Estudio Comparativo entre Apache Spark y Apache Flink en el procesamiento de streaming en entornos Big Data

Trabajo Final presentado para obtener el grado de Especialista en Inteligencia de Datos Orientada a Big Data

Autor

Hugo Manuel Fajardo

Director:

Dr. Lic. Waldo Hasperué

Facultad de Informática

Universidad Nacional de La Plata

Octubre de 2022

Estudio Comparativo entre Apache Spark y Apache Flink en el procesamiento de streaming en entornos Big Data

Resumen:

La sociedad hoy plantea crecientes demandas de soluciones informáticas, cuando estas soluciones requieren el procesamiento de grandes volúmenes de datos, las herramientas tradicionales de procesamiento muestran limitaciones e inconvenientes derivados de la cantidad de datos a procesar o del tiempo necesario para realizarlo. Surge así, la necesidad de herramientas específicas, llamadas herramientas de Big Data. Dentro de estas existe un grupo concreto para el procesamiento de flujos de datos (stream processing), entendiendo por flujo de datos la recepción y procesamiento continuo de datos ilimitados desde diferentes fuentes. Debido a su naturaleza sin límite, estos flujos no pueden descargarse de manera completa, y deben ser procesados en línea cuando se reciben. Dos de las principales herramientas para el procesamiento de streaming son Apache Spark y Apache Flink.

El objetivo del presente trabajo es realizar una comparación entre Apache Spark y Apache Flink en el procesamiento de streaming. Para realizar la comparación entre estas herramientas se utilizará el lenguaje de desarrollo Python, ya que el mismo soporta el trabajo tanto en Spark como en Flink, y a su vez es uno de los lenguajes de programación más utilizados en la actualidad.

La comparación entre los frameworks requiere el desarrollo de dos aplicaciones para el tratamiento del flujo de datos, ambas resolviendo el mismo problema. Una aplicación realizará el procesamiento de streaming en Apache Spark, mientras que la otra realizará la misma tarea en Apache Flink.

Palabras clave: Streaming de Datos, Procesamiento de Flujos, Procesamiento Distribuido de Flujos de Datos, Apache Spark, Apache Flink, Apache Kafka.

Comparative study between Apache Spark and Apache Flink in the processing of data streams in Big Data environments

Abstract:

Society today poses increasing demands for computer solutions, when these solutions require the processing of large volumes of data, traditional processing tools show limitations and drawbacks derived from the amount of data to be processed or the time required to do so. Thus arises the need for specific tools, called Big Data tools. Within these there is a specific group for the processing of data flows (stream processing), understanding by data flow the reception and continuous processing of unlimited data from different sources. Due to their unlimited nature, these streams cannot be fully downloaded, and must be processed online upon receipt. Two of the main tools for streaming processing are Apache Spark and Apache Flink.

The objective of this work is to make a comparison between Apache Spark and Apache Flink in streaming processing. To make the comparison between these tools, the Python development language will be used, since it supports work in both Spark and Flink, and in turn is one of the most used programming languages today.

The comparison between the frameworks requires the development of two applications for the treatment of the data flow, both solving the same problem. One application will perform the streaming processing on Apache Spark, while the other will perform the same task on Apache Flink.

Keywords: Data Streaming, Stream Processing, Distributed Data Stream Processing, Apache Spark, Apache Flink, Apache Kafka.

Índice

Índice de Ilustraciones

1. Introducción

La sociedad en la actualidad tiene crecientes demandas de soluciones informáticas, cuando estas soluciones requieren el procesamiento de grandes volúmenes de datos las herramientas tradicionales de procesamiento de datos comienzan a tener limitaciones y mostrar inconvenientes, derivados de la cantidad de datos a procesar o del tiempo necesario para hacerlo. A mayor volumen de datos a procesar, mayores son los inconvenientes y retos que se plantean. Surge entonces la necesidad de nuevas herramientas informáticas orientadas específicamente al procesamiento de gran cantidad de datos, las cuales de manera general se denominaron herramientas de Big Data.

Dentro de las herramientas de Big Data existe un grupo específico para el procesamiento de streaming, entendiendo por streaming a la transmisión y procesamiento de flujos continuos de datos que pueden procesarse sin obtenerse o descargarse de manera completa. Estos datos pueden estar en diversos formatos: texto, audio o video, entre otros. La adopción de tecnologías de procesamiento de streaming está impulsada por la creciente necesidad de las empresas de mejorar el tiempo necesario para reaccionar y adaptarse a los cambios en su entorno operativo. Dentro de las principales herramientas para el procesamiento de streaming encontramos a Apache Spark y a Apache Flink, estas herramientas serán el objeto de estudio del presente trabajo.

1.1. Motivación

Existe una realidad en el mundo del procesamiento de datos y es que la cantidad de dispositivos que producen información aumenta exponencialmente. Es decir, crece constantemente la cantidad de dispositivos con capacidad informática en nuestros entornos personales y profesionales, entre ellos: televisores, automóviles conectados, teléfonos inteligentes, computadoras para bicicletas, relojes inteligentes, cámaras de vigilancia, termostatos, etc. En la actualidad la gente se rodea de dispositivos destinados a producir registros de eventos, estos son flujos de mensajes que representan las acciones e incidentes que forman parte de la historia de un dispositivo en su contexto. A medida que se interconectan cada vez más dispositivos, se requieren mayores capacidades para analizar y procesar los registros de eventos generados. Este fenómeno abre la puerta a una increíble explosión de creatividad e innovación en el dominio del análisis de datos casi en tiempo real, con la condición de que se encuentre una manera de hacer que este análisis sea manejable. En este mundo de dispositivos y registros de eventos agregados, el procesamiento de streaming ofrece una forma amigable con los recursos para facilitar el análisis de flujos de datos.

Otro motivo que impulsa la adopción de tecnologías de streaming es la creciente necesidad de las empresas de mejorar el tiempo necesario para reaccionar y adaptarse a los cambios en su entorno operativo. Esta forma de procesar los datos a medida que ingresan proporciona una ventaja técnica y estratégica. Ejemplos de esta adopción continua incluyen sectores como el comercio electrónico, transmisiones de datos continuos creados por empresas que interactúan con los clientes las 24 horas del día, los 7 días de la semana, o compañías de tarjetas de crédito, que analizan las transacciones a medida que ocurren para detectar y detener actividades fraudulentas [5].

Los datos recolectados por infinidad de dispositivos deben procesarse de forma secuencial y gradual, registro por registro o en ventanas de tiempo graduales, y se utilizan para una amplia variedad de tipos de análisis, como correlaciones, agregaciones, filtrado y muestreo. La información derivada del análisis aporta a las empresas visibilidad de numerosos aspectos del negocio y de las actividades de los clientes, como el uso de servicios (para la

medición/facturación), la actividad del servidor, los clics en un sitio web y la ubicación geográfica de dispositivos, personas y mercancías, y les permite responder con rapidez ante cualquier situación que surja. Por ejemplo, las empresas pueden hacer un seguimiento de los cambios en la opinión del público sobre sus marcas y productos mediante el análisis continuo de las transmisiones de las redes sociales, y responder de manera oportuna a medida que sea necesario [13].

1.1.1. Ventajas del procesamiento de streaming

El procesamiento de streaming resulta beneficioso en la mayoría de las situaciones en las que se generan datos nuevos y dinámicos de forma continua. Es apto para la mayoría de los sectores y casos de uso de Big Data. Por lo general, las empresas comienzan con aplicaciones sencillas, que realizan recopilación de registros, e implementan cálculos de valores medios, mínimos y máximos. Más adelante, estas aplicaciones evolucionan y se pasa a un procesamiento más sofisticado, casi en tiempo real. En un principio, las aplicaciones pueden procesar transmisiones de datos para producir informes básicos y realizar acciones sencillas como respuesta, por ejemplo, emitir alertas cuando las medidas clave superan ciertos umbrales. Con el tiempo, dichas aplicaciones realizan un análisis de datos más sofisticado, como la aplicación de algoritmos de aprendizaje automático y la extracción de información más exhaustiva a partir de los datos. Posteriormente, se aplican complejos algoritmos de procesamiento de transmisiones y eventos, como intervalos de tiempo decrecientes para encontrar, por ejemplo, las películas populares más recientes, lo que enriquece aún más la información.

Si bien es creciente la adopción de tecnologías de streaming en todos los ámbitos, se mencionan algunos a modo de ejemplo [13]:

- Los sensores de vehículos de transporte, equipo industrial y maquinaria agrícola que envían datos que permiten detectar cualquier posible defecto de forma anticipada y envían el pedido de un recambio automáticamente, lo que evita el tiempo de inactividad del equipo.
- Una institución financiera que controla los cambios en la bolsa de valores en tiempo real, procesa el valor en riesgo y modifica las carteras automáticamente en función de los cambios en los precios de las acciones.
- Un sitio web inmobiliario controla un subconjunto de datos de los dispositivos móviles de los clientes y realiza recomendaciones en tiempo real acerca de los inmuebles que deben visitar en función de su ubicación geográfica.
- Una empresa editora de medios transmite miles de millones de registros de clics de sus propiedades en línea, acumula los datos y los enriquece con información demográfica sobre los usuarios, y optimiza la ubicación del contenido en su sitio, proporcionando así una experiencia mejor y más relevante a su público.
- Una compañía de juegos en línea recopila datos de las interacciones de los jugadores con el juego y los envía a su plataforma de juegos. Luego, analiza los datos en tiempo real y ofrece incentivos y experiencias dinámicas para involucrar a los jugadores.

1.1.2. Herramientas de Procesamiento de Streaming

Apache Spark es un motor de análisis para el procesamiento de datos a gran escala. Fue diseñado pensando en la rapidez y proporciona una Api de alto nivel en Java, Scala, Python y R, y un motor optimizado que admite grafos de ejecución general. También es compatible con un amplio conjunto de herramientas de alto nivel que incluyen Spark SQL para SQL y procesamiento de datos estructurados, MLlib para aprendizaje automático, GraphX para procesamiento de grafos y

Structured Streaming para procesamiento de streaming. Spark es una herramienta de propósito general desarrollada específicamente para el procesamiento distribuido por lotes que con el tiempo se adaptó y optimizó para el procesamiento de streaming. El enfoque de Spark para el procesamiento de streaming consiste en dividir un lote en pequeños microlotes y procesarlos de manera continua de manera rápida, escalable, tolerante a fallas y de extremo a extremo exactamente una vez sin que el usuario tenga que razonar sobre la transmisión subyacente.

Flink es un motor de procesamiento distribuido diseñado y construido específicamente para el trabajo con flujos de datos ilimitados y acotados. Ha sido diseñado para ejecutarse en todos los entornos de cluster comunes, realizar cálculos a la velocidad de la memoria y a cualquier escala. Proporciona un motor de transmisión de alto rendimiento y baja latencia permitiendo el procesamiento de eventos de tiempo y administración de estado.

Otra característica muy importante de Flink es que brinda tolerancia a fallos y garantiza que cada dato sea procesado exactamente una vez. La interacción con el motor puede realizarse desde lenguajes como Java, Scala, Python y SQL. Los programas Flink se ejecutan como un sistema distribuido dentro de un cluster y se pueden implementar en modo independiente, así como en configuraciones basadas en YARN, Mesos, Docker junto con otros marcos de administración de recursos.

1.2. Objetivos y Metodología

El objetivo del presente trabajo es realizar una comparación entre los dos frameworks previamente mencionados. El estudio comparativo se realizará desde varios puntos de vista, entre ellos:

- Facilidad de instalación y despliegue.
- Fuentes admitidas para el intercambio de datos.
- Lenguajes de programación soportados.
- Documentación disponible.
- Evaluación de rendimiento.

Para realizar la comparación entre estas herramientas se utilizará el lenguaje de desarrollo Python, ya que el mismo soporta el trabajo tanto en Spark como en Flink, y a su vez es uno de los lenguajes de programación más utilizados en la actualidad.

La comparación entre los frameworks requiere el desarrollo de dos aplicaciones para el tratamiento del flujo de datos, ambas resolviendo el mismo problema. Una aplicación realizará el procesamiento del flujo de datos en Apache Spark, mientras que la otra realizará la misma tarea en Apache Flink. El problema puntual será la implementación de una técnica de minería de datos, mientras que el streaming será construido de manera sintética simulando un flujo real.

