajo el autor mide la performance de dos algoritmos de machine learning: regresión lineal múltiple y k-means. Utiliza tres bases de datos sintéticas cuyos volúmenes de datos

[3] https://github.com/google/snappy

son de 140MB, 568MB y 1.4GB. Ambos algoritmos los ejecuta con las tres bases de datos en los dos frameworks estudiados comparando el número de procesos utilizado, el uso de CPU, memoria y tráfico de la red interna del cluster.

Si bien los resultados arrojan que Flink ejecuta los algoritmos en menor tiempo y consumiendo menos recursos que Spark, hay que notar que los experimentos llevados a cabo por el autor se realizaron en un cluster de un único nodo. Con lo cual no es posible sacar conclusiones de la performance de ambos frameworks si se ejecutaran en un ambiente distribuido.

En el trabajo realizado por Gordon (2018) los autores estudian y comparan las herramientas Spark, Flink, Kafka, Storm y Hadoop en tareas de procesamiento de flujos de datos haciendo experimentación solo con las dos primeras. Concluyen como resultado que Spark, en particular su módulo Spark Streaming, es la idónea para el procesamiento de los flujos analizados.

En este trabajo solo se comparan los frameworks con criterios funcionales como mantenibilidad, adaptabilidad, eficiencia y seguridad, entre otros. La evaluación obtenida por cada framework está basada en la experiencia de los propios autores y la lectura de otros trabajos de la literatura. Los autores no llevan a cabo una experimentación para medir consumo de recursos o tiempos de ejecución.

En Marcu et al., (2016) realizan la comparación entre Spark y Flink mediante dos tipos de familia de algoritmos, los de una pasada (utilizando wordcount, grep y tera sort) y los algoritmos de características iterativas (k-means, page rank y connected components). El objetivo del trabajo está puesto en determinar la mejor configuración de cada framework para cada uno de los algoritmos utilizados en la experimentación. Las distintas configuraciones con las que realizan los ensayos incluyen el número de procesadores por nodo, buffer de la red interna del cluster, cantidad de memoria RAM por nodo y la serialización de los datos. Todos los experimentos fueron realizados en un cluster homogéneo de 100 nodos midiendo el tiempo de ejecución, uso de memoria RAM, transferencia de datos por la red interna y utilización de memoria en disco. Los resultados obtenidos demuestran que, en líneas generales, Spark resultó más eficiente en el uso de los recursos.

En Alkatheri et al., (2019) comparan la performance de Apache Spark, Apache Flink, Apache Storm y Apache Hadoop midiendo varios factores de performance: tiempo de procesamiento, uso de CPU, latencias, escalabilidad, tolerancia a fallas, entre otros. Los autores realizan la comparación haciendo un review de otros trabajos publicados, no llevando a cabo una propia experimentación.

La comparación entre frameworks, para cada factor medido, la realizan asignando un valor de alto, medio o bajo. Con esa calificación llegan a la conclusión que Spark es ligeramente, en términos generales, el framework de mejor performance.

Existen otros trabajos que realizan comparaciones similares a las mencionadas anteriormente los cuales no presentan aportes diferentes a las discutidas en esta sección (Inoubli et al., 2018; Veiga et al., 2016), aunque la gran mayoría de dichos trabajos se enfocan en el análisis de la performance en procesamiento de flujos de datos (Bartolini and Patella 2017a, 2017b; Perera et al. 2016; Karakaya et al. 2017).

Basado en estos trabajos de investigación, se busca tener otra mirada que permita generar aportes para arribar a conclusiones sobre los objetivos que se plantean en el presente trabajo. En particular se estudiarán los frameworks Apache Spark y Apache Flink en modo batch, haciendo uso de una base de datos real de fármacos llevando a cabo la ejecución de consultas similares a cualquier proceso típico de un datawarhouse. La experimentación es realizada en un cluster heterogéneo de tres nodos, características similares a un cluster que puede armar una pequeña o mediana empresa.

Con la presente investigación se pretende obtener un estudio de performance que le sirva a una Pyme para elegir el mejor framework para la tarea de procesamiento de datos que deba realizar periódicamente.

CAPÍTULO II. APACHE FLINK

1. Introducción

Apache Flink, según Hueske, (2019) es un framework de código abierto para computación distribuida que permite el desarrollo de aplicaciones para el procesamiento de flujos de datos. La herramienta está creciendo con el apoyo de la comunidad, siendo uno de los motores de flujos más sofisticado. De acuerdo con Deshpande, (2017) Flink impulsa aplicaciones comerciales a gran escala en empresas de diferentes industrias en todo el mundo. Inubi et al., (2018), lo define en los siguientes términos:

> Flink is an open source framework for processing data in both real time mode and batch mode. It provides several benefits such as fault-tolerant and large scale computation. The programming model of Flink is similar to MapReduce. By contrast to MapReduce, Flink o↵ers additional high level functions such as join, filter and aggregation. Flink allows iterative processing and real time computation on stream data collected by di↵erent tools such as Flume and Kafka. It o↵ers several APIs on a more abstract level allowing the user to launch distributed computation in a transparent and easy way. Flink ML is a machine learning library that provides a wide range of learning algorithms to create fast and scalable Big Data applications. (p. 6).

El proyecto surgió en el año 2015 y sus características más importantes son las siguientes:

- Procesamiento de flujos de datos: Permite obtener resultados en tiempo real a partir de flujos de datos.
- Procesamiento Batch: Procesamiento de datos históricos y estáticos.
- Aplicaciones orientadas a eventos: Se pueden realizar acciones y dar servicios a partir de los datos procesados en tiempo real.

El procesamiento de los datos en Flink se realiza con el concepto de tupla, es decir, un conjunto de elementos o de tipos de datos simples guardados de forma consecutiva. De esta forma en Flink se define un flujo de datos como una secuencia infinita de tuplas. En la tabla 1 se presentan las características generales de este framework:

Tabla 1. Características generales de Apache Flink.

Criterio	Características
Fácil de usar	El uso de la plataforma puede descargarse directamente desde su página e instalarse en cualquier entorno UNIX, solo se requiere la instalación previa de Java 6.x o superior. Una vez que se ha descomprimido el fichero descargado para comenzar a utilizar la aplicación, se sitúa en la carpeta /bin y correr ejecutable disponible para ello (start-cluster.sh). Por su parte, para su ejecución en el entorno de un clúster, primeramente, se deben conectar los nodos. Una vez que ello se realiza, se da continuidad a los pasos antes referidos, cambiando únicamente el fichero ejecutable (start-duser-sh). De esta forma, se puede disponer de una versión ejecutable
Velocidad	Puede procesar grandes cantidades de datos con un alto rendimiento, disminuyendo la tolerancia al mínimo.
Tolerante a fallas	Ofrece especialmente en la computación Streaming, una tolerancia a fallas. Es una plataforma basada en datos que cada cierto tiempo durante la ejecución Flink guarda un resumen de los datos que se han procesado en una estructura indexada mediante una tupla clave-valor. Esta operación no supone un coste en la eficiencia del mecanismo.
Control de versiones	Es capaz de ofrecer un control de versiones por medio de puntos de salvaguarda; lo cual es muy importante, por cuanto que, permite la actualización de la ejecución desde un determinado punto sin latencias añadidas.
Escalable	Está diseñado para la ejecución de clústeres de tamaño indeterminado. La composición de la red del cluster no afecta al programador en el desarrollo del código, teniendo solo efecto en las métricas de rendimiento. Con la adicción de más recursos computacionales la evaluación de grandes volúmenes de datos

	puede llevarse a cabo en tiempos más reducidos.
Ventanas adaptables	Estas ventanas son períodos de tiempo determinados por el usuario, en los que se agrupan los eventos recibidos durante mucho tiempo para aplicarles unas operaciones determinadas. Dichos eventos pueden ser basados en el tiempo, en la repetición de sucesos, en sesiones, entre otros; es decir, las ventanas permiten que sea posible modelar la realidad del entorno en el que se crean los datos.

Fuente: Elaboración propia a partir de Deshpande, (2017).

A partir de esta sección introductoria, se puede afirmar que las características que definen a Apache Flink, según Fernández, (2018) son las siguientes:

> -It is a framework, because it allows the creation of programs through a set of public APIs, which currently support Java, Scala and the REpresentational State Transfer (REST) architecture. A web interface is also available to easily manage the jobs in execution.
> -It is a distributed engine, and it is built upon a distributed runtime that can be executed in a cluster, to benefit from high availability and high-performance computing resources. A wide range of deployment options are supported: YARN, Mesos, Docker, Kubernettes, Amazon Web Services, Google Compute Engine, and even Hadoop.
> -It is based on stateful computations. Indeed, Flink offers exactly-once state consistency, which means that it is able to ensure correctness even in case of failure. Flink is also scalable, because the state can also be distributed among several systems. (p. 10).

En función a lo expuesto, vale destacar que, Apache Flink consta de tres componentes distribuidos que deben comunicarse: el JobClient, el JobManager y el TaskManager. El JobClient toma una tarea de Flink y la envía al JobManager, quien es responsable de orquestar la ejecución del trabajo. En primer lugar, asigna la cantidad necesaria de recursos, (esto incluye principalmente los slots de ejecución en los TaskManagers), luego despliega las tareas individuales de la tarea a los respectivos TaskManagers. Al recibir una tarea, el TaskManager genera un hilo que la ejecuta. Los cambios de estado, como el inicio del cálculo o el final del mismo se envían de nuevo al JobManager. Basándose en estas actualizaciones de estado, el JobManager dirige la ejecución de la tarea hasta que esté terminada. El resultado del mismo se envía de nuevo al JobClient y éste lo comunica al usuario, tal como se ilustra a continuación:

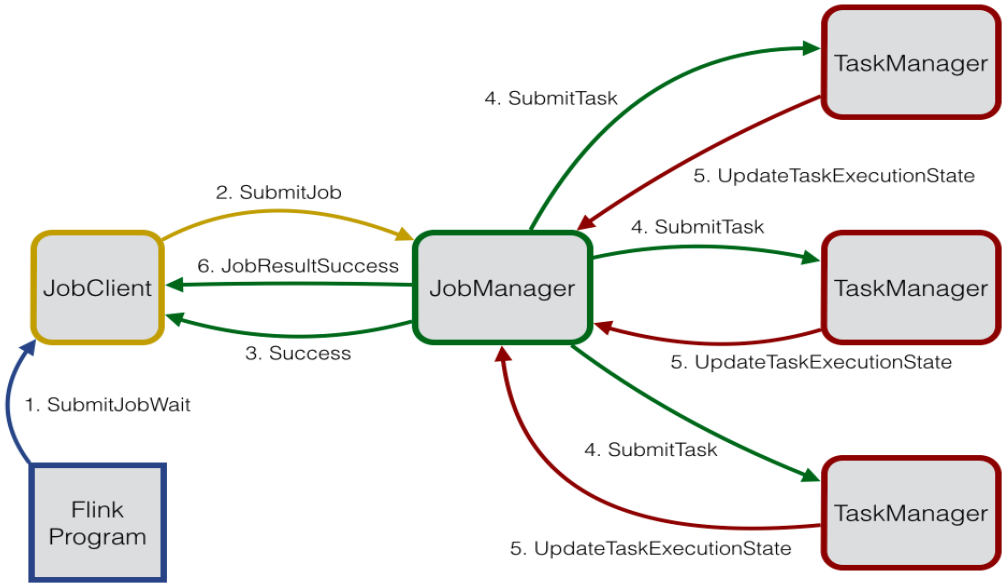

Figura 6: Proceso de ejecución del trabajo. **Fuente:** Tomado de parquet.apache.org.

2. Casos de Uso

Apache Flink, según Hueske, (2019) es una excelente opción para desarrollar y ejecutar muchos tipos diferentes de aplicaciones debido a su amplio conjunto de características. En tal sentido, se puede afirmar que las características de Flink incluyen soporte para el procesamiento de secuencias y lotes, sofisticada gestión de estados, semántica de procesamiento de tiempo de eventos y garantías de consistencia de una sola vez para el estado.

Además, Flink puede desplegarse en varios gestores de recursos como YARN, Apache Mesos, pero también como clúster autónomo en *hardware bare metal* (metal desnudo o expuesto), configurado para una alta disponibilidad. Flink fue probado para escalar a miles de núcleos y terabytes de estado de aplicación; ofrece un alto rendimiento y baja latencia.

3. Aplicaciones Basadas en Eventos

Apache Flink fue desarrollado para soportar las aplicaciones basadas en eventos. Una aplicación basada en eventos es una aplicación de estado que consume eventos

de uno o más flujos de eventos y reacciona a los eventos entrantes activando cálculos, actualizaciones de estado o acciones externas. Los límites de las aplicaciones basadas en eventos se definen en función de la capacidad de un procesador de streaming para manejar el tiempo y el estado. Muchas de las características sobresalientes de Flink se centran en estos conceptos. (Hueske y Kalavi, 2019).

Las aplicaciones basadas en eventos son una evolución del diseño de aplicaciones tradicionales con niveles de almacenamiento de datos y computación separados. En esta modalidad, las aplicaciones leen datos y se graban en una base de datos transaccional remota.

Así mismo, las aplicaciones basadas en eventos se sustentan en aplicaciones de procesamiento de flujos de estado. En este diseño, los datos y el cálculo se encuentran en una misma ubicación, lo que permite el acceso local (en memoria o en disco) a los datos. La tolerancia a fallos se consigue escribiendo periódicamente puntos de control en un almacenamiento remoto persistente. La figura 7, muestra la diferencia entre la arquitectura de aplicación tradicional y las aplicaciones basadas en eventos.

Flink, según Perera et al., (2016) proporciona un rico conjunto de primitivas de estado que pueden gestionar volúmenes de datos muy grandes (hasta varios terabytes) con garantías de consistencia de una sola vez. Además, el soporte de Apache Flink para el tiempo de evento, la lógica de ventana altamente personalizable y el control detallado del tiempo permiten la implementación de una lógica de negocio avanzada. Además, Flink dispone de una librería de Procesamiento de Eventos Complejos (CEP) para detectar patrones en flujos de datos.

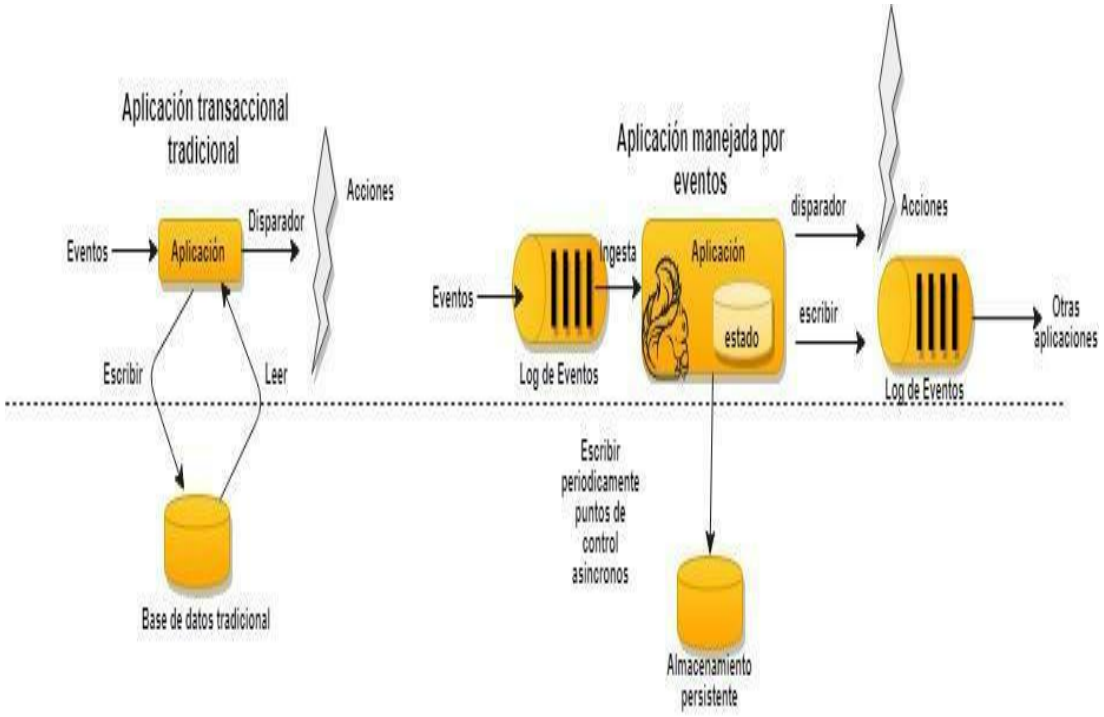

Figura 7: Arquitectura de una aplicación tradicional (izquierda) y arquitectura de una aplicación basada en eventos (derecha). **Fuente:** Tomado de Apache Hive ORC.

Sin embargo, la característica sobresaliente de Flink para aplicaciones basadas en eventos es la denominada como punto seguro (*savepoint*). Un punto seguro es una imagen de estado coherente que puede utilizarse como punto de partida para aplicaciones compatibles. (Hueske y Kalavi, 2019).

4. Aplicaciones de Análisis de Datos

Una de las características de las aplicaciones de análisis de datos es la de extraer información y conocimiento de los datos sin procesar. Tradicionalmente, estos análisis se realizan como consultas por lotes o aplicaciones en conjuntos de datos limitados de eventos registrados. Para incorporar los últimos datos en el resultado del análisis, hay que añadirlos al conjunto de datos analizado y se vuelve a ejecutar la consulta o aplicación. Los resultados se escriben en un sistema de almacenamiento o se emiten como informes. (Marcu et al., 2016).

Con un sofisticado motor de procesamiento de flujos, el análisis también puede realizarse en tiempo real. En lugar de leer conjuntos de datos finitos, las consultas o

aplicaciones de streaming ingieren flujos de eventos en tiempo real y producen y actualizan continuamente los resultados a medida que se consumen los eventos. Los resultados se escriben en una base de datos externa o se mantienen como estado interno. La aplicación Dashboard puede leer los últimos resultados de la base de datos externa o consultar directamente el estado interno de la aplicación. Apache Flink soporta aplicaciones de streaming, así como aplicaciones analíticas por lotes como se muestra en la figura 8.

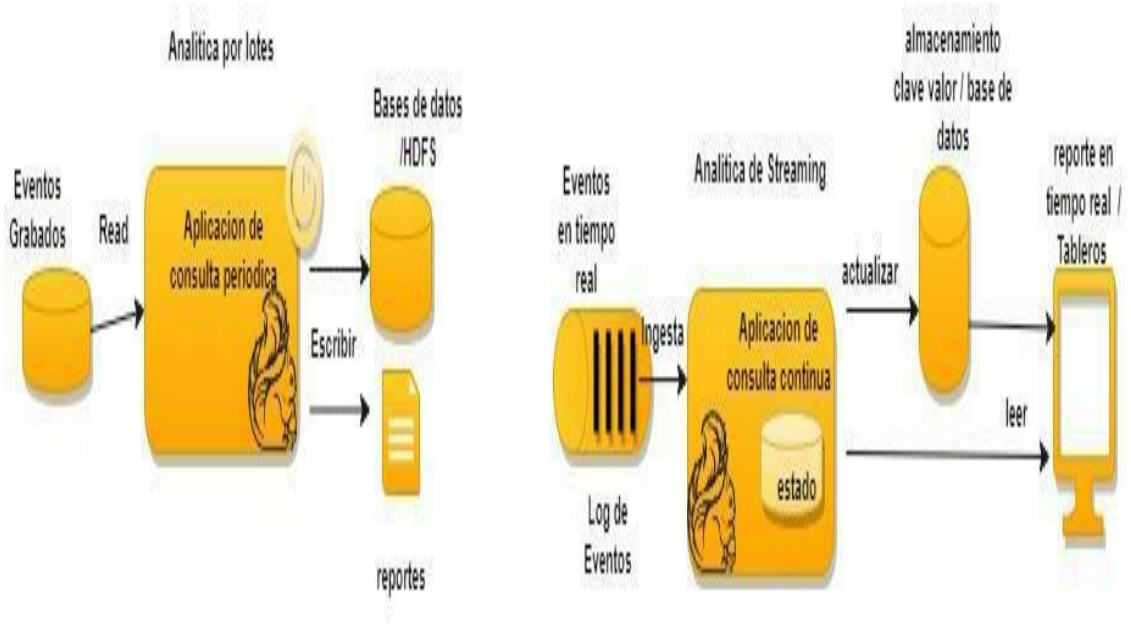

Figura 8: Arquitectura del análisis en batch (izquierda) y arquitectura del análisis en streaming (derecha). **Fuente:** Tomado de Apache Hive ORC.

Apache Flink proporciona un soporte muy bueno para el streaming continuo, así como para el análisis por lotes. Específicamente, cuenta con una interfaz SQL compatible con ANSI con semántica unificada para consultas por lotes y streaming. Las consultas SQL calculan el mismo resultado independientemente de si se ejecutan en un conjunto de datos estáticos de eventos registrados o en un flujo de eventos en tiempo real. La compatibilidad con las funciones definidas por el usuario garantiza que el código personalizado se pueda ejecutar en las consultas SQL. Si se requiere aún más lógica personalizada, la API DataStream de Flink o la API DataSet proporcionan más control de bajo nivel. Además, la biblioteca Gelly de Apache Flink proporciona

algoritmos y bloques de construcción para análisis gráficos a gran escala y de alto rendimiento en conjuntos de datos por lotes. (Hueske y Kalavi, 2019).

5. Aplicaciones de Data Pipelines.

Extract-Transform-Load (ETL) es un enfoque común para convertir y mover datos entre sistemas de almacenamiento. A menudo, los trabajos de ETL se activan periódicamente para copiar datos de sistemas de bases de datos transaccionales a una base de datos analítica o a un almacén de datos. (Hueske y Kalavi, 2019).

Las tuberías de datos tienen un propósito similar al de los trabajos de ETL: transforman y enriquecen los datos y pueden moverlos de un sistema de almacenamiento a otro. Sin embargo, funcionan en modo de streaming continuo en lugar de activarse periódicamente. Por lo tanto, son capaces de leer registros de fuentes que continuamente producen datos y moverlos con baja latencia a su destino. La figura 9 devela la diferencia entre los trabajos periódicos de ETL y las tuberías de datos continuos, tal como se muestra a continuación:

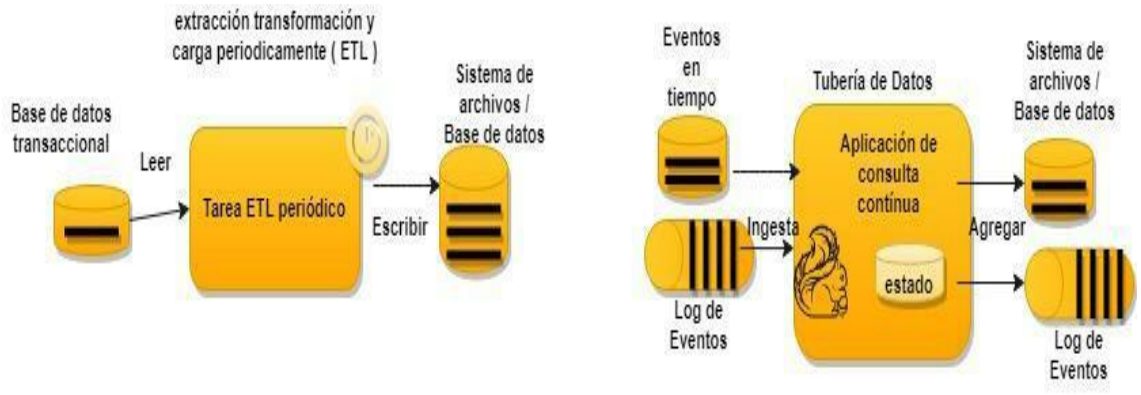

Figura 9 Arquitectura de un trabajo periódico (izquierda) y una tubería de datos (derecha). **Fuente:** Tomado de Apache Hive ORC.

La ventaja de las tuberías de datos continuos sobre los trabajos periódicos de ETL es la reducción de la latencia de mover los datos a su destino. Además, las tuberías de datos son más versátiles y pueden emplearse para más casos de uso porque son capaces de consumir y emitir datos continuamente. (Hueske y Kalavi, 2019).

Apache Flink posee la capacidad de soportar tuberías de datos. Muchas de las tareas comunes de transformación o enriquecimiento de datos pueden ser abordadas por la interfaz SQL de Flink (o Table API) y su soporte para funciones definidas por el

usuario. Las tuberías de datos con requisitos más avanzados se pueden realizar utilizando la API de DataStream, que es más genérica.

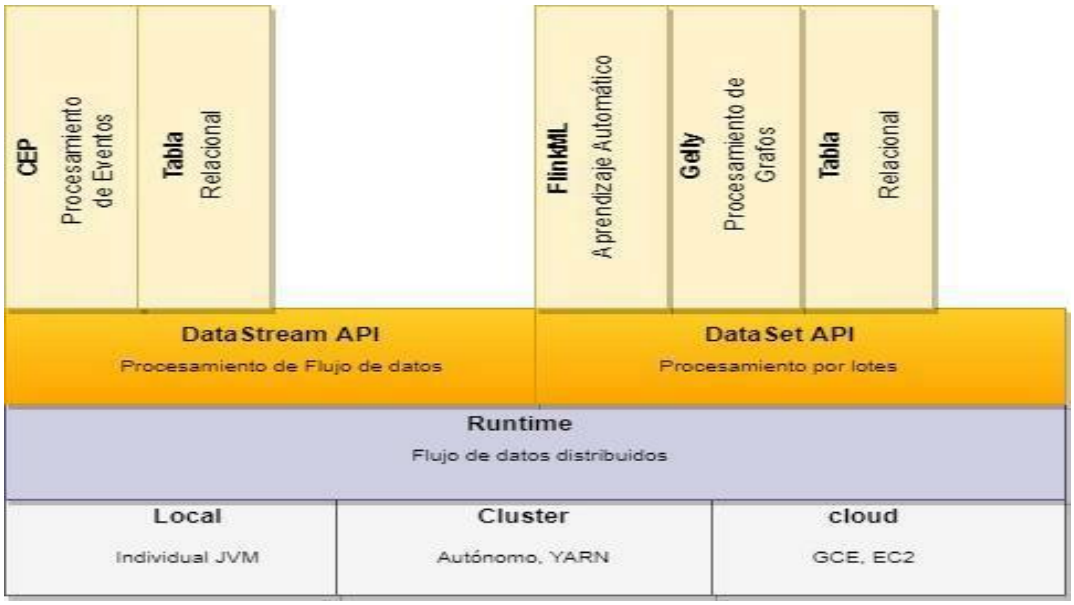

Figura 10: Arquitectura de Apache Flink. **Fuente:** Tomado de parquet.apache.org.

6. FlinkCEP - Procesamiento de Eventos Complejos para Flink.

Con la ubicuidad de las redes de sensores y los dispositivos inteligentes que recopilan cada vez más datos, el ser humano se enfrenta al reto de analizar un flujo de datos cada vez mayor prácticamente en tiempo real. Ser capaz de reaccionar rápidamente a las tendencias cambiantes o de ofrecer una inteligencia de negocio actualizada puede ser un factor decisivo para el éxito o el fracaso de una empresa. (Hueske, 2019).

Un problema clave en el procesamiento en tiempo real es la detección de patrones de eventos en los flujos de datos; el tratamiento de eventos complejos (CEP) aborda exactamente este problema de emparejar continuamente los eventos entrantes con un patrón.

El resultado de una conciliación, según Hueske, (2019) normalmente concluye en eventos complejos que se derivan de los eventos de entrada. A diferencia de los DBMS tradicionales en los que se ejecuta una consulta en datos almacenados, CEP ejecuta

datos en una consulta almacenada. Todos los datos que no son relevantes para la consulta pueden ser descartados inmediatamente. Hay grandes ventajas en este enfoque, dado que las consultas CEP se aplican a un flujo de datos potencialmente infinito. Además, las entradas se procesan inmediatamente: una vez que el sistema ha visto todos los eventos de una secuencia coincidente, los resultados se emiten inmediatamente. Este aspecto conduce efectivamente a la capacidad de análisis en tiempo real de CEP.

En consecuencia, el paradigma de procesamiento de CEP encuentra aplicación en una amplia variedad de casos de uso. Además, se utiliza en el seguimiento y la supervisión basados en RFID, El CEP también puede utilizarse para detectar intrusiones en la red especificando patrones de comportamiento sospechoso de los usuarios.