Una versión preliminar del presente trabajo fue presentado y aceptado en el Congreso Argentino de Ciencias de la Computación CACIC 2022, dicho trabajo[12] será publicado en los próximos días, cuando se realice dicho congreso.

El presente trabajo cubrirá en el capítulo 2, el desarrollo de los fundamentos del procesamiento de streaming, que sirven de base para los dos frameworks utilizados. En el capítulo 3 se cubre Apache Spark, y se enfoca en el procesamiento de streaming en esta herramienta. En el capítulo 4 se desarrolla Apache Flink como motor de procesamiento de streaming. En el capítulo 5 se describe el trabajo de experimentación para la comparación de ambos frameworks de

procesamiento de streaming. EL capítulo 6 describe en detalle las pruebas realizadas y por último, en el capítulo 7 se incluyen las conclusiones.

2. Fundamentos del Procesamiento de Streaming

En este capítulo se desarrollan los conceptos, ideas y estrategias que sirven de fundamento al procesamiento de flujos continuos de datos. Inicialmente se trata los datos de streaming, luego se compara el procesamiento de streaming con el procesamiento tradicional por lotes. Posteriormente se explicita el modelo de procesamiento de streaming, que brinda el marco teórico para la comprensión y análisis de flujos continuos de datos. Finalmente se describen los conceptos a tener en cuenta cuando se realiza procesamiento de streaming.

2.1. Datos de streaming

Actualmente se vive en un mundo globalizado, altamente conectado donde muchas de las acciones que día a día se realizan implican intercambio de información, desde comunicación con seres queridos, ver series o películas hasta incluso comprar alimentos. En esta realidad diaria hay un aspecto que muchas veces no se tiene en cuenta y que sirve de base para que cada persona pueda satisfacer sus necesidades, la tecnología de intercambio de información subyacente. Esta brinda la plataforma sobre la que servicios de procesamiento de datos están construidas. Una de ellas es el procesamiento de streams o traducido al español procesamiento de flujos.

Se define por streaming a la transmisión y procesamiento de flujos continuos de datos que pueden procesarse sin obtenerse o descargarse de manera completa. Cuando se habla de flujos continuos de datos implica que no se está en presencia de archivos con un inicio y un fin como podría ser un archivo de audio, texto o video, por el contrario, se trata de flujos de datos ilimitados. Estos flujos de datos sin límite tienen la particularidad de que deben poder obtenerse y procesarse a medida que se reciben. En caso de existir algún tipo de falla en la recepción de los datos se procede a reintentar el acceso a los mismos.

Para explicar dicho concepto podemos pensar en una gran tubería que transporta agua, por su interior fluyen miles y miles de litros. Si en algún momento se utiliza un mecanismo que permite extraer el líquido que fluye por la tubería y depositarlo en un repositorio, lo que finalmente se obtiene es agua. Esto nos permite afirmar que a pasar de que lo extraído no es la totalidad del líquido que fluye por la tubería este no pierde sus propiedades, por lo que finalmente se obtiene agua.

Esta manera de obtener y procesar datos se ha extendido masivamente en el mundo globalizado y hoy se puede acceder multitud de servicios de streaming de todo tipo de contenido, desde películas, música, textos e información financiera.

El autor Tyler Aridau [6] por su parte define los datos de streaming como un tipo de conjunto de datos que es infinito en tamaño (al menos teóricamente). Debido a que los sistemas de información se componen de hardware con recursos finitos, como la memoria y la capacidad de almacenamiento, no es posible que contengan conjuntos de datos de longitud indefinida. En su lugar, los datos se reciben en el sistema de procesamiento en forma de un flujo de eventos a lo largo del tiempo. A esto se lo llama flujo de datos.

2.2. Procesamiento por lotes y procesamiento de streaming

En términos de Big Data existen dos maneras de realizar procesamiento de datos:

- Procesamiento por lotes.
- Procesamiento de streaming en tiempo real.

Con procesamiento por lotes, nos referimos al análisis computacional de conjuntos de datos limitados, es decir acotados. En términos prácticos, esto significa que esos conjuntos de datos están disponibles y se pueden recuperar en su totalidad desde algún tipo de almacenamiento. Implica que se conoce el tamaño del conjunto de datos al comienzo del proceso computacional y la duración de ese proceso está limitada en el tiempo. La ilustración 1 ilustra el procesamiento por lotes. En la misma se observa que para realizar el procesamiento de los datos se debe primero acceder al medio de almacenamiento que contiene los datos.

Ilustración 1 – Procesamiento por Lotes

Por el contrario, con el procesamiento de flujos nos preocupamos por el procesamiento de los datos a medida que llegan al sistema. Dada la naturaleza ilimitada de los flujos de datos, los procesadores de flujos deben ejecutarse de manera constante mientras la transmisión siga entregando nuevos datos. Al estar en presencia de datos potencialmente sin límite, se puede decir que podríamos estar recibiendo datos por siempre. La ilustración 2 refleja lo antes mencionado.

Ilustración 2 – Procesamiento de Streams

Los sistemas de procesamiento de flujos deben aplicar técnicas operativas y de programación diferenciales para hacer posible el procesamiento de flujos de datos potencialmente infinitos con una cantidad limitada de recursos informáticos.

2.3. Procesamiento distribuido de streaming

Cuando la cantidad de datos a procesar es muy grande, la capacidad física de los equipos de cómputo termina condicionando el procesamiento de los mismos, surge la necesidad de añadir nuevos equipos con la finalidad de distribuir el poder de procesamiento y de cómputo.

La tarea de añadir nuevos equipos de procesamiento para distribuir la carga de trabajo en una serie de ejecutores permite dividir el flujo de entrada en particiones, donde cada ejecutor puede ver una vista parcial de la secuencia completa. En la ilustración 3 se muestra que uno de los nodos debe cumplir el rol de coordinador, será quien reciba las solicitudes y gestionará el trabajo coordinado de todos los nodos ejecutores. En una arquitectura distribuida, en caso de que la carga de trabajo sea alta pueden agregarse nuevos ejecutores, para que la capacidad de procesamiento total del sistema no sea sobrepasada.

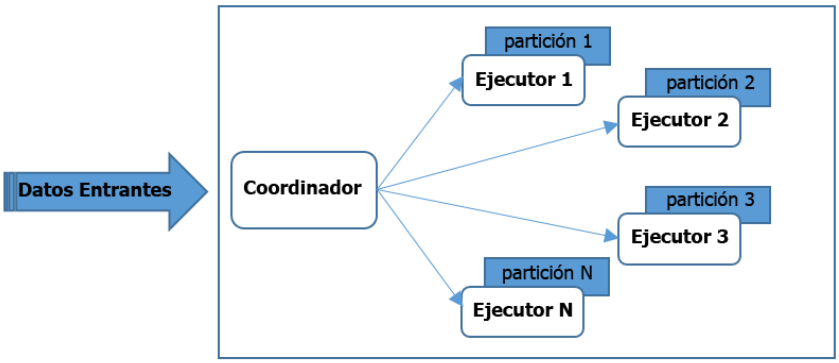

Ilustración 3 – Arquitectura de Procesamiento Distribuido de Streams

El modelo distribuido de procesamiento de streaming soluciona el inconveniente derivado de contar con hardware de procesamiento limitado a la vez que plantea un desafío, proporcionar una abstracción que oculte al usuario la complejidad subyacente, permitiendo que quien interactúe con el sistema pueda razonar sobre el flujo como un todo.

La tolerancia a fallas es otro aspecto a gestionar en un sistema distribuido y esto aumenta la complejidad del mismo, ya que situaciones como falta de conectividad, caída de nodos, necesidad de realizar tareas de mantenimiento de hardware o software pueden presentarse y necesitan ser consideradas.

2.4. Modelo de Procesamiento de Streaming

Para entender como funciona un sistema de procesamiento de flujos es necesario presentar un modelo que permita describir la manera de recibir datos en forma de streaming, manipular los datos recibidos y entregar resultados. Es importante destacar que se trata de procesamiento de streaming con estado, un tipo de procesamiento de flujo que requiere la contabilidad de cálculos anteriores en forma de algún estado intermedio necesario para procesar nuevos datos. Llamamos a esto Modelo de Procesamiento de Streaming.

Se puede pensar en un flujo de datos o datos de streaming como una fuente de datos en movimiento. Para desarrollar un modelo de comprensión de datos en movimiento se utilizarán tres conceptos:

- Fuentes de datos.
- Tubería de procesamiento.
- Sumideros de datos.

Una fuente de datos (data source) de streaming es un origen de datos que permite el acceso a un flujo continuo e ilimitado de datos. En el contexto del procesamiento de flujos, acceder a los datos de un flujo a menudo se denomina consumir el flujo. Esta abstracción se presenta como una interfaz que permite la implementación de instancias destinadas a conectarse a sistemas específicos, algunos ejemplos son: Apache Kafka, Flume, Twitter, un socket TCP, etc.

La tubería de procesamiento de datos representa específicamente el trabajo sobre el flujo continuo e ilimitado de datos. Este trabajo puede ser realizado de diversas formas, aunque varias herramientas de procesamiento de streaming adoptan el enfoque de programación funcional para declarar transformaciones y agregaciones que operan sobre los datos, asumiendo que estos son flujos inmutables.

Recibe el nombre de sumidero de datos la abstracción utilizada para escribir flujos de datos salientes. Estos datos salientes representan resultados finales del procesamiento a los datos de streaming. Los frameworks de procesamiento de flujos proporcionan conectores a varios sistemas específicos.

La noción de fuentes y sumideros representa los límites del sistema de procesamiento de flujos, y en forma general se conoce como tubería de streaming (streaming pipeline). La ilustración 4 muestra la imagen de una tubería para ayudar a entender el modelo y también permite comprender que en sistemas complejos las tuberías pueden interconectarse. En este caso, el sumidero de una tubería puede ser fuente de datos para otro sistema. Este encadenamiento se conoce con el término de *Canalización*.

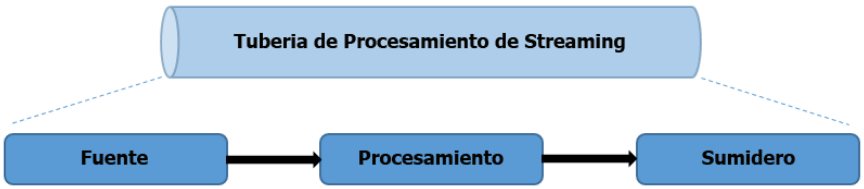

Ilustración 4 – Modelo de Procesamiento de Streaming

El hecho de utilizar programación funcional para representar el procesamiento de los datos permite a las herramientas de procesamiento operar sobre datos inmutables, utilizando transformaciones para expresar cómo procesar el contenido de un flujo para obtener un flujo de datos derivado. Esto asegura que, en cualquier punto dado de un programa, cualquier flujo de datos pueda rastrearse hasta sus entradas mediante una secuencia de transformaciones y operaciones que se declaran explícitamente en el programa. Como consecuencia, cualquier proceso particular puede reconstituir el contenido del flujo de datos utilizando solo el programa y los datos de entrada, lo que hace que el cálculo sea inequívoco y reproducible.

2.5. Procesamiento continuo vs. procesamiento en microlotes

El hecho de que los trabajos de streaming se procesan en tiempo real significa que los resultados intermedios deben proporcionarse al consumidor de esa transmisión de forma regular, es decir nos gustaría que esos resultados observables fueran coherentes, en línea y en tiempo real con respecto a la llegada de datos. Esto quiere decir que se busca obtener resultados exactos y lo antes posible. Si se tiene en cuenta la manera de procesar los datos entrantes, los frameworks de procesamiento de streaming se agrupan en dos tipos, los de procesamiento continuo y los de procesamiento en microlotes.

En los sistemas basados en procesamiento continuo, cada nodo del sistema escucha continuamente los mensajes de otros nodos y envía nuevas actualizaciones a sus nodos secundarios. Por ejemplo, suponga que su aplicación implementa un cálculo *Map-Reduce* en varios flujos de entrada. En un sistema de procesamiento continuo, cada uno de los nodos que implementan el *Map* leería los registros uno por uno de una fuente de entrada, calcularía su función en ellos y los enviaría al reductor apropiado. El reductor luego actualizará su estado cada vez que obtenga un nuevo registro. La idea clave es que esto suceda en cada registro individual. Esta manera de procesar el flujo de datos entrante también suele conocerse con el nombre de "procesamiento de un registro a la vez" y se muestra en la ilustración 5.

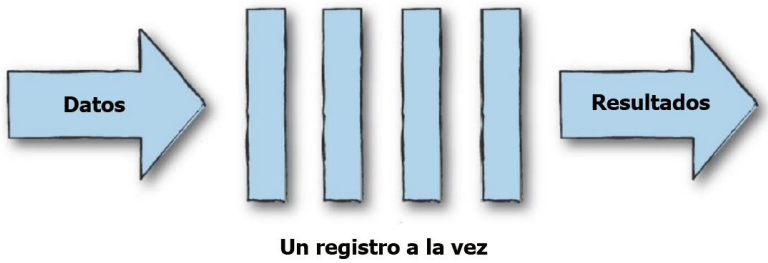

Ilustración 5 – Procesamiento Continuo

En los sistemas de procesamiento de streaming se emplea el término latencia para referir al tiempo necesario para que el sistema reaccione ante la llegada de un evento en particular. La latencia mínima, es muy diferente entre las dos maneras de procesar streaming, la latencia mínima del sistema de microlotes es, por lo tanto, el tiempo necesario para completar la recepción del microlote actual (el intervalo de lote) más el tiempo necesario para iniciar una tarea en el ejecutor donde caen los datos (también llamado tiempo de programación). Por otro lado, un sistema que procesa registros uno por uno puede reaccionar tan pronto como se encuentre con el evento de interés.

El procesamiento continuo tiene la ventaja de ofrecer la latencia más baja posible cuando la tasa de entrada total es relativamente baja, porque cada nodo responde inmediatamente a un nuevo mensaje.

Sin embargo, los sistemas de procesamiento continuo generalmente tienen un rendimiento máximo más bajo, porque incurren en una cantidad significativa de sobrecarga por registro (por ejemplo, llamar al sistema operativo para enviar un paquete a un nodo descendente). Además, los sistemas continuos generalmente tienen una topología fija de operadores que no se pueden mover en tiempo de ejecución sin detener todo el sistema, lo que puede generar problemas de equilibrio de carga.

Los sistemas de microlotes, a diferencia de los continuos, esperan para acumular pequeños lotes de datos de entrada (por ejemplo, 500 ms), luego procesan cada microlote en paralelo usando una colección distribuida de tareas, similar a la ejecución de un trabajo por lotes. Los sistemas de microlotes a menudo pueden lograr un alto rendimiento por nodo porque aprovechan las mismas optimizaciones que los sistemas por lotes (por ejemplo, procesamiento vectorizado) y no incurren en ninguna sobrecarga adicional por cada registro o elemento recibido.

En la ilustración 6 se observa como los registros individuales que se reciben se agrupan en microlotes para poder ser procesados.

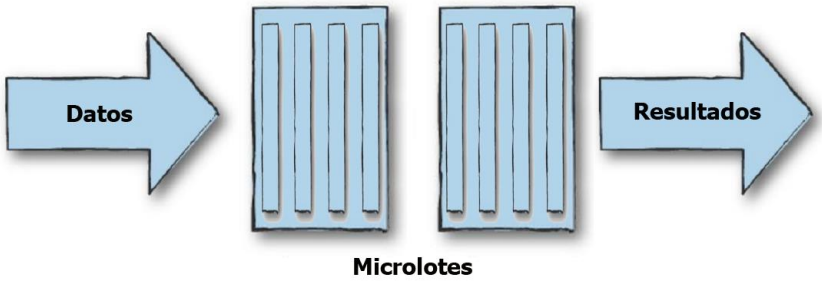

Ilustración 6 – Procesamiento en Microlotes

Los sistemas que implementan microlotes requieren menos nodos para procesar la misma tasa de datos. Estos sistemas también pueden usar técnicas de balanceo de carga dinámico para manejar cargas de trabajo variables en el tiempo (p. ej., aumentar o disminuir la cantidad de tareas). Sin embargo, la desventaja es una latencia base más alta debido a la espera para acumular un microlote. En la práctica, las aplicaciones de transmisión que son lo suficientemente grandes como para distribuir su computación tienden a priorizar el rendimiento sobre la latencia.

Al elegir entre estos dos modos de ejecución, los factores principales que se deben tener en cuenta son la latencia deseada y el costo total de operación (TCO). Los sistemas de microlotes pueden ofrecer cómodamente latencias de 100 ms a un segundo, según la aplicación. Dentro de este régimen, generalmente requerirán menos nodos para lograr el mismo rendimiento y, por lo tanto, un menor costo operativo (incluido un menor costo de mantenimiento debido a fallas de nodos menos frecuentes). Para latencias mucho más bajas, se debe considerar un sistema de procesamiento continuo.

2.6. Tiempo de evento y tiempo de procesamiento

De manera general los datos pueden encontrarse:

- En reposo, en forma de archivo, contenido en una base de datos o algún otro tipo de registro.
- En movimiento, como secuencias de señales generadas continuamente, como la medición de un sensor o señales de GPS de vehículos en movimiento.

Desde el punto de vista de una línea de tiempo, los datos en reposo son datos del pasado, podría decirse que todos los conjuntos de datos delimitados, ya sea almacenados en archivos o contenidos en bases de datos, fueron inicialmente un flujo de datos recopilados a lo largo del tiempo en algún almacenamiento. La base de datos de usuarios, los pedidos del último trimestre, las coordenadas GPS de los viajes en taxi en una ciudad, etc., todo comenzó como eventos individuales recopilados en un repositorio.

Para hablar de tiempo respecto a datos de streaming o en movimiento se plantea un desafío. No alcanza con considerar el tiempo en el que se produce un evento, es necesario considerar también el momento en que dicho evento llega al sistema para ser procesado. Esto plantea que puede existir una diferencia de tiempo entre el momento en que los datos se generan en su origen y cuando los mismos están disponibles para su procesamiento. Ese delta de tiempo puede ser muy corto, como cuando se trata de eventos de registro web generados y procesados dentro del mismo centro de datos, o mucho más largo, como cuando se trata de datos de GPS de un automóvil que viaja a través de un túnel y este solo envía los mismos cuando el vehículo restablece su conectividad inalámbrica después de salir del túnel.

Estos dos instantes de tiempo diferentes plantean la necesidad de considerar una línea de tiempo cuando los eventos son producidos y otra línea de tiempo diferente cuando los mismos son manejados por el sistema de procesamiento de flujos. Estas líneas de tiempo son tan significativas que cada una tiene nombres específicos:

- Hora del evento: representa la hora en la que se creó el evento de acuerdo al reloj local del dispositivo que genera el evento.
- Hora de procesamiento: representa la hora en que el evento llega y es procesado por el sistema de procesamiento de flujos.

La ilustración 7 permite observar ambos instantes de tiempo. La diferenciación entre estas dos líneas de tiempo se vuelve muy importante cuando se necesita correlacionar, ordenar o agregar los eventos con respecto a otros eventos.

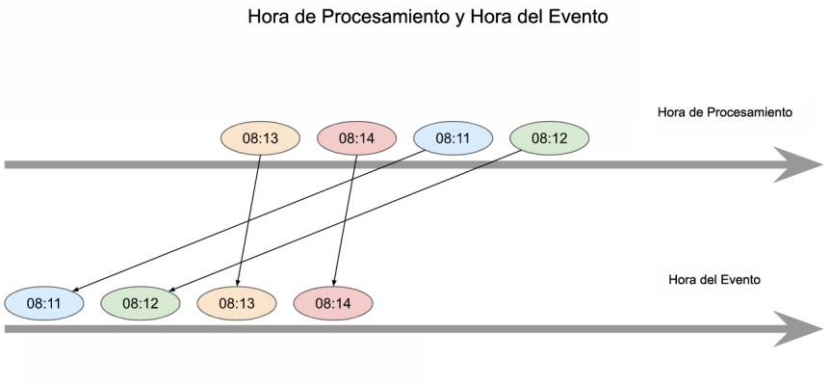

Ilustración 7 – Hora de Procesamiento y Hora del Evento

Otro desafío importante vinculado con el tiempo es la llegada de los eventos. En general, los sistemas de streaming no requieren que la entrada se produzca a intervalos regulares, todos a la vez, o siguiendo un ritmo determinado. Debido a que la computación tiene un costo en recursos, es un desafío predecir la carga máxima para hacer coincidir la llegada repentina de elementos de entrada con los recursos informáticos necesarios para procesarlos. Esto introduce un factor de incertidumbre respecto al tiempo en que llegarán los eventos en el futuro y la determinación de las capacidades de cómputo necesarias para procesar los mismos.

Si se cuenta con la capacidad informática necesaria para hacer frente a una afluencia repentina de elementos de entrada, nuestro sistema producirá los resultados esperados, pero si no se ha planificado tal explosión de datos de entrada, algunos sistemas de transmisión podrían sufrir retrasos, restricciones de recursos o incluso fallas. Lidiar con la incertidumbre es un aspecto

importante del procesamiento de flujos que necesita tenerse en cuenta y gestionarse para garantizar la obtención de los resultados esperados.

2.7. Procesamiento de eventos con marcas de agua

Como hemos notado, el procesamiento de flujo produce resultados periódicos a partir del análisis de los eventos recibidos. Cuando se está equipado con la capacidad de procesar la hora de los eventos, el sistema de procesamiento de streaming puede agrupar esos mensajes en dos categorías, una de acuerdo a la hora de los eventos y otra de acuerdo a la hora de procesamiento de los mismos.

Esta situación nos plantea nuevos interrogantes:

- ¿Qué ocurre si un dispositivo por alguna situación no puede enviar mensajes?
- ¿Qué ocurre si un dispositivo por alguna situación tiene retraso en el envío de mensajes?

Esto obliga a pensar en ¿cómo debe responder a esto el sistema de procesamiento de streaming?

Una solución posible es establecer un límite de tiempo de espera para eventos retrasados. Este límite de tiempo recibe el nombre de marca de agua (watermark). La marca de agua es, en un momento dado, la marca de tiempo más antigua que aceptaremos en el flujo de datos. Cualquier evento que sea más antiguo que esta expectativa no se incluye en los resultados del procesamiento de flujo.

Considerar marcas de agua nos obliga a pensar los resultados obtenidos como provisionales, es decir como no definitivos, ya que la llegada tardía de eventos puede afectar valores intermedios que terminen alterando resultados. Poder contar con la posibilidad de establecer marcas de agua tiene consecuencias en el sistema de procesamiento, ya que será necesario almacenar gran cantidad de resultados intermedios y consumir una cantidad significativa de memoria que corresponde aproximadamente a *longitud de la marca de agua* × *la velocidad de llegada* × *el tamaño del mensaje*.

Además, dado que se debe esperar a que caduque la marca de agua para estar seguros de que se tienen todos los elementos que componen un intervalo. Los procesos de transmisión que usan marca de agua y desean tener un resultado final único para cada intervalo, deben retrasar su salida por al menos la longitud de la marca de agua. Una vez cumplido este tiempo ya puede liberarse la memoria requerida por la computación con marca de agua.

Establecer el tiempo “correcto” de marca de agua es una tarea que debe ser cuidadosamente realizada. Una marca de agua demasiado pequeña hará que se eliminen demasiados eventos y producirá resultados incompletos. Una marca de agua demasiado grande retrasará la salida de los resultados considerados completos durante demasiado tiempo y aumentará las necesidades de recursos del sistema de procesamiento de streaming para preservar todos los eventos intermedios.

2.8. Manipulación de Datos

Los frameworks de procesamiento de streaming adoptan el enfoque de programación funcional para procesar datos, declaran las transformaciones y agregaciones que operan en los flujos de datos, asumiendo que dichos flujos son inmutables.

2.8.1. Transformaciones

Debido a que no es posible mutar elementos emplean transformaciones para expresar cómo procesar el contenido de un flujo para obtener un flujo de datos derivado. Esto asegura que cualquier flujo de datos pueda rastrearse hasta sus entradas mediante la secuencia de transformaciones y operaciones aplicadas. Como consecuencia, se puede reconstituir el contenido del flujo de datos utilizando solo el código del programa y los datos de entrada, lo que hace que el cálculo sea inequívoco, trazable y reproducible.

Las transformaciones son cálculos que se expresan de la misma manera para cada elemento del flujo. Por ejemplo, crear un flujo derivado que duplique cada elemento de su flujo de entrada corresponde a una transformación.