Apache Flink, con su verdadera naturaleza de streaming y sus capacidades de baja latencia, así como de procesamiento de streaming de alto rendimiento, es una opción natural para las cargas de trabajo CEP. En consecuencia, la comunidad Flink ha introducido la primera versión de una nueva biblioteca CEP con Flink 1.0. Un ejemplo donde se ve el uso eficiente del CEP es en el monitoreo y generación de alertas para centros de datos. (Hueske, 2019).

7. Tabla API y SQL

Apache Flink incluye dos APIs relacionales (la Table API y SQL) para un procesamiento unificado de flujos y lotes. La Table API es una API de consulta integrada en lenguaje para Scala y Java que permite la composición de consultas de operadores relacionales como selección, filtrado y unión de forma muy intuitiva. El soporte SQL de Apache Flink está basado en Apache Calcite el cual implementa el estándar SQL. Las consultas especificadas en cualquiera de las interfaces tienen la misma semántica y especifican el mismo resultado independientemente de si el input es un batch input (DataSet) o un flujo de entrada (DataStream). (Saxena y Grupta, 2017).

Las interfaces Table API y SQL están estrechamente integradas entre sí, así como las API de Flink DataStream y DataSet. El código creado puede ser modificado

fácilmente entre todas las APIs y bibliotecas que se basan en las APIs. Por ejemplo, puede extraer patrones de un DataStream usando la librería CEP y luego usar la Table API para analizar los patrones, o puede escanear, filtrar y agregar una tabla de lotes usando una consulta SQL antes de ejecutar un algoritmo de Gelly en los datos pre procesados. (Hueske y Kalavi, 2019).

7.1 FlinkSQL

Flink SQL es el SDK API para SQL proporcionado por Flink. SQL es una API de nivel superior a la de la tabla. Está integrado en la biblioteca de tablas y puede utilizarse para desarrollar consultas sobre flujos como en lotes. A continuación, se muestra la estructura de FlinkSQL.

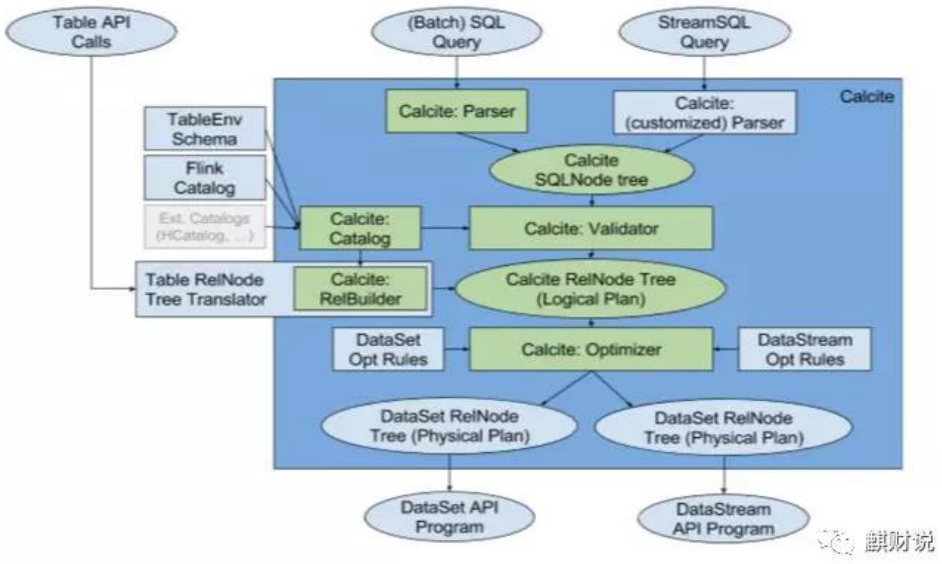

Figura 11: Arquitectura de FlinkSQL. **Fuente:** Tomado de parquet.apache.org.

8. Librerías

Se han desarrollado librerías para Flink las cuales permiten el desarrollo de aplicaciones específicas. Estas librerías incluyen algoritmos de machine learning para el desarrollo de aplicaciones inteligentes y funciones que permiten realizar gráficos de manera fácil como resultado de un procesamiento. (Perera et al., 2016).

8.1 FlinkML

Según Hueske y Kalavi (2019) FlinkML es una librería de conjuntos de algoritmos de Machine Learning soportados por Flink que pueden ser utilizados para resolver problemas complejos de uso en la vida real. Los algoritmos están construidos para que puedan usar la potencia de computación distribuida de Flink y hacer predicciones o encontrar clústeres con facilidad. Si bien en este momento sólo se admiten unos pocos conjuntos de algoritmos, la lista se incrementa.

8.2 API de gráficos Flink – Gelly

En la era de los medios sociales donde todos están conectados entre sí por algún medio, cada objeto tiene una relación con otro. Facebook y Twitter son excelentes ejemplos, se resalta que son redes sociales, y es a partir de la información extraída de ellas se puede construir de gráficos sociales; donde X es amigo de Y y P sigue a Q, y así sucesivamente. (Hueske y Kalavi, 2019). Estas gráficas son tan grandes que se requiere un motor que pueda procesarlas eficientemente.

9. API de Flink DataStream

Perera et al., (2016) señala que los programas DataStream en Flink son programas regulares que implementan transformaciones en los flujos de datos (por ejemplo: filtrado, estado de actualización, definición de ventanas, agregación). Los flujos de datos se crean inicialmente a partir de varias fuentes (por ejemplo: colas de mensajes, flujos de socket, archivos). Los resultados se devuelven a través de Sinks que pueden escribir los datos en archivos o en una salida estándar (por ejemplo: terminal de línea de comandos). Los programas Flink se ejecutan en una variedad de contextos, de forma independiente o incrustados en otros programas. La ejecución puede ocurrir en una JVM local, o en clúster de muchas máquinas.

9.1 Transformaciones de DataStream

Algunas de las transformaciones más usadas en operaciones de flujos, según Hueske y Kalavi, (2019) son los Map, los cuales toman un elemento y produce un elemento el cual es resultado de una transformación; los FlatMap que se caracterizan

por tomar un elemento y produce cero, uno o más elementos; los Filter que evalúan una función booleana para cada elemento y retiene aquellos para los que la función devuelve verdadero; el Reduce, estableciendo una reducción "rodante" en un flujo de datos clave, además combina el elemento actual con el último valor reducido y emite el nuevo valor y por último el Windows el cual agrupa los datos en cada clave de acuerdo a alguna característica (por ejemplo, los datos que llegaron en los últimos 5 segundos).

9.2API de Flink DataSet

9.2.1 Transformaciones de DataSet

Las transformaciones de DataSet se enumeran en la tabla 2.

Tabla 2. Transformaciones de DataSet.

Elemento	Características
Map	Aplica una función definida por el usuario en cada elemento del datasets. Implementa un mapeo uno a uno y exactamente un elemento debe ser devuelto por la función.
FlatMap	Toma un elemento y produce cero, uno o más elementos
Filter	Evalúa una función booleana para cada elemento y retiene aquellos para los que la función devuelve verdadero.
Reduce	Combina un grupo de elementos en un solo elemento combinando repetidamente dos elementos en uno. Reduce puede aplicarse a un conjunto de datos completo o a un conjunto de datos agrupados.

Fuente: Elaboración Propia a partir de, *https://programmerclick.com/article/74841043334/.*

CAPÍTULO III. APACHE SPARK

En este capítulo se profundiza en los elementos que distinguen a Apache Spark como otro framework que está siendo utilizado para el procesamiento de datos a gran escala. (https://spark.apache.org/).

1. Introducción

Apache Spark es un motor informático unificado y un conjunto de librerías utilizadas para el procesamiento paralelo de datos en clústeres informáticos. Apache Spark es el motor de código abierto más activamente desarrollado, lo cual lo convierte en una herramienta estándar para cualquier desarrollador o científico de datos interesado en datos de gran tamaño. Bartolini y Patella, (2017), la definen en los siguientes términos:

> Apache Spark Streaming Apache Spark (spark.apache.org) is an open source platform, originally developed at the University of California, Berkeley's AMPLab. The main idea of Spark is that of storing data into a so-called Resilient Distributed Dataset (RDD), which represents a read-only fault-tolerant collection of (Python, Java, or Scala) objects partitioned across a set of machines that can be stored in main memory. RDDs are immutable and their operations are lazy. The "lineage" of every RDD, i.e., the sequence of operations that produced it, is kept so that it can be rebuilt in case of failures, thus guaranteeing faulttolerance. Every Spark application is therefore a sequence of operations on such RDDs, with the goal of producing the desired final result. (p. 2).

Apache Spark soporta múltiples lenguajes de programación utilizados de manera masiva (Python, Java, Scala y R), incluye bibliotecas para diversas tareas que van desde SQL hasta flujos y aprendizaje automático. Puede ejecutarse desde cualquier lugar, ya sea desde una computadora portátil hasta un grupo de miles de servidores. Esto hace que sea un sistema fácil de iniciar y escalar hacia un gran procesamiento de datos o a una escala increíblemente grande.

A partir de lo antes expuesto, se pueden distinguir las características generales detalladas en la tabla 3.

Tabla 3. Características generales de Apache Spark.

Criterios	Características
Fácil de usar	La aplicación se puede obtener de su página web, seleccionando la

	versión requerida, considerando un sistema operativo UNIX, se necesita tener instalada una máquina Java. Se configura el archivo de arranque del terminal (. bashrc) añadiendo las variables de entorno Java-HOME y Spark-HOME definiendo su ruta y localización ejecutables. Al completar este procedimiento, se fija la ruta a través del cual se descomprime el fichero descargado. En tal sentido, accediendo a esa ruta a través de la terminal una versión ejecutable de Spark está disponible en Scala o Python, contando así con un entorno interactivo en el que se pueden desarrollar aplicaciones.
Escalable	Posibilita la adición de recursos computacionales sin necesidad de hacer modificaciones de códigos de las aplicaciones. Esta versatilidad, posibilita que la arquitectura de la que esté compuesto el entorno donde se ejecuta el programa permanezca invisible al desarrollador.
Tolerante a fallas	Spark cuenta con dos mecanismos que le distinguen como tolerante a fallas: primero la replicación de los datos en diferentes nodos y la construcción de una línea con el histórico de operaciones realizadas sobre datos. A partir de ello, en caso de que un fragmento de la información se corrompa, la aplicación puede a partir de uno de los datos originales continuar las operaciones que se estaban realizando antes del fallo para reconstruir el estado de la ejecución en dicho momento.
Rápido	Se distingue su rapidez por el almacenamiento en memoria de datos y en su motor de ejecución. En tal sentido, Spark permite el almacenamiento de los datos en caché, reduciendo el número de operaciones de entrada y salida que realizan en disco, disminuyendo con ello la latencia en las operaciones. El acceso a la memoria es de orden de 100 veces más rápido que la lectura de los mismos datos en disco.

Fuente: Elaboración propia a partir de Macías et al., (2016).

Apache Spark comienza en la Universidad de California en Berkeley en 2009 como el primer proyecto de investigación de Spark. (https://spark.apache.org/). En ese momento, Hadoop MapReduce era el framework de programación paralela dominante para clústeres, siendo el primer sistema de código abierto en abordar el procesamiento paralelo de datos agrupados en miles de nodos. Zaharia y sus colaboradores del AMPlab estaban trabajando con varios usuarios de MapReduce para entender los beneficios e inconvenientes de este nuevo modelo de programación, y por lo tanto fue capaz de sintetizar una lista de problemas de uso en varios casos y comenzar a diseñar a un nivel más general un nuevo framework que solucionara todos los inconvenientes identificados.

Al respecto, Alkateri et al., (2019), explica las siguientes características de Spark:

> Apache Spark is an open source framework that was established at the University of California, Berkeley. It became an Apache project in 2013, providing faster services with large-scale data processing. Spark framework is to Hadoop what MapReduce is to data processing and HDFS. In addition, Spark has data sharing known as Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG). Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation. (p. 68).

En este sentido, se puede afirmar que Spark, se convirtió en este momento como un proyecto verdaderamente novedoso, que proporcionó servicios de procesamiento de datos mucho más rápidos y a gran escala.

Además, Zaharia, et al., (2010) también ha trabajado con usuarios de Hadoop en UC Berkeley para entender cuáles son sus necesidades respecto a la plataforma. A través de este trabajo, quedan esclarecidas dos cuestiones. En primer lugar, la computación en clúster tuvo un enorme potencial: en cada organización que utiliza MapReduce, se pueden construir aplicaciones totalmente nuevas utilizando los datos existentes. En segundo lugar, el motor MapReduce hizo que fuera a la vez desafiante e ineficiente construir una gran aplicación.

Por ejemplo, el típico algoritmo de aprendizaje de la máquina puede necesitar hacer 10 o 20 pasadas por encima de los datos, y en MapReduce cada pase tiene que ser escrito como un MapReduce separado. A su vez cada uno debe lanzarse por separado en el clúster y cargar así los datos desde cero.

Para resolver este problema, el equipo de Spark diseña primero una API basada en programación funcional que puede expresar aplicaciones de varios pasos. (https://spark.apache.org/). A continuación, el equipo implementa esta API a través de un nuevo motor que permite compartir datos en memoria de forma eficiente durante todos los pasos de la computación.

También comienza a probarse este sistema tanto con Berkeley como con usuarios externos. La primera versión de Spark sólo soporta aplicaciones por lotes, pero muy pronto otra se hizo evidente: la ciencia de datos interactiva y las consultas ad hoc. Simplemente conectando el intérprete de Scala a Spark, el proyecto puede proporcionar una interfaz interactiva ágilmente utilizable para ejecutar consultas en cientos de máquinas.

2. DataFrame

Una de las principales herramientas con las que cuenta Apache Spark son los DataFrame. Según Veija et al., (2016) estos representan la API estructurada más común y consiste en una tabla de datos con filas y columnas. La lista que define las columnas y los tipos dentro de ellas se denomina esquema. Se puede pensar en un DataFrame como una hoja de cálculo con columnas con nombre con una diferencia fundamental: una hoja de cálculo se almacena en una única computadora, mientras que un Spark DataFrame puede abarcar miles de computadoras. Existen motivos por los que almacenar datos en más de una computadora: o bien los datos son demasiado grandes y no caben en una sola, o requiere demasiado tiempo realizar ese cálculo en una única máquina.

Apache Spark tiene varias abstracciones centrales: Datasets, DataFrames, Tablas SQL y Resilient Distributed Conjuntos de datos (RDDs). Todas estas diferentes abstracciones representan colecciones de datos distribuidos. Los de uso más fácil y eficientes son los DataFrame, que se encuentran disponibles en casi todos los lenguajes de programación.

2.1 Particiones

Para permitir que cada ejecutor realice su trabajo en paralelo, Apache Spark divide los datos en trozos llamados particiones. Una partición es una colección de filas que se

graban en una máquina física en su clúster. (Marcu et al., 2016). Las particiones de DataFrame representan la distribución física de los datos a través del clúster durante la ejecución. Si sólo existe una partición, Apache Spark emplea el paralelismo de sólo una, incluso si ella tiene miles de ejecutores. Si tiene muchas particiones, pero sólo un ejecutor, puede seguir manteniendo un único paralelismo porque sólo hay un recurso de computación. Una cosa importante a tener en cuenta con respecto a los DataFrame, es que no se manipulan (en su mayor parte) manualmente o individualmente. Simplemente se especifican las transformaciones de alto nivel de los datos en él y Apache Spark determina cómo se ejecuta el trabajo en el clúster.

2.2 Transformaciones

En Apache Spark, las estructuras de datos básicos son inmutables, lo que significa que no se pueden modificar luego de ser creadas. Las instrucciones que permiten cambiar un DataFrame se denominan transformaciones. En la figura 12 se muestra una transformación simple para encontrar todos los números pares en el DataFrame actual:

```
// in Scala
val divisBy2 = myRange.where("number % 2 = 0")

# in Python
divisBy2 = myRange.where("number % 2 = 0")
```

Figura 12: Transformación Simple. **Fuente:** Tomado de Apache Hive ORC.

Estos datos no devuelven ninguna salida debido a que sólo se especifica una transformación abstracta, y Apache Spark no actúa sobre las transformaciones hasta ser llamado a la acción. Las transformaciones son el núcleo de la forma en que se expresa la lógica de negocio utilizando dicho framework. Hay dos tipos de transformaciones: las que especifican dependencias estrechas, y las que especifican dependencias anchas. Las transformaciones que consisten en dependencias o transformaciones estrechas son aquellas para las cuales cada partición de entrada

contribuye a una sola partición de salida. En el ejemplo ilustrado a través de la figura 12, la sentencia *where* especifica una dependencia estrecha, donde sólo una partición contribuye a un máximo de una partición de salida como se muestra 13.

Figura 13: Transformaciones Estrechas. **Fuente:** Tomado de Apache Hive ORC.

Una transformación de estilo de dependencia (o transformación amplia) tiene particiones de entrada que contribuyen a muchas particiones de salida. (Macía et al., 2016). A menudo esta tarea se conoce como un Shuffle (barajada) en la que Spark intercambia particiones a través del clúster. Con transformaciones estrechas, Spark puede realizar automáticamente una operación llamada pipelining, lo que significa que, si se especifican múltiples filtros en los DataFrames, todos se realizan en memoria.

3. Arquitectura y Componentes

El componente Spark Core contiene las funcionalidades básicas de Apache Spark indispensable para la ejecución de trabajos y también necesarios por otros componentes. La más importante de ellas es el conjunto de datos distribuidos resilientes (RDD), que es la unidad de trabajo principal de la API de Spark. Es una abstracción de una colección distribuida de ítems con operaciones y transformaciones aplicables al conjunto de datos. Es capaz de reconstruir conjuntos de datos en caso de fallas en los nodos. (Ryza et al., 2017). La figura 14 muestra los distintos componentes que conforman a Spark.

Figura 14. Componentes que conforman a Spark. **Fuente:** Tomado de diegocalvo.es.

Spark Core contiene la lógica para acceder a varios sistemas de archivos, como lo son los del tipo HDFS. Además, proporciona un medio de intercambio de información entre los sistemas informáticos, nodos con variables de emisión y acumuladores. Otras funciones fundamentales, tales como manejo de redes, seguridad, programación y transferencia de datos, también forman parte de Spark Core, como se muestra a continuación:

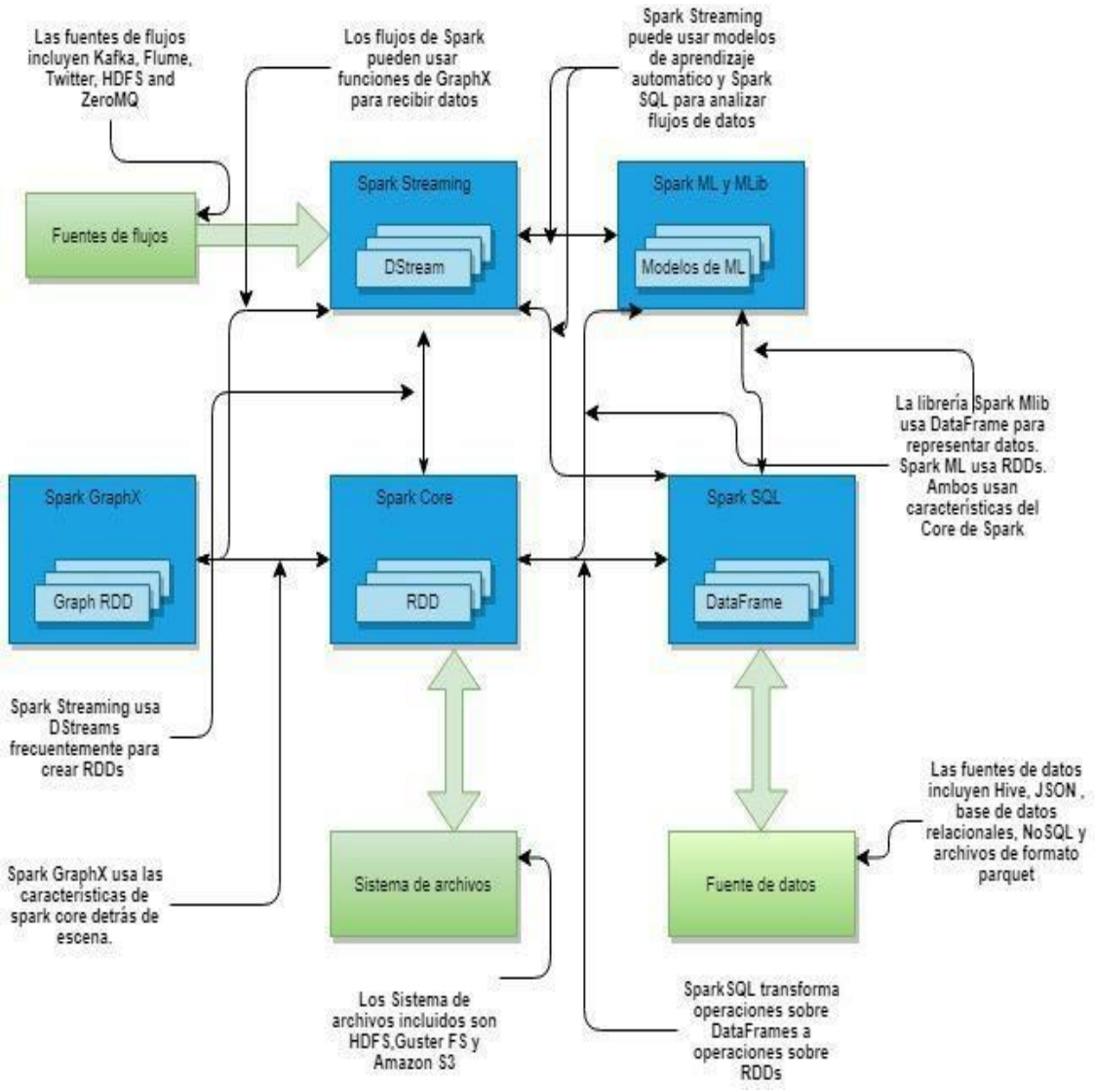

Figura 15. Componentes de Spark. **Fuente:** Tomado de diegocalvo.es.

Spark SQL proporciona funciones para manipular grandes conjuntos de datos distribuidos y estructurados utilizando un subconjunto SQL soportado por Spark y Hive SQL (HiveQL). Los DataFrames introducidos en Spark 1.3, y DataSets introducidos en Spark 1.6, simplificaron el manejo de datos estructurados y han permitido optimizaciones radicales en el rendimiento, así Spark SQL se ha convertido en uno de los componentes más importantes de Spark. Otra utilización común está destinada a leer y escribir datos desde (y hacia) diversos formatos estructurados, como es el caso

de los archivos JSON (JavaScript Object Notation), archivos de Parquet, bases de datos relacionales, Hive y otros. (https://spark.apache.org/).

Las operaciones en DataFrames y DataSets en determinado momento se traducen en operaciones RDDs y se ejecutan como trabajos normales de Spark. Spark SQL proporciona un marco de optimización de consultas llamado Catalyst que puede ser extendido por reglas de optimización personalizadas. Spark SQL también incluye un servidor Thrift, el cual puede ser utilizado por sistemas externos, tales como el correspondiente a inteligencia de negocios, con el objetivo de consultar datos a través de Spark SQL haciendo uso de los protocolos clásicos JDBCy ODBC. En la figura 16 se puede observar la arquitectura de Spark SQL.

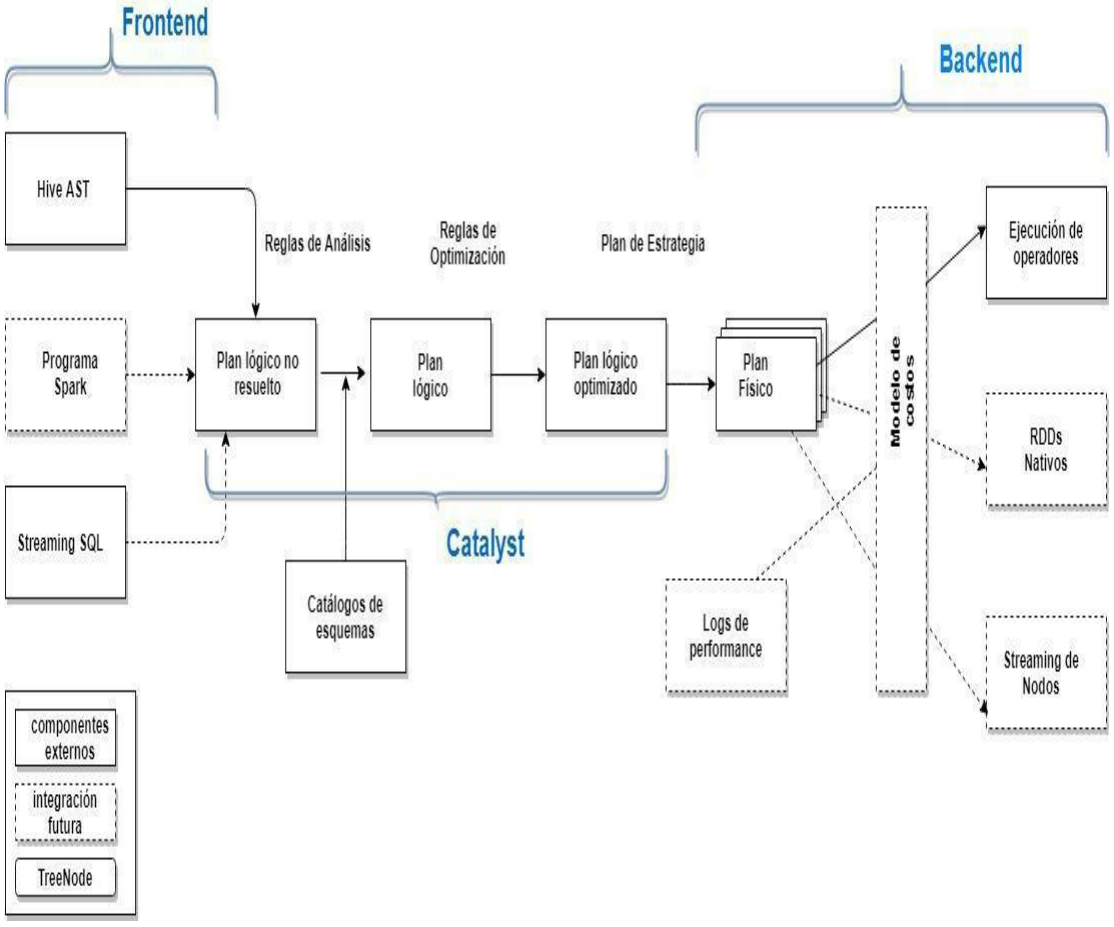

Figura 16. Arquitectura de Spark SQL. **Fuente:** Tomado de blog.bi-geek.com/.

Spark Streaming según Macía et al., (2016) es un marco de trabajo para la ingesta de datos provenientes de streaming en tiempo real. Estos datos pertenecen a diversas fuentes, las soportadas incluyen HDFS, Kafka, Flume y otras fuentes personalizadas. Las operaciones de Spark Streaming se recuperan de un fallo automáticamente, lo cual es importante para el procesamiento de datos en línea. La transmisión por Spark representa datos de streaming utilizando flujos discretizados (DStreams), quienes periódicamente crean RDDs que contengan los datos que entraron durante la última ventana de tiempo. Spark Streaming puede combinarse con otros componentes de Spark en un solo programa, unificando el procesamiento en tiempo real con tareas de aprendizaje automático, el SQL y las operaciones gráficas. Esto representa algo único en el ecosistema de Hadoop. Desde Spark 2.0, la nueva API estructurada de streaming hace que los programas de streaming de Spark sean más similares a los programas batch de Spark, tal como se ilustra en la figura 17.

Marcu et al., (2016) indica que, Spark MLlib es una librería de algoritmos de aprendizaje de máquina desarrollados a partir del proyecto MLbase en la Universidad de California en Berkeley. Los algoritmos soportados incluyen regresión logística, clasificación de Naive Bayes, máquinas vectoriales de sporte (SVMs), árboles de decisión, regresión lineal y k-means, tal como se puede observar en la figura 18. Además, Spark MLlib maneja modelos de aprendizaje de máquina utilizados para la transformación de conjuntos de datos, que se representan como RDDs o DataFrames.

Las herramientas de extracción, transformación y carga (ETL) proliferaron junto con el crecimiento de los datos de las organizaciones. Trasladar los datos desde una fuente a uno o más destinos, procesándolos antes de que lleguen a destino, comenzando entonces a ser un requisito. En un principio estas herramientas admiten pocos tipos de datos, fuentes y destinos. Esto se debe a las limitaciones mencionadas, puesto que los procesos de transformación de un solo paso se ejecutan en múltiples pasos, de modo que se produce un desperdicio de recursos en términos de eficiencia, tiempo y costos. Debido a ello, muchas organizaciones entran en el bloqueo de proveedores debido a la profunda necesidad de procesar datos. Casi todas las herramientas introducidas antes del año 2005 no utilizan el poder real de la arquitectura multinúcleo de las computadoras, sino que se ejecutan utilizando un solo núcleo, por lo tanto, los trabajos

de procesamiento de datos simples pero voluminosos tardan demasiado tiempo en completarse. Apache Spark se ha convertido en un éxito instantáneo en el mercado debido a su capacidad para procesar una gran cantidad de tipos de datos con un número creciente de fuentes y destinos.