Dicho de otra manera, las transformaciones expresan dependencias estrechas (narrow) ya que, para producir un elemento de la salida, solo se necesita un único elemento de entrada.

Mientras que las agregaciones representan dependencias amplias (wide), donde para producir un elemento de salida es necesario observar muchos elementos del flujo de entrada.

2.8.2. Agregaciones

Las agregaciones son acciones que tienen por objetivo resumir los datos obteniendo valores totales que representan al flujo de datos recibido o a una parte de los mismos. Esto puede implicar contabilizar cada uno de los elementos que componen el flujo de entrada o un grupo de ellos.

Un ejemplo de agregación es recopilar los cinco números más grandes de un flujo de entrada. Otro ejemplo puede ser calcular el valor promedio de alguna lectura cada 10 minutos.

En un sistema por lotes las agregaciones representan al total de datos procesados, mientras que en un sistema de streaming las agregaciones se utilizan para representar totales parciales bajo determinados criterios como por ejemplo por tiempos (en la última hora, en los últimos minutos).

2.8.2.1. Agregaciones de Ventana

Los sistemas de procesamiento de streaming se alimentan de eventos que ocurren en tiempo real, como ejemplos de esto podemos mencionar los mensajes de redes sociales, los clics en páginas web, las transacciones de comercio electrónico, los eventos financieros o las lecturas de sensores. Las aplicaciones de procesamiento de streaming a menudo tienen una vista centralizada de los registros de varios lugares, ya sean tiendas minoristas o simplemente servidores web para una aplicación común. Aunque ver cada transacción individualmente puede no ser útil o incluso práctico, podríamos estar interesado en ver las propiedades de los eventos vistos durante un período de tiempo reciente; por ejemplo, los últimos 15 minutos o la última hora, o tal vez incluso ambos. A medida que estos eventos continúan apareciendo, los más antiguos generalmente se vuelven cada vez menos relevantes para cualquier procesamiento que esté tratando de lograr.

Encontramos muchas aplicaciones que emplean agregaciones temporales regulares y recurrentes. Este tipo de agregaciones reciben el nombre de agregaciones ventana y pueden ser de ventanas estáticas o de ventanas deslizantes.

2.8.2.2. Ventanas Estáticas

La noción más natural de una agregación de ventana es una función de agrupación cada x período de tiempo. Por ejemplo, *la temperatura ambiente máxima y mínima por cada hora* o *el consumo total de energía eléctrica cada 30 minutos*. Estos son ejemplos de agregaciones de ventanas.

En los ejemplos se observa cómo los períodos de tiempo son inherentemente consecutivos y no se superponen. Cuando se trata de agregación donde cada periodo de tiempo es fijo y sigue al anterior sin superponerse, se denomina agregación de ventanas estáticas.

Las agregaciones de ventana estática se utilizan cuando se necesita producir agregados de datos durante períodos regulares de tiempo, con cada período independiente de los períodos anteriores. Otra denominación que tienen este tipo de agregaciones es de ventana giratoria. La ilustración 8 grafica un ejemplo de ventana estática donde se realiza agregación de datos con un intervalo regular de 10 minutos.

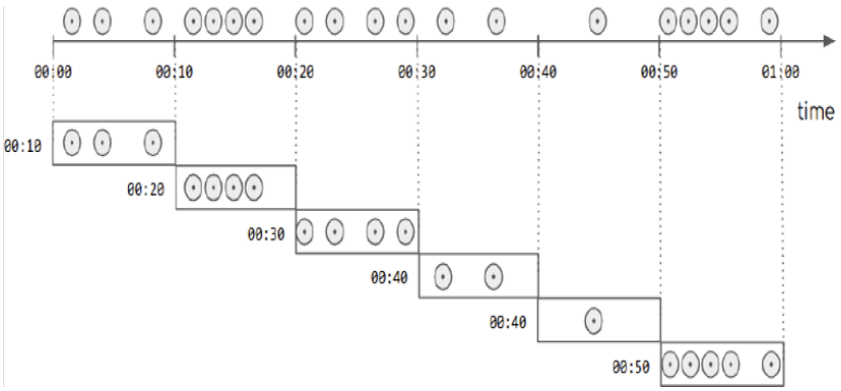

Ilustración 8 – Ejemplo de Ventana Estática

2.8.2.3. Ventanas Deslizantes

Las ventanas deslizantes o móviles son agregados durante un período de tiempo que se informa con una frecuencia más alta que el período de agregación en sí. Como tal, las ventanas deslizantes se refieren a una agregación con dos especificaciones de tiempo, la duración de la ventana y la frecuencia de informe. Por lo general, se lee como *una función de agrupación durante un intervalo de tiempo x informado cada frecuencia y*. Por ejemplo, *el precio promedio de las acciones durante el último día reportado cada hora*. Como se puede notar, esta combinación de una ventana deslizante con la función de promedio es la forma más conocida de una ventana deslizante, comúnmente conocida como promedio móvil.

A continuación, en la ilustración 9 se muestra una ventana deslizante con un tamaño de ventana de 30 segundos y una frecuencia de informe de 10 segundos. En la ilustración 9 se puede observar una característica importante de las ventanas deslizantes, las mismas no se definen por periodos de tiempo menores al tamaño de la ventana. Podemos ver que no hay ventanas reportadas para el tiempo 00:10 y 00:20.

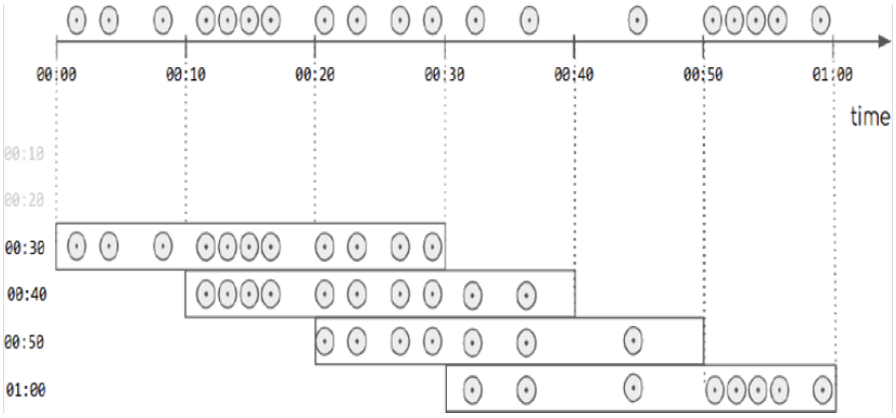

Ilustración 9 – Ejemplo de Ventana Deslizante

La ilustración 9 muestra además una característica interesante, se puede construir y mantener una ventana deslizante agregando los datos más recientes y eliminando los elementos vencidos, manteniendo todos los demás elementos en su lugar.

2.9. Streaming sin estado y streaming con estado

El procesamiento de flujos sin estado es aquel donde se recibe un evento, se procesa y la respuesta al mismo no depende de otros elementos recibidos con anterioridad. Este tipo de flujo que requiere una respuesta aislada no es el único tipo, ya que con frecuencia se busca reaccionar en tiempo real a nuevos elementos basándose en un análisis que depende de todo el flujo, como la detección de valores atípicos en una colección o en base al cálculo de estadísticas agregadas recientes a partir de datos de eventos. Por ejemplo, podría ser interesante encontrar patrones de vibración más altos de lo habitual en un flujo de lecturas de un motor de avión, lo que requiere comprender las mediciones de vibración regulares para el tipo de motor que se está utilizando.

Los casos donde se busca comprender datos nuevos en el contexto de los datos ya vistos, a menudo nos lleva al procesamiento de flujo con estado. El procesamiento de flujo con estado es el enfoque mediante el cual se calcula algo a partir de los nuevos elementos observados en los datos de entrada y se actualiza los datos internos que ayudan a conseguir ese cómputo.

El procesamiento de flujo con estado se refiere a cualquier procesamiento de flujo que emplea información pasada para obtener su resultado. En este tipo de flujo es necesario mantener cierta información de estado en el proceso de cálculo del siguiente elemento de la secuencia.

El procesamiento con estado es más costoso en términos de los recursos que utiliza y también presenta desafíos ante situaciones de falla, esto lleva a pensar ¿qué sucede si nuestra computación falla a la mitad del flujo? Aunque una regla general segura es elegir la opción sin estado, si está disponible, muchas de las preguntas interesantes que podemos hacer sobre un flujo de datos suelen ser de naturaleza con estado. Por ejemplo: ¿cuánto tiempo estuvo la sesión del usuario en un determinado sitio? ¿cuál fue el camino que usó un taxi a través de una ciudad? ¿cuál es el promedio móvil del sensor de presión en una máquina industrial?

Desde el punto de vista de los recursos utilizados, el procesamiento con estado requiere mayores recursos como memoria y tiempo de procesador para llevar a cabo los cómputos.

La gestión de estados también trae consecuencias respecto al mecanismo de tolerancia a fallas, ya que, para poder ofrecer recuperación en situaciones de falla, el motor de procesamiento debe

guardar resultados obtenidos de estado, y posteriormente gestionarlos para hacer uso de los mismos en situaciones de falla.

Se mencionó que el procesamiento de estado implica uso adicional de recursos como memoria y tiempo de procesador, de la misma manera, proporcionar tolerancia a fallas también tiene su consecuencia, mayor uso de recursos de cómputo entre ellos tiempo de procesador, cantidad de memoria y almacenamiento utilizado.

3. Apache Spark

En este capítulo se describe el framework Apache Spark como motor distribuido de procesamiento de datos a gran escala y en particular como herramienta para el procesamiento de streaming.

3.1. Introducción

Apache Spark se inició como un proyecto de investigación en la universidad de Berkeley en el año 2009 en el AMPlab. Al año siguiente se publicó en el paper "Spark: Cluster Computing with Working Sets" de Matei Zaharia, Mosharaf Chowdhury, Michael Franklin, Scott Shenker e Ion Stoica [11].

La primera versión de Spark solo admitía aplicaciones por lotes, pero pronto su uso incluyó la ciencia de datos interactiva y las consultas ad hoc. Al conectar el intérprete de Scala a Spark, el proyecto pudo proporcionar un sistema interactivo muy útil para ejecutar consultas en cientos de máquinas.

Con el lanzamiento de esta versión quedó claro que los agregados más potentes a Spark serían las nuevas bibliotecas, por lo que el proyecto comenzó a seguir el enfoque de librería estándar que tiene hoy. En particular, diferentes grupos de AMPlab iniciaron *MLlib*, *Spark Streaming* y *GraphX*. También se aseguró que estas Api sean altamente interoperables, lo que permitió escribir aplicaciones de Big Data completas, end to end en el mismo motor por primera vez.

En 2013, el proyecto había crecido e incorporó más de 100 colaboradores de más de 30 organizaciones externas a Berkeley. El AMPlab contribuyó con Spark a *Apache Software Foundation* ese mismo año. El primer equipo también lanzó una empresa, Databricks, para fortalecer el proyecto, y se unió a la comunidad de empresas y organizaciones que contribuyen a Spark. En el año 2014 se elevó a Spark como Top Level Project.

Hacia 2015, ya tenía más de 1000 colaboradores convirtiéndose en uno de los proyectos más activos de la *Apache Software Foundation* y en uno de los proyectos de Big Data de código abierto más activos.

En 2016 se introdujo un nuevo motor de streaming de alto nivel, *Structured Streaming*. Esta tecnología es la manera en que las empresas resuelven sus desafíos de datos a gran escala, desde compañías tecnológicas como Uber y Netflix que utilizan las herramientas de streaming y aprendizaje automático, hasta instituciones como la NASA, el CERN y el Instituto Broad del MIT y Harvard aplican Spark en análisis de datos científicos.

Apache Spark es un motor de computación unificado y un conjunto de librerías para el procesamiento de datos en paralelo en clusters de computadoras. Se basa en *Hadoop MapReduce* y amplía el modelo *MapReduce* para usarlo de manera eficiente para más tipos de cálculos, que incluyen consultas interactivas y procesamiento de streaming. La característica principal de Spark es su computación en cluster en memoria que aumenta la velocidad de procesamiento permitiendo conseguir resultados en menor cantidad de tiempo que *Hadoop MapReduce.*

Spark es uno de los motores de código abierto más activamente desarrollado para esta tarea, esto lo convierte en una herramienta estándar para cualquier desarrollador o científico de datos interesado en Big Data. Admite múltiples lenguajes de programación, e incluye librerías para diversas tareas que van desde procesamiento por comandos SQL hasta transmisión de flujos y

aprendizaje automático, y se ejecuta en cualquier lugar, desde una computadora portátil hasta un cluster de miles de servidores. Esta versatilidad lo convierte en un sistema fácil de usar para comenzar, y brinda la posibilidad de crecer a gran escala.

La ilustración 10 resume el conjunto de herramientas que Apache Spark proporciona.

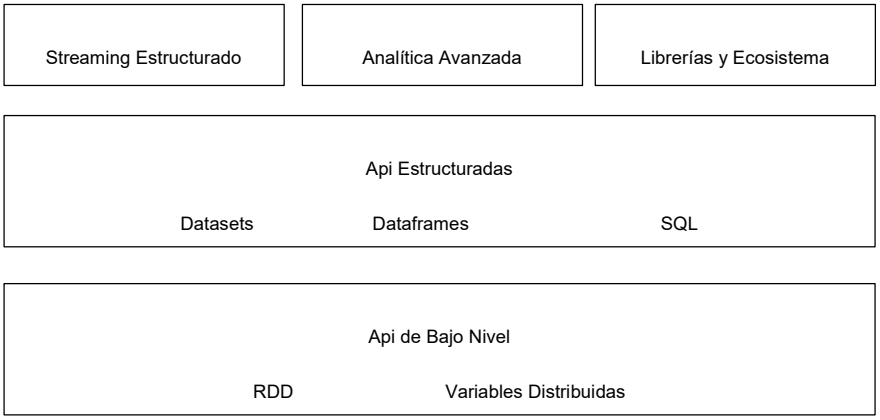

Ilustración 10 – Caja de Herramientas de Apache Spark[3]

El objetivo principal de Spark es ofrecer una plataforma unificada para escribir aplicaciones de Big Data. Está diseñado para admitir una amplia gama de tareas de análisis de datos, que van desde la simple carga de datos y consultas SQL hasta el aprendizaje automático y el cómputo de streaming, sobre el mismo motor de procesamiento y con un conjunto uniforme de Api. Las tareas de análisis de datos del mundo real, ya sean análisis interactivos o desarrollo de software tradicional para aplicaciones de producción, tienden a combinar muchos tipos de procesamiento y librerías diferentes, la idea central de Spark es reunir todo esto en una sola plataforma, integrando Api generales y ejecución de alto rendimiento.

Apache Spark es un motor de cómputo y se encarga de manejar la carga de datos de los sistemas de almacenamiento y realizar cálculos en ellos, pero el almacenamiento permanente no es su finalidad. Se puede usar Spark con una amplia variedad de sistemas de almacenamiento persistente, incluidos los sistemas de almacenamiento en la nube como *Azure Storage* y *Amazon S3*, sistemas de archivos distribuidos como *Apache Hadoop*, almacenes de clave-valor como *Apache Cassandra* y sistemas de mensajería como *Apache Kafka*. La idea central es que la mayoría de los datos ya residen en una combinación de sistemas de almacenamiento. Los datos son costosos de mover, por lo que Spark se enfoca en realizar cálculos sobre los datos, sin importar dónde residan.

Una parte importante de Spark son sus librerías, proporciona librerías estándar y permite integrar paquetes de terceros desarrollados por las comunidades de código abierto. Hoy en día, las librerías estándar son en realidad la mayor parte del proyecto de código abierto, el motor central ha cambiado poco desde que se lanzó por primera vez, pero las librerías han crecido para proporcionar más y más funcionalidad. Incluye funcionalidad para procesamiento SQL y datos estructurados (*Spark SQL*), aprendizaje automático (*MLlib*), procesamiento de streaming y análisis de grafos (*GraphX*). Más allá de lo que proporciona Spark, hay cientos de librerías externas de código abierto que van desde conectores para varios sistemas de almacenamiento hasta algoritmos de aprendizaje automático.

3.2. Arquitectura de Spark

De manera cotidiana utilizamos computadoras y estas funcionan bien para finalidades diversas como ver películas o trabajar con procesadores de texto o planillas de cálculo. Sin embargo, hay tareas de procesamiento específicas para las cuales una sola computadora no es lo suficientemente poderosa para realizarlo, o el tiempo requerido para hacerlo será demasiado extenso.

Un área particularmente desafiante es el procesamiento de datos a gran escala. Las máquinas individuales, las que habitualmente utilizamos en casa o en el trabajo no tienen suficiente potencia y recursos para realizar cálculos que involucren procesar grandes volúmenes de datos. Para poder manejar grandes volúmenes de procesamiento es necesario modificar este escenario tradicional.

Un cluster o grupo de computadoras reúne los recursos de muchas máquinas, lo que nos brinda la capacidad de utilizar todos los recursos acumulados como si fueran una sola gran computadora. Esto se logra dividiendo la totalidad de datos a procesar en la cantidad de computadoras del cluster o grupo. La consecuencia de esto es que se incrementa el poder de procesamiento ya que varias máquinas pueden trabajar en paralelo sobre porciones de datos y así conseguir obtener los resultados en menor cantidad de tiempo.

Ahora, un cluster de máquinas por sí solo no es poderoso, necesita un framework para coordinar el trabajo entre ellas. Spark hace exactamente eso, administrar y coordinar la ejecución de tareas de datos en un grupo de computadoras.

Para poder llevar a cabo su propósito Spark está organizado en componentes. Los componentes principales son:

- *Spark Core* es el componente principal. En él se encuentra la funcionalidad necesaria para la ejecución de trabajos y sirve de cimiento para los demás componentes. La abstracción central que proporciona es el núcleo, y es conocida como *RDD* (*Resilient Distributed Datasets* o conjunto de datos resilientes). Un *RDD*, según *Spark*, se define como una colección de elementos con la capacidad de ser tolerante a fallas y de ser capaz de operar en paralelo. Es importante enfatizar que operar en paralelo es clave y forma parte de la filosofía de *Apache Spark*. Por ello contiene la lógica para acceder al sistema de archivo distribuido. Otras responsabilidades que incluye el núcleo son gestión de redes, la seguridad, programación de trabajos, transferencia de datos y el api de lenguajes que permite escribir código Spark en varios lenguajes de programación.
- *Spark SQL* es un componente que proporciona funciones para manipular grandes conjuntos de datos distribuidos y estructurados utilizando un subconjunto de *lenguaje SQL,* soportado por Spark y *Hive SQL* (*HiveQL*). Presenta una nueva abstracción de datos llamada *SchemaRDD*, que brinda soporte para datos estructurados y semiestructurados. Por medio de *DataFrame* y *Datasets* simplifica el manejo de datos estructurados y permite realizar optimizaciones para mejorar el rendimiento. Por estas razones, este componente se ha convertido en uno de los más importantes y utilizados de Spark. Las operaciones sobre *DataFrames* y *DataSets* para ser procesadas por *Spark Core* se traducen a operaciones *RDDs* y de esta manera se ejecutan como trabajos normales de Spark.
- *Spark Streaming* proporciona la funcionalidad necesaria para el procesamiento de streaming. Es necesario aclarar que Spark ha sido diseñado e implementado para el procesamiento por lotes, por lo que para procesar flujos de datos en tiempo real divide el flujo de datos entrante en flujos discretizados o micro lotes llamados *DStreams*. Para ser procesados por *Spark Core* los micro lotes serán traducidos a *RDDs* y de esta manera será

tratado como un trabajo normal. También permite acceder a datos de diversas fuentes, entre ellas *HDFS*, *Kafka*, *Flume* entre otras. Una capacidad valiosa de *Spark Streaming* es brindar tolerancia a fallas, lo cual es muy importante para el procesamiento de datos en tiempo real.

- *Mlib* brinda la posibilidad de contar con aprendizaje automático a gran escala. Proporciona una librería integrada de algoritmos de machine learning. Su objetivo es hacer que el aprendizaje automático sea práctico y escalable. Este componente permite el preprocesamiento, la manipulación, el entrenamiento de modelos y la realización de predicciones a escala sobre los datos. Spark proporciona una Api de aprendizaje automático sofisticada para realizar una variedad de tareas de machine learning, desde la clasificación hasta la regresión, incluyendo también agrupación en clusters hasta el aprendizaje profundo (deep learning).
- *GraphX* es el componente para grafos y computación paralela de grafos. En un nivel alto, *GraphX* amplía *Spark RDD* al presentar una nueva abstracción de Graph, un multigrafo dirigido con propiedades adjuntas a cada vértice y borde. Para admitir el cálculo de grafos, *GraphX* expone un conjunto de operadores fundamentales (p. ej., subgraph, joinVertices y addedMessages), así como una variante optimizada de la Api de Pregel. Además, incluye una colección cada vez mayor de algoritmos y constructores de grafos para simplificar las tareas de análisis de los mismos.

La ilustración 11 muestra los componentes de Spark y como se estructuran sobre el núcleo.

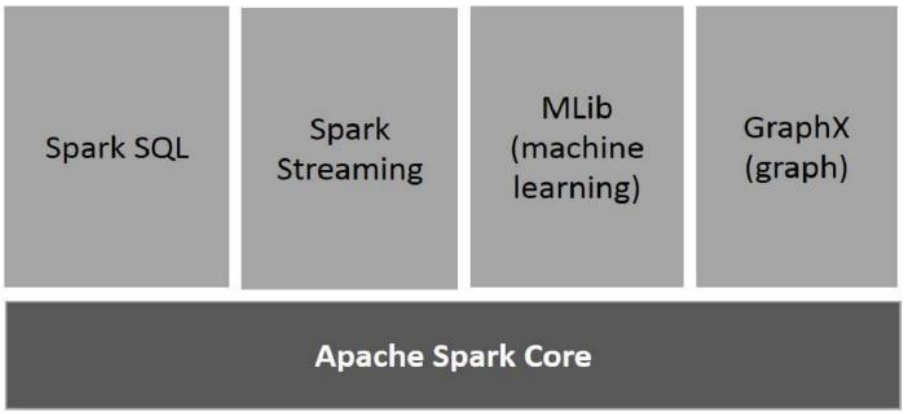

Ilustración 11 – Componentes de Apache Spark

Los componentes que se sitúan sobre *Spark Core* constituyen lo que se conoce como las librerías de Spark. Si bien estas librerías son las principales, hoy existen cientos de otras externas de código abierto que van desde conectores para varios sistemas de almacenamiento hasta algoritmos de aprendizaje automático.

3.3. Despliegue de Spark

En el mundo del Big Data existen protagonistas que tienen bastante trayectoria, entre ellos se encuentra el ecosistema *Hadoop*. Spark está destinado a mejorar, no a reemplazar, la pila de *Hadoop*. Desde el primer día, Spark se diseñó para leer y escribir datos desde y hacia *HDFS*, así como otros sistemas de almacenamiento, como *HBase* y *S3* de Amazon. Como tal, los usuarios de *Hadoop* pueden enriquecer sus capacidades de procesamiento al combinar *Spark* con *Hadoop MapReduce*, *HBase* y otros frameworks de Big Data.

Hay tres formas de implementar *Spark* en un cluster de *Hadoop*:

- Implementación Independiente.
- Implementación *Hadoop Yarn*.
- Implementación en *MapReduce* (SIMR).

La implementación independiente de Spark significa que Spark está en la parte superior de *HDFS* (Sistema de archivos distribuido de Hadoop). Aquí, Spark y *MapReduce* se ejecutarán en paralelo para cubrir todos los trabajos de Spark en el cluster. Con esta implementación, se pueden asignar recursos de forma estática en todas las máquinas o en un subconjunto de ellas en un cluster de Hadoop y ejecutar Spark junto con *Hadoop MapReduce*. Luego, el usuario puede ejecutar trabajos de Spark arbitrarios en sus datos *HDFS*. Su simplicidad hace que esta sea la implementación elegida por muchos usuarios de *Hadoop* 1.x.

En *Hadoop Yarn* los usuarios de Hadoop que ya implementaron o planean implementar *Hadoop Yarn* pueden simplemente ejecutar Spark en Yarn sin necesidad de preinstalación o acceso administrativo. Esto permite a los usuarios integrar fácilmente Spark en su pila de Hadoop y aprovechar toda la potencia de Spark, así como otros componentes que se ejecutan sobre Spark. Este tipo de implementación ayuda a integrar Spark en el ecosistema de Hadoop o en la pila de Hadoop.