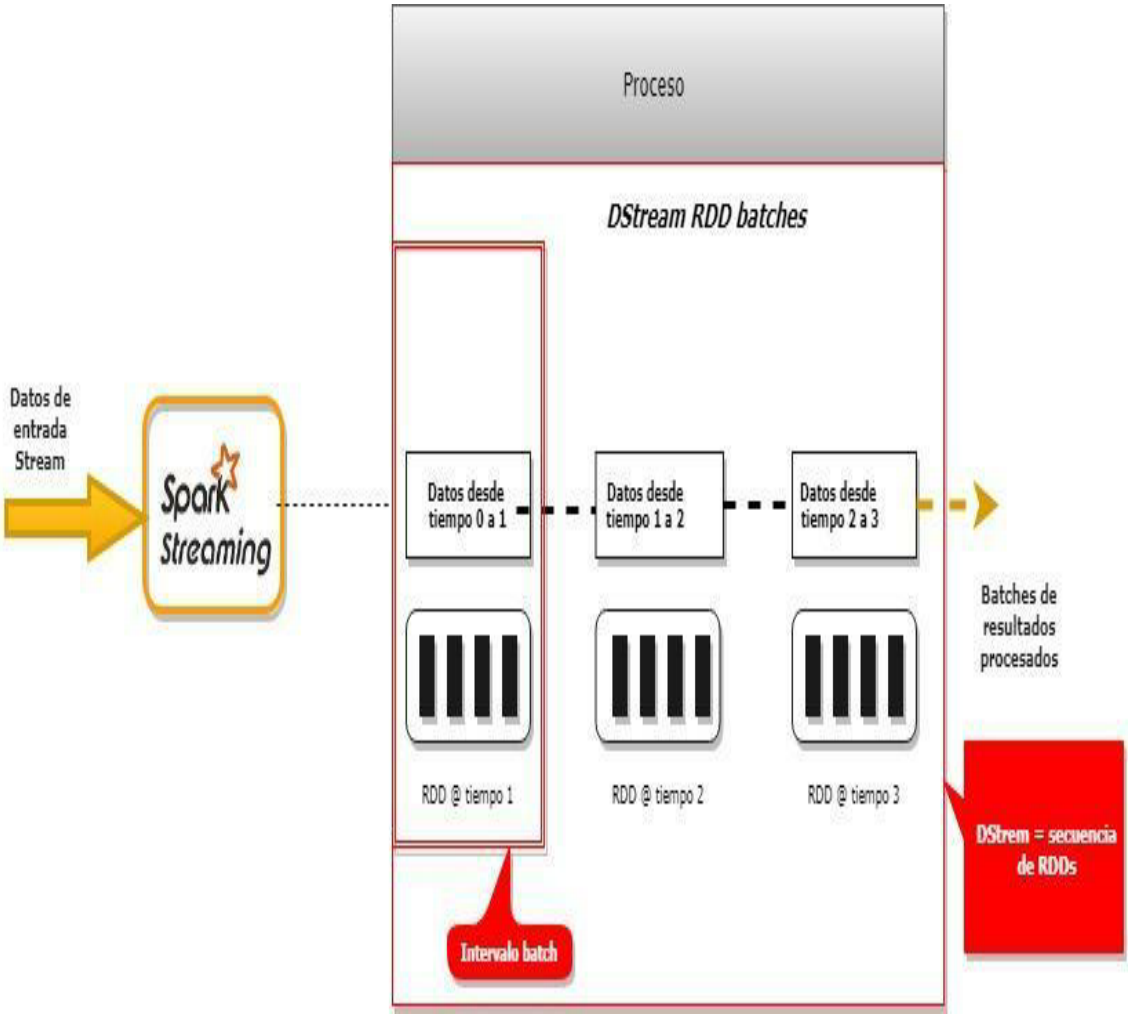

Figura 17: Procesamiento de Spark Streaming. **Fuente:** Tomado de Spark Overview.

Figura 18: Arquitectura de Spark MLlib. **Fuente:** Tomado de Spark Overview.

La abstracción de datos más importante y básica que proporciona Spark es el conjunto de datos distribuidos (RDD), lo que hace que soporte el procesamiento distribuido en un grupo de nodos dentro de un clúster. (Ryza et al., 2017). Cuando hay un grupo de nodos, existen posibilidades de que alguno de ellos deje de funcionar durante el procesamiento de los datos. La parte resiliente de RDD ha sido diseñada para solventar esta falla.

Si hay una gran cantidad de datos por procesar y hay nodos disponibles en el clúster, el framework debe tener la capacidad de dividir el gran conjunto de datos en trozos más pequeños y distribuirlos para ser procesados en más de un nodo en un clúster, en paralelo. Spark es capaz de realizar esa acción y eso es lo que significa la parte distribuida en el RDD. (Macía et al., 2016). En otras palabras, Apache Spark está diseñado desde cero para tener su abstracción básica de conjuntos de datos capaz de dividirse en piezas más pequeñas de forma determinística y distribuirse a más de un nodo del clúster para su procesamiento en paralelo, a la vez que maneja con elegancia las fallas en los nodos.

4. Programación Funcional con Spark

La mutación de objetos en tiempo de ejecución y la incapacidad de obtener resultados consistentes de un programa o función debido al efecto secundario que la lógica del programa crea, hace que muchas aplicaciones sean muy complejas. Si las funciones de los lenguajes de programación se comportan exactamente igual que las funciones matemáticas, de tal manera que la salida de la función depende sólo de las entradas, esto da mucha previsibilidad a las aplicaciones. (Macía et al., 2016). El paradigma de programación de computadoras que otorga demasiado énfasis al proceso de construcción de tales funciones y otros elementos basados en eso, y el uso de esas funciones justo en la forma en que cualquier otro tipo de datos están siendo utilizados, es popularmente conocido como el paradigma de programación funcional. Fuera de los lenguajes de programación basados en JVM, Scala es uno de los más importantes que tiene una capacidad de programación funcional muy fuerte sin perder ninguna orientación a objetos. Spark se escribe predominantemente en Scala. Por eso mismo, Spark ha asimilado muchos conceptos positivos de Scala.

5. Spark RDD

RDD es la estructura de datos fundamental de Apache Spark. Se trata de una colección de particiones de sólo lectura de registros. Sólo puede ser creado a través de una operación determinista en cualquiera de los dos: Datos en almacenamiento estable, otros RDDs, y paralelización de colecciones ya existentes en el programa de RDD. El mismo refiere a una colección distribuida inmutable de datos, particionada a través de nodos en el clúster que puede ser operada en paralelo con una API de bajo nivel ofreciendo transformaciones y acciones.

La característica más importante que Apache Spark toma de Scala, es la capacidad de usar funciones como parámetros para sus transformaciones y acciones. Muy a menudo, el RDD de Spark se comporta como un objeto de colección en Scala. Por eso, algunos de los nombres de los métodos de transformación de datos de las colecciones de Scala se utilizan en Spark RDD para realizar la misma acción. Este es un enfoque muy limpio y aquellos que tienen experiencia en Scala encontraran el mismo fácil de programar con RDDs.

Algunas de sus características se distinguen en la Tabla 4.

Tabla 4. Características de Spark RDD.

Característica	Descripción
Es inmutable	Existen algunas reglas fuertes sobre las cuales se crea un RDD. Una vez que un RDD es creado, intencionalmente o no, no puede ser cambiado. Esto brinda otra idea de la construcción de un RDD. Por ello, cuando los nodos que procesan alguna parte de un RDD mueren por alguna falla, el programa controlador puede recrear esas partes y asignar la tarea de procesarlo a otro nodo, completando en última instancia el trabajo de procesamiento de datos con éxito. Dado que el RDD es inmutable, el procedimiento de dividir uno grande entre uno más pequeño, distribuirlo a varios nodos de trabajo para su procesamiento, y finalmente compilar los resultados para producir el resultado final se puede realizar con seguridad sin que los datos subyacentes cambien
Es distribuible	Si Apache Spark se ejecuta en modo clúster donde hay múltiples nodos de trabajo disponibles para llevar a cabo las tareas, todos estos nodos tendrán diferentes contextos de ejecución. Las tareas individuales se distribuyen y se ejecutan en diferentes JVM. Todas estas actividades de un gran RDD, que se divide en trozos pequeño, se distribuyen para su procesamiento a los nodos de trabajo y, finalmente, se ensamblan los resultados. Todas estas acciones son transparentes para los usuarios. Spark tiene su propio mecanismo para la recuperación de las fallas del sistema y otras formas de errores que ocurren durante el procesamiento de datos y, por lo tanto, esta abstracción de datos es altamente resistente.

Tabla 4. Continuación

Característica	Descripción
Se ejecuta en la memoria.	Apache Spark mantiene todos los RDDs en la memoria tanto como le es posible. Sólo en situaciones poco usuales, cuando se está quedando sin memoria o si el tamaño de los datos está creciendo más allá de su capacidad, se escribe en el disco. La mayor parte del procesamiento en RDD ocurre entonces en la memoria, y esa es la razón por la que Spark es capaz de procesar los datos a una alta velocidad.
Está fuertemente tipado	Spark RDD puede ser creado usando cualquier tipo de datos soportados. Estos tipos de datos pueden ser tipos de datos intrínsecos soportados por Scala/Java o tipos de datos creados a medida como sus propias clases.

Fuente: Elaboración propia a partir de, https://programmerclick.com/article/608915293/.

5.1 Evaluación Perezosa

La evaluación de los RDDs es completamente perezosa. Apache Spark no empieza a computar las particiones hasta que se llame a una acción. Una acción es una operación de Spark que devuelve algo que no sea un RDD, desencadenando la evaluación de las particiones, por ejemplo, devolver los datos al driver (con operaciones como count o collect). Las acciones son desencadenadas por el driver, que construye un gráfico acíclico dirigido (llamado DAG), basado en las dependencias entre las transformaciones de los RDDs. (Macía et al., 2016). En otras palabras, Spark evalúa una acción que consiste en ir hacia atrás para definir qué serie de pasos tiene que dar para producir determinado objeto en cada partición. Entonces, utilizando esta serie llamada plan de ejecución driver, se calcula las particiones que faltan para cada etapa hasta que finalmente se compute el resultado.

Aunque es preciso destacar que no todas las transformaciones en Spark son 100% perezosas, por ejemplo, la función sortByKey necesita evaluar el RDD para determinar que rango de datos va a ordenar. Esto involucra una transformación y una acción.

La evaluación perezosa según Ryza et al., (2017) permite a Apache Spark combinar operaciones que no requieren comunicación con el driver (llamadas transformaciones de dependencia uno a uno) y esto impide que se hagan varias lecturas a los mismos datos. Por ejemplo, si se parte del supuesto de que un programa Spark llama a las funciones Map y a filter para procesar un RDD. Apache Spark puede enviar las instrucciones para map y filter en cada ejecutor, entonces realiza ambas operaciones sobre cada partición y de esta forma acceder a los datos solo una vez. Esto resulta más eficiente que enviar dos conjuntos de instrucciones y acceder a cada partición dos veces. Según la teoría esta acción baja el tiempo computacional a la mitad.

5.2 Limitaciones de las RDDs

Apache Spark RDD presenta distintas limitaciones que mediante un uso apropiado de DataFrame y Dataset pueden ser superadas. Entre estas limitaciones se destaca que no tiene un motor de optimización de entrada, en este sentido, partiendo de la base de que no existe ninguna disposición en el RDD para la optimización automática, no puede hacer uso de optimizadores de avance de Spark como el optimizador de Catalyst y el motor de ejecución de tungsteno. Sin embargo, se puede optimizar cada RDD manualmente. (Saxena y Grupta, 2017). Esta limitación se supera en Dataset y DataFrame, ambos hacen uso de Catalyst para generar un plan de consulta lógico y físico optimizado. Además, se puede utilizar el mismo optimizador de código para R, Java, Scala o Python DataFrame/Dataset APIs. Aunque es prudente indicar que esto proporciona un incremento en términos de eficiencia en cuanto al espacio y a la velocidad.

Inoubli et al., (2018) indica que otra de las limitaciones consiste en la seguridad del tipo de funcionamiento, dado que el RDD se degrada cuando no hay suficiente memoria para almacenar el RDD en memoria o en el disco. El problema de almacenamiento aparece cuando existe una falta de memoria para almacenar el RDD. Las particiones que se desbordan de RAM se pueden almacenar en el disco y

proporciona el mismo nivel de rendimiento. Es posible superar este inconveniente aumentando el tamaño de la memoria RAM y del disco.

Asimismo, es necesario destacar la limitación en cuanto a rendimiento, en el sentido que el RDD es un objeto JVM en memoria, implica la sobrecarga del Garbage Collector y la serialización de Java, lo que resulta costoso cuando los datos crecen. Dado que el coste de la recogida de basuras es proporcional al número de objetos Java, el uso de estructuras de datos con menos objetos reducirá el coste. La otra opción posible es persistir en el objeto en forma serializada.

De la misma manera, vale destacar de acuerdo con Saxena y Grupta, (2017) que el manejo de datos estructurados no proporciona una vista de esquema de los datos y tampoco tiene ninguna disposición para el manejo de datos estructurados. Dataset y DataFrame proporcionan la vista esquema de datos, la cual es una colección distribuida de datos organizada en columnas con nombre. Estas representaban las principales limitaciones de RDD en Apache Spark.

Como resultado de las limitaciones de RDD, se devela la necesidad de crear DataFrame y Dataset. De esta manera, el sistema se ha hecho más amigable facilitando el trabajo con un gran volumen de datos.

6. La Aplicación Spark

Una aplicación Spark, según Macia et al., (2016) corresponde a un conjunto de tareas definidas por un SparkContext en el programa driver. Una aplicación Spark comienza cuando se inicia un SparkContext. Al iniciarlo, se inicia un driver y una serie de ejecutores en los nodos de trabajo del clúster. Cada ejecutor posee su propia máquina virtual Java (JVM), y cada uno no puede abarcar múltiples nodos, aunque si un nodo puede contener varios ejecutores. El SparkContext determina cuántos recursos se asignan a cada ejecutor.

Cuando se lanza un trabajo de Apache Spark, cada ejecutor tiene sus slots destinado a ejecutar las tareas necesarias para calcular un RDD. De esta manera, se puede pensar en un SparkContext como un conjunto de configuración de parámetros que tienen como finalidad ejecutar los trabajos de Spark. Estos parámetros se exponen en la sección SparkConf, la cual sirve para crear un SparkContext. Un nodo puede

tener varios ejecutores de Spark, pero un nodo no puede abarcar múltiples nodos. (Karau, 2017). Un RDD es evaluado a través de los ejecutores en las particiones. Cada ejecutor puede tener varias particiones, pero una partición no puede extenderse a varios ejecutores.

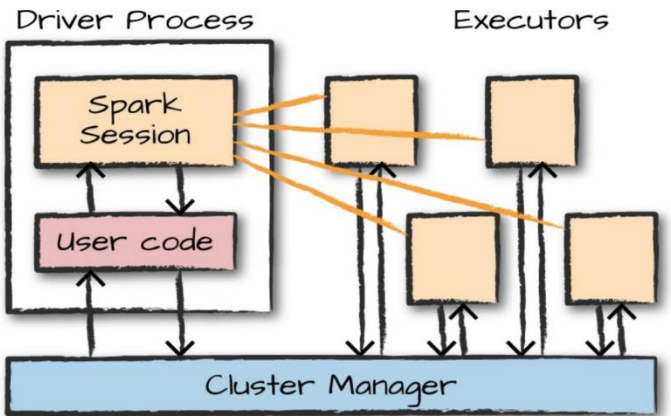

Figura 19. Arquitectura de la ejecución de un SparkContext. **Fuente:** Tomado de Spark Overview.

7. Agregaciones

La agregación es el acto de resumir y se considera la piedra angular de los grandes análisis de datos. En una agregación, se especifica una clave o agrupación y una función de agregación que especifique cómo debe transformar una o más columnas. Esta función debe producir un resultado con múltiples valores de entrada. Las capacidades de agregación de Spark son sofisticadas, con una variedad de diferentes casos de uso y posibilidades. (Karau, 2017). En general, las agregaciones se utilizan para integrar datos numéricos generalmente por medio de alguna agrupación. Esto puede ser una suma, un producto o un simple conteo, además de trabajar con cualquier tipo de valores.

Agregar a tipos complejos en Spark, puede realizar agregaciones no sólo de valores numéricos utilizando fórmulas, sino que también puede realizarlos en tipos complejos. (Inoubli et al., 2018). Por ejemplo, se puede recopilar una lista de valores presentes en un archivo o sólo los valores unívocos recogidos en un set. Puede utilizarlo para llevar

a cabo algún acceso más programático más adelante en el pipeline o pasar la colección completa en una función definida por el usuario (UDF).

7.1 Funciones de Agregación

Todas las agregaciones están disponibles como funciones. (Inoubli et al., 2018). Entre estas agregaciones se destaca Sum, la cual constituye una tarea simple que suma todos los valores en una columna y Avg, aunque se puede calcular el promedio dividiendo la suma por el conteo, Apache Spark proporciona una manera fácil de obtener ese valor a través de las funciones AVG o media. En este ejemplo, se usa alias para obtener más y reutilizar fácilmente estas columnas.

8. Persistencia en Memoria y Gestión de la Memoria

La ventaja de rendimiento de Spark sobre MapReduce es mayor en casos donde el uso involucre cálculos repetidos. Gran parte de este incremento en el rendimiento se debe de contar con persistencia en memoria. En lugar de escribir en el disco entre cada paso a través de la ventana de diálogo, Spark tiene la opción de mantener los datos de los ejecutores cargados en la memoria. (Karau, 2017). De esta manera, los datos de cada partición están disponibles en memoria cada vez que sea necesario a la que se ha accedido. Apache Spark ofrece tres opciones para la gestión de la memoria: en memoria como datos de serializados, en memoria como datos serializados, y en disco.

9. La Anatomía de un Job de Spark

En el paradigma de evaluación de Spark, una aplicación de Spark no realiza ninguna acción hasta que el programa driver lo inicie. Con cada acción, el programador de Apache Spark crea un gráfico de ejecución y lanza un trabajo. Cada tarea consta de etapas, que son pasos en la transformación de los datos necesarios para materializar el RDD final. Cada operación consiste en un conjunto de tareas que representan un cálculo paralelo y son los ejecutores. La figura muestra un árbol de los diferentes componentes de una aplicación de Apache Spark y cómo corresponden a las llamadas de la API. Cada trabajo puede contener varias etapas que corresponden a cada transformación amplia. (Karau, 2017). A su vez, cada etapa se compone de una o

varias tareas que corresponden a una unidad de cálculo realizada en cada etapa. Hay una tarea para cada partición en el RDD resultante de esa etapa. Este proceso se ilustra en la Figura 20.

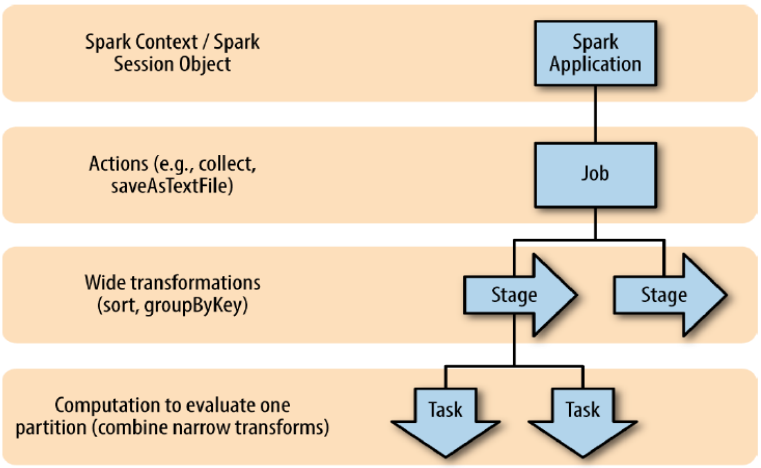

Figura 20: La anatomía de un Job de Spark. (Fuente: Tomado de Spark Overview)

10. Gráfico Acíclico dirigido DAG

Apache Spark viene con un motor de procesamiento de datos DAG (Grafo Acíclico Dirigido) avanzado. Lo que significa que por cada trabajo de Spark, se crea un DAG de tareas para ser ejecutado por el motor. El DAG en lenguaje matemático consiste en un conjunto de vértices y bordes dirigidos que los conectan. Las tareas se ejecutan según la disposición DAG. (Macia et al., 2016).

En el caso de MapReduce, el DAG consiste en sólo dos vértices, uno para la tarea de mapa y el otro para la tarea de reducir. El borde se dirige desde el vértice del mapa, hacia el vértice de reducción. El procesamiento de datos en memoria combinado con su motor de procesamiento de datos basado en DAG hace que Spark sea muy eficiente.

En el caso de Apache Spark, el DAG de las tareas puede ser tan complicado como sea posible. Apache Spark viene con utilidades que pueden dar una excelente visualización del DAG de cualquier trabajo de Spark que se esté ejecutando.

CAPÍTULO IV. RESULTADOS

Luego de haber analizado los conceptos básicos de los dos frameworks objeto de estudio para esta investigación, se procede a exponer de manera minuciosa los experimentos realizados y los resultados obtenidos en la comparativa de la ejecución del cálculo del KPI en una base de datos entre los frameworks Apache Flink y Apache Spark.

La base de datos de estudio pertenece a una empresa multinacional que se dedica a la venta de información de medicamentos para farmacias y laboratorios. Contiene información de puntos de ventas de medicamentos de los cuales también se cuenta con mucha información: drogas que contiene, fórmula química, presentaciones (packs) en las que se vende, etc.

Las consultas realizadas con esta base de datos resultan de un trabajo particular que tuve que realizar para un cliente el cual necesitaba el cálculo de dos índices, denominados en este trabajo como Rotation Vault (RV) y Rotation Unit (RU). En la sección 7 de este capítulo se dan más detalles sobre el cálculo de estas consultas.

1. Descripción del Problema de datawarehouse

En Datawarehouse, la mayor parte del procesamiento de datos es por lotes, los cuales demoran mucho tiempo en su ejecución, dado que generalmente no contemplan todos los cálculos requeridos, debiendo realizar agregaciones un tanto complejas. Dada esta realidad, en el ámbito corporativo se plantea la cuestión respecto a qué herramientas y tecnologías deben ser implementadas en la organización para satisfacer al negocio en términos de tiempo y toma de decisión.

Una de las tareas más comunes dentro de una organización o empresa es el cálculo de los indicadores claves de rendimiento (KPI, key performance indicator). Un KPI es un valor medible que informa si la organización o empresa es eficaz en los logros de sus objetivos principales. Dentro de la organización o empresa, los KPI pueden utilizarse en diferentes áreas o a diferentes niveles para evaluar si se alcanzan o no los objetivos a nivel general de la organización, empresa, un departamento, un producto o

un proceso concreto. Todos los KPI se recogen y organizan en un cuadro que aporta una visión amplia del logro de los objetivos de la organización o empresa.

En este mismo orden de ideas, en este trabajo se utiliza el cálculo de un KPI particular sobre información de ventas de productos. Es importante notar que en este trabajo no interesa obtener el valor propio del KPI ni determinar si es bueno o no, sino medir la performance del cálculo del mismo en un ambiente distribuido, utilizando los frameworks Flink y Spark para la ejecución del proceso de cálculo.

Para poder emprender el cálculo de estos (KPI), por la cantidad de datos a procesar, de la base de datos original se necesita crear un conjunto más pequeño de datos limpios. A este conjunto se le ha denominado tabla *view_precalculated* que contiene los datos que se procesan en las dos consultas siguientes. Así, cada una de ellas ofrece un KPI que devela la tendencia sobre una variable del negocio, que indica si éste va bien o no, siendo ella la única función del KPI.

De igual forma, la Tabla *view_precalculated* se conforma de otras tablas, para reducir el tiempo de consulta, a razón de que no sea necesario construir productos cartesianos entre tablas, así como ninguna otra operación adicional que promueva una mayor inversión en tiempo. De esta manera, teniendo todos los resultados precalculados, los cálculos de las otras dos consultas van a demorar menos.

Ahora bien, la primera consulta denominada *Rotation Unit* brinda información respecto a cuánto se vende a cada sucursal respecto de cada producto, de esta forma se puede identificar el nivel de penetración que tiene el producto en la zona donde está ubicado el punto de venta. De manera que, este nivel es muy importante, ya que indica si hay que comprar más o menos productos de un tipo determinado o marca según la salida que tiene al público, pudiendo así orientar la adquisición de mercadería de forma más eficaz.

De la misma manera, una segunda consulta denominada *Rotation Vault* ofrece información sobre a qué proveedor comprarle más, dado que representa a cierta marca que presenta mayor volumen de ventas y a qué proveedor comprarle menos por la poca rotación del producto.

Por su parte, esta segunda consulta suma en unidades y divide por punto de venta, ofreciendo información respecto a la penetración de los puntos de ventas sobre las ganancias de los productos para un tiempo determinado.

Es de destacar que ambas consultas en el Datawarehouse generan un mal tiempo de respuesta, ocasionando problemas de tiempo de respuesta de otros procesos y retrasando de esta forma la operatoria en general del almacén de datos.

2. Planteo del Problema utilizando Spark y Flink

En función a lo expuesto anteriormente, a la luz de desarrollar el análisis comparativo entre Apache Flink y Apache Spark, que posibilite la medición de la performance en la ejecución de algoritmos tradicionales de un Datawarehouse a fin de valorar la efectividad de ambos framework, se presentan los resultados que se han obtenido en la ejecución de las pruebas desarrolladas.

Para llevar a cabo las pruebas se ha creado en primera instancia la tabla *view_precalculated* a modo de soporte y fuente de datos para la creación de agregaciones. En segundo lugar, se construyen dos agregaciones basadas en los datos contenidos en la tabla. Asimismo, es de destacar que la generación de la tabla *view_precalculated* tendrá la información que le servirá al proceso siguiente, el cual podrá obtener la información ordenada para un buen rendimiento.

De igual forma, es de destacar nuevamente que la agregación *Rotation Unit* (de ahora en más llamada como RU) suma la cantidad de unidades de un producto y la divide por los puntos de venta, esto muestra la penetración de los puntos de venta por cantidad de productos, mientras que la agregación *Rotation Vault* (de ahora en más llamada como RV) suma las unidades y las divide por los puntos de venta. Esto da la penetración de los puntos de venta sobre la ganancia de los productos.

3. Descripción de la Base de Datos

La base de datos de pruebas fue convertida de una base de datos Oracle a un clúster, respetando su formato y su estructura. Para la prueba se utilizan las tablas detalladas a continuación:

- Packs: contiene 40 columnas con información acerca del precio, fecha de lanzamiento y nombre comercial del fármaco entre otros detalles sobre el medicamento. Tiene una cantidad de 99.155 registros y los tipos de datos son String.
- Products: contiene 20 columnas que dan información acerca de los productos que se utilizan para el procesamiento. Esta tabla precisa información descriptiva acerca del país donde se vende el producto, a qué tipo de fármaco pertenece, si es un antibiótico, un antidepresivo, etc. Cuenta con un set de datos tipo String de 81.402 registros. No tiene compresión. Es la entidad más importante ya que el negocio se basa en la venta de productos.
- Sell_point: contiene 62 columnas con información de los lugares donde se vende el producto: identificador del país, razón social, si es farmacia o distribuidor, dirección, entre otros datos. Cuenta con un total de 627.432 registros donde cada campo es tipo String y no tiene compresión. Usando esta tabla se puede determinar qué producto se vende más en cada punto de venta y bajo qué forma (Packs).
- Details: contiene las transacciones por punto de venta y pack proveedor. La información es transaccional y cuenta con 8.351.286.956 registros, 10 columnas tipo String y un peso de 500MB.
- typed_sell_point**:** contiene los tipos de puntos de venta según su ubicación. Cuenta con 20 registros y siete campos de tipo String sin compresión.
- Nidde: tabla auxiliar que contiene el código postal según la zona geográfica. Tiene un total de 1.083.393 registros y un peso de 23.8 MB.
- Regions: contiene 28 registros y ubica en qué regiones se venden los productos.
- Presentations: indica la presentación del medicamento. Contiene 6042 registros sin compresión.