La implementación en *MapReduce* (SIMR) es para los usuarios de Hadoop que aún no ejecutan YARN, es usar SIMR para iniciar trabajos de Spark dentro de *MapReduce*. Con SIMR, los usuarios pueden comenzar a experimentar con Spark y usar su shell en un par de minutos después de descargarlo. Esto reduce enormemente la barrera de implementación y permite interactuar con Spark de manera rápida.

Para resumir las alternativas de despliegue de Spark se adjunta la ilustración 12. Allí se puede observar cada una de las tres alternativas disponibles.

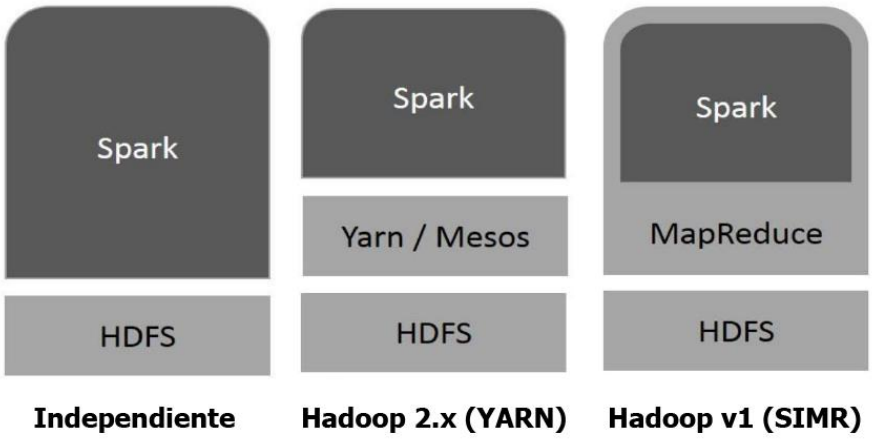

Ilustración 12 – Implementaciones de Apache Spark

Spark puede interactuar no solo con Hadoop, sino también con otras tecnologías populares de Big Data.

Entre ellas se encuentran:

- *Apache Hive*: a través de Shark, Spark permite a los usuarios de Hive ejecutar sus consultas mucho más rápido. Hive es una solución popular de almacenamiento de datos que se ejecuta sobre Hadoop, mientras que Shark es un sistema que permite que el marco Hive se ejecute sobre Spark en lugar de Hadoop.
- *AWS EC2*: los usuarios pueden ejecutar fácilmente Spark (y Shark) sobre EC2 de Amazon, ya sea utilizando los scripts que vienen con Spark o las versiones alojadas de Spark y Shark en Elastic MapReduce de Amazon.
- *Apache Mesos*: Spark se ejecuta sobre Mesos, un sistema de administración de clusters que proporciona un aislamiento de recursos eficiente en todas las aplicaciones distribuidas, incluidas MPI y Hadoop. Mesos habilita el uso compartido detallado, lo que permite que un trabajo de Spark aproveche dinámicamente los recursos inactivos en el cluster durante su ejecución. Esto conduce a mejoras de rendimiento considerables, especialmente para trabajos de ejecución prolongada.

3.4. Api de Lenguajes de Spark

Apache Spark permite que se ejecute código Spark utilizando diversos lenguajes de programación. Esta funcionalidad es provista por la Api de lenguajes de Spark, la misma se encarga de traducir código de diferentes lenguajes a código nativo de Spark. Si usa solamente la Api estructurada, puede esperar rendimientos similares en todos los lenguajes de programación. A continuación, se resumen los lenguajes soportados por la Api:

- *Scala*: el framework Spark está escrito principalmente en *Scala*, lo que lo convierte en el lenguaje "predeterminado" de Spark.
- *Java*: la implementación actual de *Scala* se ejecuta en la máquina virtual de *Java* y es compatible con aplicaciones *Java* existentes. Esto garantiza que pueda escribir código Spark en *Java*.
- *Python*: es uno de los lenguajes de programación de más amplia difusión en la actualidad, a la vez que admite casi todas las construcciones que admite *Scala*.
- *SQL*: el framework Spark admite un subconjunto del estándar ANSI SQL 2003. Esto facilita que los analistas, los no programadores y los especialistas en base de datos aprovechen las capacidades Big Data de Spark.
- *R*: los científicos de datos también tienen su acceso preferencial a Spark. Spark tiene dos librerías de *R* de uso común: una como parte del núcleo de Spark (*SparkR*) y otra como un paquete impulsado por la comunidad de *R* (sparklyr).

Cada lenguaje de la Api mantiene los mismos conceptos básicos. Hay un objeto *SparkSession* disponible para el usuario, que es el punto de entrada para ejecutar código Spark. Cuando se usa Spark desde *Python* o *R*, no se escriben instrucciones *JVM* explícitas; en su lugar, se escribe código *Python* y *R* que Spark traduce en código que luego puede ejecutar en las *JVM* del ejecutor.

3.5. Abstracciones de Datos

Para realizar cálculos paralelos y distribuidos, *Apache Spark* brinda varias alternativas para manejar grupos de datos. Utilizar la abstracción de datos correcta es fundamental para acelerar la ejecución de trabajos de Spark y aprovechar las optimizaciones internas que brinda. Además, elegir una buena estructura de datos acelera el proceso de desarrollo.

Una partición es un conjunto de filas que se encuentra en una máquina física del cluster. Este concepto es vital para poder conseguir distribuir datos entre varias máquinas y de esta manera conseguir procesarlos en forma distribuida y en paralelo, por lo tanto, las particiones representan cómo los datos se distribuyen físicamente en el grupo de máquinas durante la ejecución. Es oportuno mencionar que para conseguir paralelismo es necesario dividir los datos en varias particiones como así también contar con varios nodos ejecutores, esto permite que cada ejecutor pueda procesar parte de los datos y el cómputo general avance en paralelo.

Las abstracciones que Spark proporciona pueden ser clasificadas en abstracciones de alto nivel y de bajo nivel. También son conocidas como Api de bajo nivel y Api de alto nivel o estructuradas, respectivamente. Las abstracciones de alto nivel son más sencillas de implementar y de optimizar, por ello, los desarrolladores de Spark recomiendan utilizarlas en casi todos los escenarios.

Por lo general, se recurre a las Api de nivel inferior cuando se necesita alguna funcionalidad que no es provista en las Api de nivel superior, o en situaciones puntuales como cuando se debe mantener una base de código heredada escrita en bajo nivel. Incluso si es un desarrollador avanzado que espera sacar el máximo provecho de Spark, se recomienda utilizar la Api estructurada por las optimizaciones que pueden aplicarse.

Clasificadas en alto y bajo nivel, las abstracciones que Spark brinda son:

- Alto nivel:
    - DataFrames.
    - Tablas SQL.
    - Datasets.
- Bajo nivel:
    - RDDs.

Todas estas diferentes abstracciones representan colecciones distribuidas de datos y serán descriptas en las siguientes secciones.

3.5.1. DataFrames

Tradicionalmente una manera natural de organizar datos es en forma de tabla, formada por filas y columnas. Esta forma de representar información es seguida por Spark para presentar una de sus principales abstracciones de datos, los DataFrames.

Un *DataFrame* representa los datos en forma de filas y columnas, y tiene además una estructura, conocida con el nombre de *Esquema*, este se compone por cada una de sus columnas y su tipo de dato correspondiente.

Hasta aquí no hay nada nuevo, incluso podríamos comparar un *DataFrame* con una planilla de cálculo, y hasta el momento no habría diferencia alguna. Pero existe una pregunta crucial que nos muestra la diferencia: ¿qué ocurre cuando los datos a representar exceden la capacidad de un solo computador? En este escenario es evidente que se necesita una manera distribuida de representar, acceder y procesar dichos datos.

Un *DataFrame* es una abstracción distribuida de datos en forma de filas y columnas, que puede ocupar más de una computadora. La razón para poner los datos en más de una computadora puede ser, por ejmplo, cuando los datos son demasiado grandes para caber en una sola máquina o simplemente cuando llevaría demasiado tiempo realizar cálculos en un único computador.

Los DataFrames en Spark son estructuras de datos inmutables por lo que no pueden ser cambiados una vez que son creados. El trabajo con DataFrames no consiste en manipularlos, sino que se trata de indicar transformaciónes y/o acciones de alto nivel a realizar, y Spark determinará la mejor manera de realizar ese trabajo en el cluster.

Para representar los datos Spark emplea un lenguaje de programación propio. Internamente, Spark usa un motor llamado *Catalyst* que mantiene su propia información de tipo a través de la planificación y el procesamiento de trabajos. Esto abre una amplia variedad de optimizaciones de ejecución que marcan diferencias de rendimiento significativas. Los tipos de Spark se asignan directamente a las diferentes Api de lenguaje que Spark mantiene y existe una tabla de búsqueda para cada uno de estos en *Scala*, *Java*, *Python*, *SQL* y *R*. Incluso si se usa la Api estructurada de Spark de *Python*, la mayoría de nuestras manipulaciones operarán estrictamente en los tipos Spark optimizados, no en los tipos de *Python*.

Los DataFrames son conjuntos de datos de tipo fila. El tipo "Fila" es la representación interna de Spark de su formato de memoria optimizado. Este formato permite un cómputo altamente especializado y eficiente porque, en lugar de usar tipos de *JVM*, que pueden generar altos costos de recolección de elementos no utilizados y creación de instancias de objetos, Spark puede operar en su propio formato interno sin incurrir en ninguno de esos costos. Para aprovechar las ventajas de la representación interna optimizada, Spark recomienda el uso de DataFrames.

3.5.2. SQL

SQL es un lenguaje para expresar operaciones relacionales sobre datos. Se utiliza en todas las bases de datos relacionales, e incluso bases de datos "NoSQL" crean su dialecto *SQL* para facilitar el trabajo con sus bases de datos. *SQL* está en todas partes, y aunque los expertos en tecnología profetizaron su muerte, es una herramienta de datos extremadamente resistente de la que dependen muchas empresas. Spark implementa un subconjunto de *ANSI SQL:2003*.

Spark SQL es posiblemente una de las características más importantes y poderosas de Spark. Con *Spark SQL* puede ejecutar consultas *SQL* en vistas o tablas organizadas en bases de datos. También puede usar funciones del sistema o definir funciones de usuario y analizar planes de consulta para optimizar sus cargas de trabajo.

La abstracción de más alto nivel en *Spark SQL* es el *Catálogo*. El *Catálogo* es una abstracción para el almacenamiento de metadatos sobre los datos almacenados en sus tablas, así como también para funciones y vistas. El *Catálogo* es la interfaz programática para *Spark SQL*, está disponible en el paquete *org.apache.spark.sql.catalog.Catalog* y contiene las funciones útiles para hacer cosas como listar tablas, bases de datos y funciones.

Una *Tabla SQL* es equivalente a un *DataFrame* en el sentido de que ambas son una estructura de datos sobre la que es posible operar. Tienen en común que cada una de ellas posee una determinada estructura que las define. Se puede definir tablas, filtrarlas, unirlas y realizar otras manipulaciones. La principal diferencia entre las tablas y los dataframes es que un *DataFrame* se definen en el ámbito de un lenguaje de programación, mientras que las tablas se definen adentro de una base de datos.

En Spark 2.X, las tablas siempre contienen datos. No existe la noción de una tabla temporal, solo una vista, que no contiene datos. Esto es importante porque si va a eliminar una tabla, puede correr el riesgo de perder los datos al hacerlo.

Las tablas almacenan dos piezas importantes de información. Los datos dentro de las mismas, y también los datos sobre ellas; es decir, sus metadatos. De la misma manera que en una base de datos relacional es posible crear tablas, accederlas y manipular su contenido por medio de sintaxis *SQL*.

Las bases de datos son una herramienta para organizar tablas. Esto implica que cuando se crea una tabla se lo hace en el contexto de una base de datos, en caso de no especificar explícitamente la base de datos, esta se creará en la base de datos predeterminada. Cualquier instrucción *SQL* que ejecute desde Spark (incluidos los comandos de *DataFrame*) se ejecutan en el contexto de una base de datos.