El formato de todos los archivos es CSV.

4. Experimentos Realizados

La experimentación realizada en este trabajo consta de dos análisis. En el primer análisis se definen los contextos de configuración aplicados sobre los frameworks. Estos se basaron en los recursos *cores* y *memory*. El objetivo es determinar que configuración resulta más eficiente en términos de memoria, uso de CPU y transferencia de datos.

Tabla 5. Contextos y sus Configuraciones.

	Cores	Memoria GB
Contexto C1	1	2
Contexto C2	8	4
Contexto C3	16	8

Fuente: Elaboración Propia. (2021).

Para el segundo análisis se toma, del análisis anterior, la configuración que obtuvo el mejor desempeño, se reduce la dimensionalidad de las tablas de la base de datos de estudio y se aplica la conversión de datos al formato columnar ORC. El objetivo de este análisis es poder realizar una comparativa entre la performance de ambos frameworks cuando los datos están almacenados en archivos con formato de texto plano (csv) y cuando están almacenados en archivos de formato columnar.

5. Hardware Utilizado

Para la experimentación se contó con el siguiente hardware

- Una computadora con un procesador i7 de octava generación 32GB de RAM disco M2 de 256GB y un disco SATA de 2TB. Interfaz de Red ethernet. Placa de Red Ati Radeon 4GB
- Una computadora con un Core i7 770k con 64GB de RAM 2 Placas Nvidia 1080 TI paralelas, disco estado sólido de 500GB y una disco sata de 2TB, Interfaz de red ethernet.

- Una computadora con un Core i7 de octava generación 32GB de RAM disco M2 de 256GB y un disco SATA de 2TB con una placa Nvidia 1060, Interfaz de red ethernet.

Todas las computadoras se conectaron de forma cableada con un switch de 12 bocas. También se contó con un almacenamiento externo de 6 discos rígidos (12TB de almacenamiento en total).

6. Herramientas de Monitoreo

Para la medición de recursos consumidos por los diferentes ensayos se utilizaron herramientas de monitoreo específicas, como Ganglia y JConsole.

Ganglia

Es un sistema de monitoreo escalable y distribuido de computación de alta performance como clústeres y redes. Utiliza XML para representación de datos, XDR para transporte de datos y RRD para el almacenamiento de datos y visualización. Ha sido diseñado para lograr bajo overhead y alta concurrencia por nodo conectado.

La herramienta contiene los siguientes componentes:

- Gmond: Es un proceso que corre en segundo plano el cual tiene la tarea de recolectar la información de cada nodo del clúster y enviarla al nodo central para su procesamiento.
- Gmetad: Es un proceso que corre en segundo plano que recibe la información recolectada por el proceso gmond y guarda la información en una base de datos la cual es usada por la interface web para armar los gráficos de medición.
- Web Interface: Este componente se encarga de renderizar los gráficos.

JConsole

Es una herramienta de monitoreo de procesos gráfica construida en Java. Esta aplicación visualiza cuatro gráficas, el uso de la memoria, los hilos del proceso, las clases cargadas, el uso de la CPU y el tiempo de ejecución. Además, incluye información detallada sobre el uso de la memoria. Igualmente aporta gráficas y “combos” para seleccionar un tipo de memoria concreta. Como tercera utilidad permite monitorear información detallada e individual del uso de hilos, e igualmente muestra su

propia gráfica. También cuenta con una utilidad de monitoreo que informa sobre el número de clases cargadas y descargadas. Además, muestra información sobre la máquina virtual, como el uso de hilos, memoria consumida y algunos parámetros del OS. Una última utilidad permite presentar una serie de directorios en árbol que contiene los métodos y clases que permiten controlar el proceso java.

7. Consultas SQL Implementadas

En esta sección se describen las consultas SQL para la ejecución de las pruebas involucradas en la experimentación: la creación de una tabla principal llamada *view_precalculated* que contiene los datos para la creación de las agregaciones y las consultas SQL para la creación de las agregaciones RU y RV.

7.1 Tabla view_precalculated

La tabla *view_precalculated* es creada en base a una consulta SQL y el resultado es grabado en el disco rígido. La misma trae un extracto de la información cuyo objetivo es simplificar la creación y obtención de las agregaciones RU y RV en cuanto a cantidad de columnas y cantidad de registros. De esta manera se pueden tener los campos para ambas agregaciones sin la necesidad de tener que hacer consultas muy complejas.

La consulta SQL es la siguiente:

```
SELECT d.code_date, d.counter, d.price, pks.pack_code,
       prod.product_code, pe.presentation_code,
       sp.sell_point_code, re.code_state
```

Figura 21. 7.1. Tabla view_precalculated. **Fuente:** Elaboración Propia. (2021).

```
FROM Details d, sell_points sp, Typed_sell_point ty, NIDD
       nx, Regions re, packs pks, products prod,
       presentations pe
WHERE d.ch = %s AND d.counter != %s
    AND d.type_transaction = %s
    AND d.code_date > %s
    AND d.code_sell_point = sp.sell_point_code
```

```
    AND sp.type_sell_point_code = ty.type_sell_point_code
    AND ty.channel_code = %s
    AND Nx.zipcode = sp.postal_code
    AND Nx.state = re.state
    AND d.code_pack = pks.pack_code
    AND cast(pks.code_product as int) = prod.product_code
    AND pks.code_presentation = pe.presentation_code
```

Figura. 22. Consulta SQL. **Fuente:** Elaboración Propia. (2021).

7.2 Agregación RU

La agregación RU suma la cantidad de unidades de un producto y la divide por los puntos de venta, esto muestra la penetración de los puntos de venta por cantidad de productos.

La consulta realiza la suma de las cantidades de unidades de un producto especificado en el campo “counter” para obtener el resultado de las unidades del producto que se obtiene aplicando los filtros sobre los campos código de paquete y código de fecha de venta. Luego divide este resultado por la cantidad de puntos de ventas que vendieron el producto. Para esta operación, se toma siempre el código de paquete y su fecha de venta para la obtención de la métrica sobre la vista *view_precalculate*.

El cálculo utilizando consultas SQL es el siguiente:

```
ru = ( SELECT sum( cast( counter as double ) )
       FROM view_precalculate
       WHERE pack_code = %s ) /
     ( SELECT count( DISTINCT sell_point_code )
       FROM view_precalculate
       WHERE pack_code = %s )
```

Figura 23. Agregación RU. **Fuente:** Elaboración Propia. (2021).

7.3 Agregación RV

Esta agregación suma las unidades y las divide por los puntos de venta esto da la penetración de los puntos de venta sobre la ganancia de los productos, la consulta suma los precios de los productos en una fecha determinada y luego lo divide por la cantidad de puntos de venta distintos en donde se vendió este producto.

El cálculo utilizando consultas SQL es el siguiente:

```
rv = ( SELECT sum( cast( price as double ) )
       FROM view_precalculate
       WHERE pack_code = %s ) /
     ( SELECT count( DISTINCT sell_point_code )
       FROM view_precalculate
       WHERE pack_code = %s )
```

Figura 24. Agregación RV. **Fuente:** Elaboración Propia. (2021).

Tabla 6. Definición del dataset según la consulta SQL que construye la tabla *view_precalculated*.

DataSet: view_precalculated			
Tabla	Alias	Campo	Detalle
Details	d	code_date	Fecha
Details	d	counter	Cantidad de productos
Details	d	price	Cantidad de productos
Packs	pks	pack_code	Código de Embalaje
Products	prod	producto_code	Código de producto
Presentations	pe	presentation_code	Código de presentación
sell_points	sp	Sell_point_code	Código de punto de venta

Regions	re	code_state	Código de estado

Fuente: Elaboración propia. (2021).

Tabla 7. Descripción de objetos de tipo Tabla alcanzados en la conformación de la vista view_precalculated.

Objetos	Tabla	Registros
1	Packs	99.155
2	Products	81.402
3	Sell_point	627.432
4	Details	8.351.286.956
5	typed_sell_point	20
6	Nidde	1.083.393
7	Regions	28
8	Presentations	6.042

Fuente: Elaboración propia. (2021).

8. Experimentación

8.1 Resultados de la ejecución en un datawarehouse convencional

Para poder realizar comparaciones entre la ejecución de las consultas en Spark y Flink se toma como base el tiempo de ejecución que tardan en un entorno datawarehouse convencional.

El almacén de datos que corre los procesos estudiados es una plataforma de base de datos Oracle 11G sobre un sistema operativo RedHatlinux de 64 bits, corriendo en

dos equipos Blades de 24 núcleos con cinco discos de 2TB y una memoria de 256GB por cada blade.

Utilizando la base de datos descripta en la sección 3 de este capítulo, el tiempo de respuesta aproximado para cada consulta es el siguiente:

- 3 horas para el armado de la tabla *view_precalculated*
- 2 horas para la ejecución de la agregación RU
- 2.5 horas para la ejecución de la agregación RV

8.2 Experimentación con Spark y Flink

Para tener un orden sobre las mediciones y resultados se han realizado dos análisis, el primero es el procesamiento del formato de archivos semi estructurado CSV y el segundo es el procesamiento de los archivos ORC columnares con reducción de dimensiones.

En el primer análisis se procede a realizar pruebas con tres configuraciones de hardware distintas (Tabla 5). Los resultados develan que en el contexto C1 Flink demora 45 minutos en crear la tabla view_precalculated y Spark tarda 49 minutos. Para la segunda configuración con el contexto C2, Flink demora 28 minutos y Spark demora 43 minutos y para en el contexto C3 Flink demora 17 minutos y Spark 59 minutos. En este sentido, se puede afirmar que Flink, además de superar a Spark en los tres contextos de ejecución, mejora su rendimiento a medida que se mejora la configuración de hardware.

Asimismo, se puede observar que en la ejecución de la agregación RU en el contexto C1, Flink demora 33 minutos y Spark 19 minutos, pero cuando se ejecuta con la configuración C3, el tiempo empeora, lo que significa que, para estos datos, el óptimo es la configuración C2 donde finalizó con la tarea luego de 33 minutos (Tabla 9).

De igual forma, vale destacar que en la ejecución de la agregación RV, en la configuración C1 Flink demora 20 minutos y Spark 12 minutos, pero cuando se recurre a la configuración C3 el tiempo empeora para Spark y se mantiene constante para Flink lo que significa que para estos datos el óptimo es la configuración C2 (Tabla 10).

Como conclusión preliminar, se puede decir que en la etapa 1 de este experimento la performance está directamente relacionada con el formato de archivo,

para este caso CSV, cantidad de registros y cantidad de columnas, dando ambos framework una mejor performance en agregaciones que en consultas con filtros comunes donde la agregación siempre es un cálculo pesado para el procesamiento de datos. Siempre a más datos más demora por eso estos framework usan concurrencia para dividir los datos en porciones más pequeñas y poder analizarlas de forma independiente.

8.2.1 Valores medidos por Ganglia y JConsole

De acuerdo a lo expuesto anteriormente se procede ahora a dar los valores de hardware usados por cada operación que el programa realiza y siendo esto medido por Ganglia y JConsole. El indicador más importante es el tiempo de proceso completo, aunque también se mide el uso de disco, CPU, memoria y red durante la prueba. Esto es un valor a considerar porque siendo un framework más rápido que otro ofrece información acerca de cuántos recursos usa y así se puede decidir cuántos procesos pueden convivir en el proceso actual en un ambiente productivo.

Tabla 8. Tiempos de ejecución (en minutos) de Spark y Flink al construir la tabla *view_precalculated*.

	Contexto C1	Contexto C2	Contexto C3
Flink	45	28	17
Spark	49	43	59

Fuente: Elaboración propia. (2021).

Tabla 9. Tiempos de ejecución (en minutos) de la agregación RU llevada a cabo por Spark y por Flink.

	Contexto C1	Contexto C2	Contexto C3
Flink	43	33	37

Spark	38	19	29

Fuente: Elaboración propia. (2021).

Tabla 10. Tiempos de ejecución (en minutos) de la agregación RV llevada a cabo por Spark y por Flink.

	Contexto C1	Contexto C2	Contexto C3
Flink	37	20	20
Spark	37	12	32

Fuente: Elaboración propia. (2021).

En el transcurso total del procesamiento, Flink consumió un promedio de 12,5 % de recursos de CPU, un pico máximo estimado de consumo de 16 % con la configuración C3. Spark obtuvo su mejor comportamiento bajo el contexto C2 con un promedio de 15.2 % y un máximo de 18 % aproximadamente. Respecto del consumo de memoria, Flink utilizó un mínimo de 17.8 GB y un máximo de 21.4 GB. Spark necesitó 17.5 GB como mínimo y 30.5 GB como máximo. El tiempo de carga de Flink rondó los 4.9 minutos de promedio mientras que el de Spark 7.9 minutos.

El consumo promedio de networking de Flink es de 6.6 MB y de Spark 8.8 MB. Se observa que para el framework Flink la mejor configuración de recursos es con la configuración C3. En cambio, para el Framework Spark la mejor configuración se da en la configuración de C2. A través de los resultados obtenidos Flink demuestra tener la mejor respuesta a la resolución de consultas con operadores tipo Joins en relación a Spark.

En cuanto al procesamiento de la agregación RU, Spark obtuvo un consumo promedio de 18 % de recursos de CPU en el transcurso del procesamiento que fue de 19 minutos en total. El mejor comportamiento de Flink se obtuvo bajo un promedio de 20.7 % de consumo de CPU en el transcurso del procesamiento de 33 minutos. Spark utilizó un mínimo de 19.6GB de memoria, un máximo de 37.3GB y un promedio de 36.5GB. En cambio, Flink utilizó 17.5GB como mínimo y 30.5GB como máximo. La

carga de datos de Spark promedió los 3.7 minutos como mínimo y 7.6 como máximo, mientras que Flink promedió los 8.8 minutos alcanzando un máximo de 13.8 min. Para el recurso Networking, Spark consumió un promedio de 11.5 MB, un máximo de 16.1 MB y un mínimo de 3.4 K.

En este sentido, se concluye preliminarmente que Spark responde mejor a la resolución de trabajos con agregaciones que su contraparte Flink, aunque demuestra una mayor penalización de recursos. Cabe destacar que ambos encontraron su mejor performance para la configuración C2.