3.5.3. Datasets

Un *Dataset* es una estructura de datos tipada. Por ser tipada no puede ser utilizada desde *Python* y *R*, ya que estos lenguajes son no tipados. Se desarrollaron específicamente para *Scala* y *Java* que pueden trabajar con código estáticamente tipado.

Los Datasets brindan a los usuarios la capacidad de asignar una clase de *Java*/*Scala* a los registros dentro de un dataset y manipularlos como una colección de objetos, similar a ArrayList en *Java* o Seq en *Scala*. Las Api disponibles son de tipo seguro, lo que significa que no se puede ver los objetos en un *Dataset* como si fueran de otra clase que la clase que se especificó inicialmente.

La clase *Dataset* se parametriza con el tipo de objeto que contiene Dataset<T> en *Java* y Dataset[T] en *Scala*. Por ejemplo, se garantizará que un Dataset<Person> contenga solo objetos de la clase Person. A partir de Spark 2.0, los tipos admitidos son clases que siguen el patrón JavaBean en *Java* y clases de casos en *Scala*. Estos tipos están restringidos porque Spark necesita poder analizar automáticamente el tipo T y crear un esquema apropiado para dichos datos.

Para admitir de manera eficiente los objetos específicos del dominio, se requiere un concepto especial llamado *Codificador*. El *Codificador* es el responsable de asignar el tipo T específico del dominio al sistema de tipo interno de Spark.

Cuando se usan Datasets, para cada fila de los datos, Spark convierte el formato de Spark Fila al objeto que se especificó (una clase de caso o una clase de *Java*). Esta conversión ralentiza el trabajo, pero puede proporcionar más flexibilidad. Notará un impacto en el rendimiento, pero este es en un orden de magnitud muy diferente de lo que podría ver en algo como una función definida por el usuario (UDF) en *Python*, porque los costos de rendimiento no son tan extremos como cambiar los lenguajes de programación, pero es algo importante a tener en cuenta.

En este punto surge una pregunta: ¿en qué casos conviene utilizar Datasets?

El primer caso a considerar es cuando la(s) operación(es) que le gustaría realizar no se pueden expresar mediante manipulaciones de *DataFrame*. Otro caso a considerar para utilizar Datasets es cuando se desea o se necesita seguridad tipográfica y está dispuesto a aceptar el costo de rendimiento para lograrlo.

Un beneficio adicional que brindan los Datasets es que permiten acceder de manera directa a la clase especificada, sin conversiones extra. En caso de necesitar acceder a un atributo en particular se pueden simplemente indicar el nombre del mismo, lo que permitirá hacerlo tanto al tipo del mismo como a su valor correspondiente.

3.5.4. RDDs

Es importante comprender que cuando se utiliza la Api de alto nivel (DataFrames, Datasets o Tablas SQL) todas las cargas de trabajo se compilan a una estructura primitiva fundamental, el *conjunto de datos distribuido resiliente* (*RDD*). Estos constituyen la piedra angular sobre la que se cimienta Spark y prácticamente todo está construido sobre *RDD*.

Si bien desde Spark y su documentación se propicia la utilización de Api de alto nivel, puede ocurrir que en un momento determinado la manipulación de alto nivel no resuelva algún problema puntual que se está tratando de resolver. Para esos casos, es posible que deba usar las Api de nivel inferior de Spark, específicamente los RDDs. Los RDDs tienen un nivel más bajo que los DataFrames porque revelan características de ejecución física (como particiones) a los usuarios finales. Otro caso donde se necesite utilizar Api de bajo nivel es cuando se recibe código Spark heredado escrito con RDDs.

Un *RDD* representa una colección inmutable y dividida de datos que se pueden operar en paralelo y forma parte del núcleo de Spark. Sin embargo, a diferencia de los DataFrames, donde cada registro es una fila estructurada que contiene campos con un esquema conocido, en los *RDD* los registros son solo objetos *Java*, *Scala* o *Python* a elección del programador. Están diseñados para el almacenamiento de datos en memoria RAM de manera distribuida en un cluster. Una característica muy importante que poseen es que son tolerantes a fallas, ya que sobre ellos se puede realizar el seguimiento de las diferentes operaciones realizadas. Se construyen a partir de la lectura del contenido de un archivo o de una base de datos, como así también mediante la paralelización de colecciones de datos.

El punto en contra que tienen los RDDs respecto de las abstracciones de datos de Api de alto nivel es que las Api de alto nivel brindan un cúmulo de funcionalidad que viene lista para usar, mientras que si se decide trabajar con RDDs debe desarrollarse dichas características. Otro punto en contra de los RDDs es que las Api de alto nivel están sujetas a optimización del optimizador *Catalyst* mientras que los RDDs deben optimizarse manualmente.

En los RDDs no existe el concepto de "filas", los registros individuales son solo objetos *Java*/*Scala*/*Python* sin procesar, y se manipulan manualmente en lugar de acceder al repositorio de funciones que tiene en las Api estructuradas. La Api de *RDD* está disponible en *Python*, así como en *Scala* y *Java*. Para *Scala* y *Java*, el rendimiento es en su mayor parte el mismo. En *Python*, sin embargo, puede perder una cantidad sustancial de rendimiento al usar *RDD* ya que ejecutar *RDD* desde *Python* equivale a ejecutar funciones definidas por el usuario (UDF) de *Python* fila por fila.

La documentación de la Api de Spark señala que hay muchas subclases de *RDD*. Sin embargo, como usuario, es probable que solo cree solo dos tipos de *RDD*:

- *RDD* "genérico".
- *RDD* de clave-valor, que proporciona funciones adicionales, como la agregación por clave.

Para uso general, estos serán los dos tipos de *RDD* más importantes. Ambos solo representan una colección de objetos, pero los *RDD* clave-valor tienen operaciones especiales, así como un concepto de partición personalizada por clave.

3.6. Operaciones sobre los Datos

Anteriormente se mencionó que las abstracciones de datos en Spark son estructuras de datos inmutables, lo que implica que no pueden ser cambiados una vez que son creados. El procesamiento de datos en Spark consiste en aplicar operaciones sobre los datos. Estas se agrupan en dos grupos, uno de ellos conocido con el nombre de transformaciones y el restante conocido con el nombre de acciones.

3.6.1. Transformaciones

Las transformaciones son funciones que se pueden aplicar a los datos con el objetivo de manipularlos. Las transformaciones se expresan de manera abstracta y no se ejecutan en el instante en el que se declaran, *Apache Spark* no actúa sobre las transformaciones hasta ser llamado posteriormente por una acción. Este tipo de comportamiento es llamado perezoso (lazy) y todas las funciones de este grupo se comportan de esta manera.

A continuación, se incluye un listado de las transformaciones más comunes:

- Map.
- Filter.
- FlatMap.
- MapPartitions.
- MapPartitionsWithIndex.
- Sample.
- Unión.
- Intersection.
- Distinct.
- GroupByKey.

Dentro de las transformaciones, hay de dos tipos, las que especifican dependencias estrechas (narrow) y las que especifican dependencias amplias (wide). Las transformaciones estrechas son aquellas en las que cada partición de entrada contribuirá a una sola partición de salida. Estas pueden ser ejecutadas sobre una partición sin necesidad de transferir datos entre nodos.

Una transformación amplia tendrá particiones de entrada que contribuirán a muchas particiones de salida. Este tipo de transformaciones necesitan transferir datos entre nodos del cluster para poder realizar la transformación. A menudo esta tarea se conoce como un Shuffle (barajada) en la que Spark intercambia particiones a través del cluster.

Con transformaciones estrechas, Spark puede realizar automáticamente una operación llamada pipelining, lo que significa que, si se especifican múltiples filtros en los datos, todos se realizan en memoria.

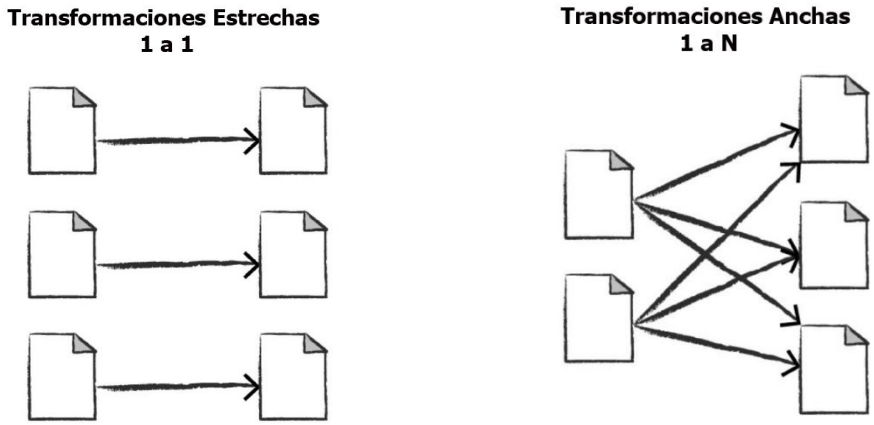

Ilustración 13 – Ejemplo de Transformaciones Estrechas y Amplias

La ilustración 13 se incluye para mostrar de manera gráfica las transformaciones estrechas y las transformaciones amplias. Allí se puede observar y comparar el funcionamiento de cada una de ellas.

3.6.2. Acciones

Las acciones son un grupo de funciones que se aplica sobre los datos y tienen por objetivo devolver resultados. Pueden ser resultados de acciones devolver uno o más valores, o guardar los datos en un medio de almacenamiento.

Las siguientes son las acciones más comunes:

- Reduce.
- Collect.
- Count.
- First.
- Take.
- Foreach.
- CountByKey.
- TakeSample.
- TakeOrdened.
- SaveAsTextFile.
- SaveAsSequenceFile.
- SaveAsObjectFile.

Por basarse en el paradigma MapReduce la acción más frecuente de utilizar es *reduce*, que da nombre al paradigma. Otras acciones se centran en obtener elementos, y también se encuentra un grupo de acciones encargadas de guardar resultados.

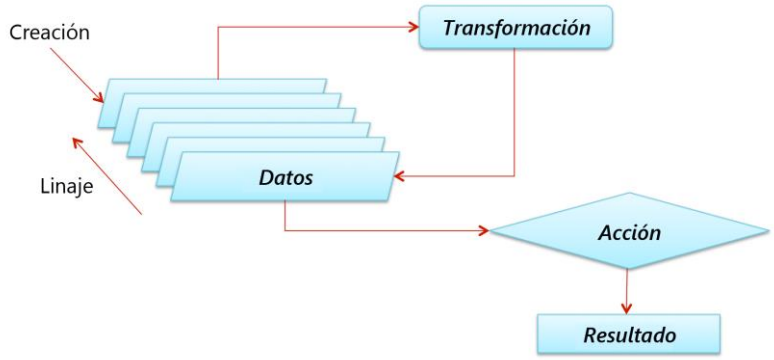

Ilustración 14 - Transformaciones y Acciones

La ilustración 14 muestra como las trasformaciones se pueden aplicar iterativamente a los datos produciendo en con cada nueva trasformación nuevos datos inmutables. Por otro lado, se puede ver como la aplicación de una acción sobre los datos genera resultados de salida.

3.7. Spark en Funcionamiento

La ejecución de aplicaciones en Spark requiere de un proceso controlador y un conjunto de procesos ejecutores. El proceso controlador al ser iniciado ejecuta su función main() y se ubica en un nodo del cluster. Este es responsable de tres cuestiones:

- Mantener la información sobre la aplicación Spark.
- Responder al programa o entrada de un usuario.
- Analizar, distribuir y programar el trabajo entre los ejecutores.

El proceso controlador es absolutamente esencial, es el corazón de Spark y mantiene toda la información relevante durante la ejecución. Este proceso controlador recibe el nombre de *SparkSession*. La instancia de *SparkSession* es la forma en que Spark ejecuta manipulaciones definidas por el usuario en todo el cluster. Existe una correspondencia uno a uno entre una *SparkSession* y una aplicación Spark.

Los ejecutores son responsables de realizar efectivamente el trabajo que el controlador les asigna. Esto significa que cada ejecutor es responsable de:

- Ejecutar el código que le asigna el controlador.
- Informar el estado de la computación en ese ejecutor al nodo del controlador.

La ilustración 15 grafica el funcionamiento de Spark. Allí se puede observar los componentes principales y la interacción entre ellos.