Durante el procesamiento de la agregación RV el consumo promedio de recursos de CPU de Spark fue del 4.6%, alcanzando un máximo de 16.6% y un mínimo de 9.1%. El consumo óptimo de recursos de CPU de Flink tuvo un promedio de 2.4%. Los indicadores de memoria de Spark mostraron un uso mínimo de 26.6 GB, un máximo de 37.3 GB y un promedio de 26.1 GB. Flink utilizó un mínimo de 22.2 GB, un máximo de 24.7 GB y un promedio de 24.0 GB de memoria. Para la carga de datos Spark requirió un promedio de 2 minutos, con un máximo en 8.4 min y mínimo de 4.5 min. Flink requirió un promedio de 4.5 min, un máximo de 14.4 min. y un mínimo de 0.08 min. Con respecto al recurso de networking, Spark consumió 10.8 MB promedio, 16.1 MB máximo y 3.4 K mínimo. Flink consumió un promedio de 124.8 KB, 2.6 KB de mínimo y 11.4 KB de máximo.

8.3 Análisis de Datos Columnares

Para este análisis se toma el contexto de mejor desempeño durante el análisis anterior, es decir el Contexto C2, reduciendo la dimensionalidad de tablas para tener registros de memoria más pequeños de mayor capacidad de procesamiento y aplicando un formato columnar ORC por tener mejor respuesta a consultas analíticas que el formato orientado a filas.

Ahora bien, tal como se evidencia en las tablas planteadas en esta sección, se permite presentar como resultado que Flink tuvo un consumo promedio del 2% de recurso de CPU en el transcurso del procesamiento de 3 minutos de duración, un máximo 13.5 % y un mínimo de 4.1%. El mejor comportamiento de Spark se obtuvo

bajo un promedio de consumo del 2% en el procesamiento de 4 minutos y un consumo mínimo y máximo de 7.7% y 13.4% respectivamente.

Los indicadores de memoria de Flink mostraron un mínimo de 22.6 GB, un máximo de 35.5 GB y un promedio de 27.8 GB. Spark utilizó un promedio de 28.6 GB, un mínimo de 22.1 GB y un máximo de 38.9 GB de la memoria. Para la carga Flink promedió los 3.7 minutos, con un máximo en 14.2 y un mínimo de 0.04 minutos. Spark cargó en promedio en 3.8 minutos, con un máximo de 14.2 y un mínimo de 0.04 minutos. Con respecto al recurso de networking, Flink consumió un promedio de 157.7 KB y un máximo y mínimo entre 551.7 K y 1.4K. Spark consumió un promedio de 1.5 MB, un mínimo de 1.4 KB con un máximo de 12.4 MB.

Se concluye preliminarmente que ambos responden mejor al procesamiento de agregaciones bajo un modelo columnar, aunque la mejor prestación se identifica con Flink. También se observa que la utilización de este modelo de almacenamiento mejora la administración y optimización de los recursos. Esto se ve reflejado significativamente en ambos frameworks (Tabla 11).

Tabla 11. Tiempos de ejecución (en minutos) de las agregaciones RU y RV con formato columnar llevadas a cabo por Spark y por Flink.

	Agregación RU	Agregación RV
Flink	3	1
Spark	4	4

Fuente: Elaboración propia. (2021).

CAPÍTULO V

1 Conclusiones

Este trabajo aborda la necesidad de indagar en un punto de gran importancia en el área de Big Data referido a la elección idónea del tipo de framework a usar para realizar procesamiento distribuido, teniendo en cuenta que las empresas disponen de data warehouse y otras fuentes de datos que no son de origen estructurado proveniente de redes sociales como Twitter, Facebook y otras.

De este modo se hace manifiesta la necesidad de disponer de frameworks que se adapten y manipulen estos datos sin mucho esfuerzo; la gran cantidad de datos variados hoy impactan en todas las empresas y es necesario contar con herramientas de este tipo, para que luego otros procesos tomen estos resultados y los muestren en tableros (dashboards) y así se puedan tomar decisiones.

A partir de la creación de Apache Spark en 2014, y a medida que se ha venido popularizando, el desarrollo en ambientes de grandes datos, se modifica para siempre; es por ello que actualmente se plantean estos desarrollos como proyectos de software orientados a datos, en los cuales se cuenta con todas las ventajas que trae trabajar con un proyecto de software.

Previo al surgimiento de estas herramientas, el procesamiento se tornaba difícil de construir y mantener y básicamente no se desarrollaban módulos de software extensible, lo cual implicaba un problema para las empresas debido a sus altos costos de mantenimiento y operación.

Al comienzo de este proyecto se formulan objetivos específicos fundamentales que motivaron la realización del trabajo. Luego de la investigación y experimentación realizadas y a partir de los resultados obtenidos de las pruebas, queda en evidencia que de tratarse en un estudio temprano de un proyecto que involucre procesamiento por lotes la herramienta a utilizar debe ser Spark apache Flink, que resulta ser un framework diseñado para procesamiento de Streaming y en donde su API para procesamiento por lotes carece de comunidad activa, hay pocos ejemplos, y poca bibliografía para procesamiento por lotes, pero por el contrario si hay mucho material cuando se habla de procesamiento de Streaming, donde Flink se destaca.

Ambos son realmente rápidos sin lugar a dudas en la resolución de las agregaciones planteadas como problema para la experimentación, y han ganado popularidad por disponer de la flexibilidad para procesar datos de distintos tipos y grandes en un tiempo récord, siendo Apache Spark el destacado, además de brindar la posibilidad de crear aplicaciones que usen su potencial en distintos lenguajes de programación como R, Python, Scala y Java, teniendo también integración nativa con HDFS, YARN, HIVE, HBASE y una infinidad de herramientas nativas de los ecosistemas de llamados de Big Data.

Flink, en cambio, permite crear aplicaciones en Python, Scala, Java, pero la integración con HDFS, YARN, HIVE no es nativa con los servicios mencionados, y si bien la integración con otras herramientas de los llamados ecosistemas de Big Data está creciendo, todavía no existen muchos conectores con esta herramienta.

A través de la experimentación se observa que los mejores tiempos obtenidos para ambas herramientas se alcanzaron al utilizar un storage con un formato columnar, lo que indica que por el tipo de procesamiento que este formato mejora los tiempos en comparación con el formato CSV usado en el primer análisis. En tal sentido, se desprende que el formato del storage es otro punto a optimizar e indica que, dependiendo de la operación a emprender para hacer un formato u otro, ofrecen mejores tiempos de procesamiento.

En la creación de la tabla view_precalculated durante la primera etapa, Apache Flink ha demostrado hacer mejor uso de recursos de cómputo y obtiene mejor tiempo de respuesta que Apache Spark. No obstante, en la creación de las agregaciones, Apache Flink se devela un mejor uso de los recursos de cómputo, pero penaliza en performance, de modo que Apache Spark resulta ser la mejor opción para cálculos de agregaciones. Ambos frameworks demostraron tener su mejor respuesta con la configuración del contexto C2 y Spark tuvo el mejor desempeño.

En la etapa dos, se toma el mejor contexto de la etapa uno, además se reduce a dimensión de las tablas y se convierten los datos a ORC (Formato Columnar) entendiendo que las consultas realizadas bajo este formato responden más eficientemente. Flink logro tener el mejor tiempo respuesta y uso de recursos bajo estas pruebas.

La codificación para Flink resulta ser muy compleja, por la falta de documentación mencionada. Además, se han presentado problemas técnicos muy complejos por esta misma razón. Por otro lado, a la hora de decidir qué framework usar, también se toma en cuenta qué tan complejo es encontrar personal idóneo que pueda integrarse al proyecto de forma rápida, y que disponga de la experiencia y la formación en estos temas.

Como conclusión, se deja que hay una relación directa entre storage y el framework que los procesa. Esto significa que se debe optimizar tanto el framework de procesamiento como los datos a trabajar.

Durante la preparación de la prueba, se observa que en el caso de Apache Flink resulta difícil encontrar ejemplos y material de lectura, como documentación técnica y mejores prácticas basadas en la experiencia de la comunidad, que son necesarias para construcción de soluciones de software que trabaje con dicho framework.

Como trabajo futuro se pretende analizar el comportamiento de este tipo de frameworks en arquitecturas de mayor tamaño con mayor cantidad de nodos y otras variantes en la configuración del clúster. Por otro lado, de cara a futuro, se pretende trabajar con agregaciones y ETLs sobre Streaming viendo si este modelo de procesamiento y el escalamiento de más nodos mejora los tiempos de los procesos y da más información para poder seguir comparando ambos frameworks.

REFERENCIAS BIBLIOGRÁFICAS

Alkatheri, S., Abbas, S.A., Siddiqui, M.A. A Comparative Study of Big Data Frameworks. International Journal of Computer Science and Information Security (IJCSIS), Vol. 17, No. 1. 2019.

Apache Spark References. Disponible en: https://spark.apache.org/docs/latest/sql-reference.html. Información sustraída en abril 2020.

Apache Flink References Documentación de Apache Flink. Disponible en: https://ci.apache.org/projects/flink/flink-docs-stable/ Información Sustraída en Enero 2020.

Apache Hadoop. Guía de Comandos Hadoop. Disponible en: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/CommandsManual.html Información Sustraída en Febrero 2020.

Apache ORC. El almacenamiento en columnas más pequeño y rápido para cargas de trabajo de Hadoop. Format https://orc.apache.org/

Apache Mesos. Almacenamiento en Columnas. http://mesos.apache.org/documentation/latest/ Información Sustraída en enero 2020.

Avro format. Formatos Comprimidos. Disponible en https://avro.apache.org/docs/1.8.1/spec.htm. Información Sustraída en enero 2020.

Bartolini, I. and Patella, M. Comparing Performances of Big Data Stream Processing Plataforma with RAM^{3}S. 25th Italian Symposium on Advanced Database Systems, SEBD, pp. 138-145. 2017a.

Bartolini, I. and Patella, M. A general framework for real-time analysis of massive multimedia streams. Multimed. Syst., pp. 1–16, 2017b.

Borja, M. Comparativa de rendimiento entre algoritmos de Machine Learning sobre plataformas de procesamiento distribuido. Universidad Politécnica de Madrid. Trabajo de grado no publicado. 2017.

Data Flair. Apache Flink vs Apache Spark - Una guía de comparación. Disponible en: https://data-flair.training/blogs/comparison-apache-flink-vs-apache-spark/. Información sustraída en enero 2020.

Dharmesh, K. Apache Mesos Essentials: Build and execute robust and scalable applications using Apache Mesos. Packt Publishing. 2015. ISBN: 978-1783288762.

Deshpande, T. Learning Apache Flink. Packt Publishing Ltd. UK. 2017. ISBN: 978-1786466228.

Fernández, F. (2018). Modularizing Flink Programs to Enable Stream Analytics in IoT Mashup Tools. Tesis de Maestría no Publicada. Universidad Politécnica de Madrid. Recuperado de, http://oa.upm.es/52898/1/TESIS_MASTER_FEDERICO_FERNANDEZ_MORENO.pdf.

Fundación de Software Apache. Architecture of Mesos. Disponible en: http://mesos.apache.org/documentation/latest/architecture/

Fundación Software Apache. Parquet Format https://parquet.apache.org/documentation/latest/

Grover, M. and Malaska, T. Hadoop Application Architectures: Designing Real-World Big Data Applications. O'Reilly Media. 2015. ISBN: 978-1491900086.

Hernández, E; Duque, N y Moreno, J. Big Data: Una explicación de investigaciones, tecnologías y casos de aplicación. Revista Tecnologías. Vol. 20 (39). Instituto Tecnológico Metropolitano. 2017. Recuperado de, https://doi.org/10.22430/22565337.685

Hurtado, J. El Proyecto de Investigación. Caracas: Quirón. 2015. ISBN: 980-6306-06-6.

Hueske, F. and Kalavri, V. Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications. O'Reilly Media, Inc. 2019. ISBN: 978-1491974292.

Inoubli, W., Aridhi, S., Mezni, H., Maddouri, M. and Nguifo, E. M. An experimental survey on big data frameworks. Future Gener. Comput. Syst., 2018.

Karakaya, Z., Yazici, A. and Alayyoub, M. A Comparison of Stream Processing Frameworks. International Conference in Computer and Applications (ICCA), pp. 1–12. 2017.

Karau, H. and Warren, R. High Performance Spark: Best practices for scaling and optimizing Apache Spark. O'Reilly Media, Inc. 2017. ISBN: 978-1491943205.

Macías Lloret, M., Gómez Parada, M., Tous Liesa, R., Torres Viñals, J. Introducción a Apache Spark. Editorial UOC, S.L. 2016. ISBN: 978-8491160373.

Marcu, O.-C., Costan, A., Antoniu, G., Pérez, M.S. Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks. The IEEE International Conference on Cluster Computing, Taipei, Taiwan. Sep 2016.

Monleón, A. El impacto del Big Data en la sociedad de la información: significado y utilidad. Revista Historia y comunicación social. Vol. 20 (2), pp 427-445. 2015. Recuperado de, http://dx.doi.org/10.5209/rev_HICS.2015.v20.n2.51392.

Perera, S., Perera, A. and Hakimzadeh, K. Reproducible experiments for comparing apache flink and apache spark on public clouds. 2016. https://arxiv.org/abs/1610.04493.

Pereira Villazón, Tatiana, Portilla Manjón, Idoia, & Rodríguez Salcedo, Natalia. Big data y Relaciones Públicas: Una revisión bibliográfica del estado de la cuestión. Revista de Comunicación, 18(1), 151-165. 2019. https://dx.doi.org/10.26441/RC18.1-2019-A8

Ryza, S., Laserson, U., Owen, S. & Wills, J. Advanced Analytics with Spark: Patterns for Learning from Data at Scale. O'Reilly Media, Inc. 2017. ISBN: 978-1491972953.

Saxena, S. and Gupta, S., Practical Real-time Data Processing and Analytics: Distributed Computing and Event Processing using Apache Spark, Flink, Storm, and Kafka. Packt Publishing Ltd. 2017. ISBN: 9781787281202.

Stitchdata. Base de Datos en Columnas. Disponible en: https://www.columnardatabase.com/ Informacion Sustraída en enero 2020.

Veiga, J., Expósito, R. R., Pardo, X. C., Taboada, G. L. and Tourifio, J. Performance evaluation of big data frameworks for large-scale data analytics. IEEE International Conference on Big Data. pp. 424–431. 2016.

White, T. Hadoop: The Definitive Guide: Storage and Analysis. O'Reilly Media. 2015. ISBN: 978-1491901632.

Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I. Spark: Cluster Computing with Working Sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10). USENIX Association, USA, 2010.