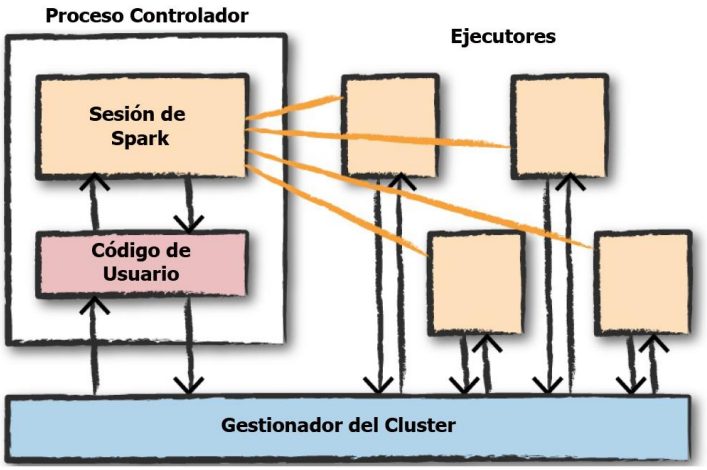

Ilustración 15 – Ejecución de Spark

El *ClusterManager* es quien controla las máquinas físicas y asigna los recursos a las aplicaciones Spark. Esto significa que puede haber varias aplicaciones Spark ejecutándose en un cluster al mismo tiempo. El usuario puede especificar cuántos ejecutores deben crearse en cada nodo a través de configuraciones. Durante la ejecución de aplicaciones el controlador es quien realiza el seguimiento de los recursos disponibles.

El controlador y los ejecutores son simplemente procesos, lo que significa que pueden vivir en la misma máquina o en diferentes máquinas. En modo local, el controlador y los ejecutores se ejecutan (como subprocesos) en su computadora individual en lugar de hacerlo en un cluster. En este caso se puede disponer de tantos ejecutores como núcleos de procesamiento posea la computadora física utilizada.

Los procesos ejecutores siempre ejecutan código Spark, pero este código puede ser controlado desde varios lenguajes de programación por medio de la Api de Lenguajes de Spark.

La escritura de una aplicación en Spark requiere una forma de enviar comandos y datos de usuario, esto se logra instanciando una *SparkSession* en el código de la aplicación que se está construyendo. Posteriormente, a la instancia *SparkSession* se puede solicitar acceder a datos de una fuente de datos concreta, lo cual conduce a la creación de abstracciones de datos específicas.

El código que se encarga de manipular los datos es analizado por el (*SparkContext*) quien crea un grafo de ejecución. El controlador convierte implícitamente el código de usuario, que contiene transformaciones y acciones, en un plan lógico denominado *DAG*.

Luego el *SparkContext* invoca a un proceso interno llamado *DAGScheduler*. Este es la capa de programación de Spark que implementa la programación orientada a etapas mediante trabajos (Jobs) y etapas (Stages). Una tarea (Task) es una abstracción de las unidades de ejecución individuales más pequeñas que se pueden ejecutar.

Para una comprensión de trabajos, etapas, tareas y la vinculación entre ellos se adjunta la ilustración 16.

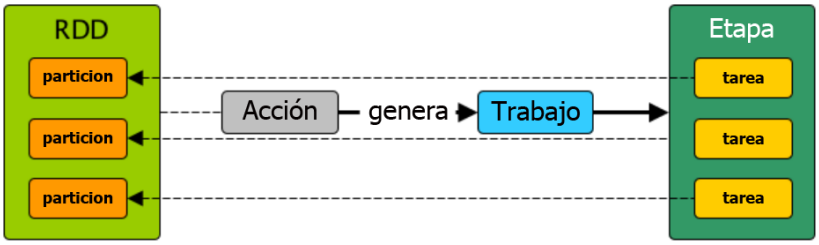

Ilustración 16 – Trabajos y Etapas (https://books.japila.pl/apache-spark-internals/scheduler/Stage/)

Luego el *DAGScheduler* transforma un plan de ejecución lógico (*linaje RDD* de dependencias creadas mediante transformaciones *RDD*) en un plan de ejecución físico (en forma de etapas). Después de que se haya invocado una acción en un *RDD*, *SparkContext* entrega un plan lógico al *DAGScheduler* que, a su vez, se traduce en un conjunto de etapas que se envían como TaskSets para su ejecución.

La ilustración 17 resume la tarea del *DagScheduler*. Allí se puede observar el desglose de un trabajo en etapas y el orden de ejecución de cada una de ellas.

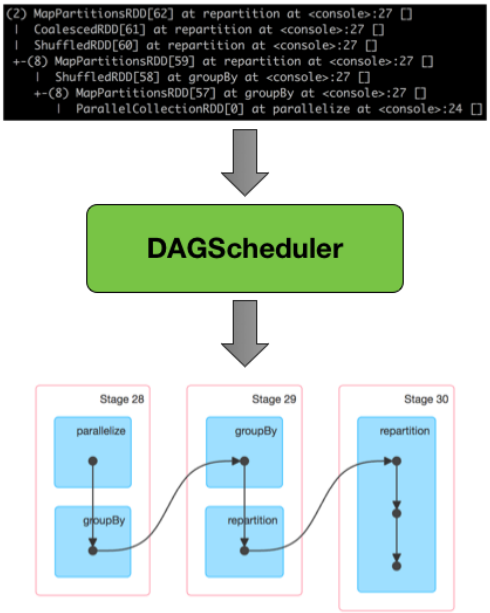

Ilustración 17 – Trabajo del DAGScheduler (https://books.japila.pl/apache-spark-internals/scheduler/DAGScheduler/)

Se puede pensar en las etapas como cálculos que producen resultados intermedios, que de hecho pueden persistir. Las etapas se componen de transformaciones que irán aplicándose de manera perezosa produciendo resultados intermedios en forma de RDDs. Estos se terminan ejecutando cuando se procesan acciones que requieren obtener o devolver resultados, con lo que se termina materializando el *RDD* final resultante.

3.7.1. El DAGScheduler en detalle

El *DAGScheduler* es un componente muy importante en el trabajo de Spark. Este funciona únicamente en el controlador y se crea como parte de la inicialización de *SparkContext*.

Las responsabilidades que asume el *DAGScheduler* son:

- Calcular un *DAG* de ejecución (*DAG* de etapas) para un trabajo, realizando un seguimiento de qué *RDD* y salidas de etapa se materializan y encuentra un cronograma mínimo para ejecutar trabajos. Luego se encarga de envíar las etapas al *TaskScheduler*.
- Determinar las ubicaciones preferidas para ejecutar cada tarea, según el estado actual de la memoria caché, y pasar la información al *TaskScheduler*. El *DAGScheduler* solo está interesado en las coordenadas de ubicación de la memoria caché, es decir, en la identificación del host y del ejecutor, por partición de un *RDD*.
- Proporcionar tolerancia a fallas. Este se encarga de manejar fallas debido a la pérdida de archivos de salida aleatorios; en ese caso, es muy posible que tenga que volver a enviar las etapas antiguas. Las fallas dentro de una etapa que no son causadas por pérdida aleatoria de archivos son manejadas por el propio *TaskScheduler*, que volverá a intentar cada tarea una pequeña cantidad de veces antes de cancelar toda la etapa.

DAGScheduler utiliza una arquitectura de cola de eventos en la que un subproceso puede publicar eventos *DAGSchedulerEvent*, por ejemplo, un nuevo trabajo o etapa que se envía, y que el mismo lee y ejecuta secuencialmente.

Cuando el *DAGScheduler* programa un trabajo como resultado de ejecutar una acción en un *RDD* o de llamar directamente al método *SparkContext.runJob()*, este genera tareas paralelas para calcular resultados (parciales) por partición. Para resumir el trabajo del *DagScheduler* se incluye la ilustración 18.

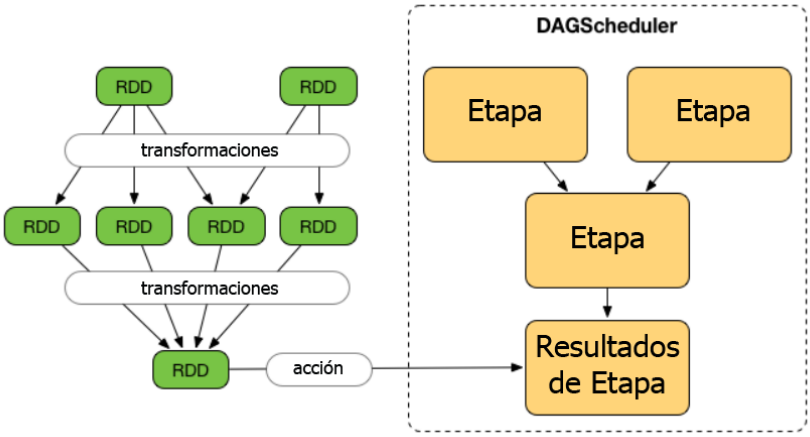

Ilustración 18 – Resumen del Trabajo del DAGScheduler (https://books.japila.pl/apache-spark-internals/scheduler/)

3.8. Procesamiento de Streaming en Spark

Spark Streaming fue construido sobre las capacidades de procesamiento distribuido del motor principal de Spark. Se introdujo en la versión 0.7.0 de Spark, en febrero de 2013, como una versión alfa. Esta versión evolucionó con el tiempo para convertirse en la actualidad en una Api madura, ampliamente adoptada en la industria para procesar datos de streaming a gran escala.

Spark Streaming se basa conceptualmente en una premisa simple pero poderosa, aplicar las capacidades informáticas distribuidas de Spark al procesamiento de streaming mediante la transformación de flujos continuos de datos en flujos discretos sobre los que Spark puede operar.

El enfoque de procesamiento de streaming que utiliza Spark se denomina *modelo de procesamiento por microlotes* y contrasta con el *modelo de procesamiento continuo*, que lo realiza elemento por elemento. Este enfoque prevalece en la mayoría de las otras implementaciones de procesamiento de streaming.

Spark Streaming usa el mismo paradigma de programación funcional que el núcleo de Spark, pero presenta una nueva abstracción, *Discretized Stream* o *DStream*, que expone un modelo de programación para operar específicamente sobre datos de streaming. El trabajo con *DStream* es bastante similar a trabajar con *RDD* por lo que no implica grandes cambios.

Posteriormente el equipo de Spark desarrollo una nueva manera de procesar streaming denominada *Structured Streaming.* Este es un procesador de flujos construido sobre la abstracción *Spark SQL*. Extiende las Api de *Dataset* y *DataFrame* con capacidades de streaming. Como tal, adopta el modelo de transformación orientado a esquemas, que le confiere la parte estructurada de su nombre, y hereda todas las optimizaciones implementadas en *Spark SQL* por medio del optimizador *Catalyst*. Se presentó como una Api experimental con Spark 2.0 en julio de 2016. Un año más tarde, paso a formar parte de la versión estable 2.2.

Structured Streaming utiliza un modelo declarativo para adquirir datos de una secuencia o conjunto de secuencias. Para usar la Api es necesario especificar un esquema de los datos a procesar en el streaming. Además de admitir el modelo de transformación general proporcionado por las Api *Dataset* y *DataFrame* presenta características específicas de streaming, como soporte para tiempo de evento, uniones (Streaming Joins) y separación del tiempo de ejecución subyacente. Esta última característica abre la puerta a la implementación de tiempos de ejecución con diferentes modelos de ejecución. La implementación predeterminada utiliza el enfoque clásico de microlotes, mientras que un backend de procesamiento continuo más reciente brinda soporte experimental para el modo de ejecución continua casi en tiempo real. El trabajo con *Streaming Datafram*e es bastante similar a trabajar con *DataFrame* por lo que no implica grandes cambios.

Aunque el procesamiento por lotes y el procesamiento de flujos suenan diferentes, en la práctica, a menudo necesitan trabajar juntos. A manera de ejemplo, las aplicaciones de streaming a menudo necesitan unir los datos de entrada con conjuntos de datos escritos por un trabajo por lotes, y la salida de los trabajos de streaming suele ser archivos o tablas que se consultan en trabajos por lotes. Para satisfacer estas necesidades, *Structured Streaming* se diseñó desde el principio para interoperar fácilmente con el resto de Spark, incluidas aplicaciones por lotes. De hecho, los desarrolladores de *Structured Streaming* acuñaron el término “aplicaciones continuas” para referirse a aplicaciones de extremo a extremo que consisten en trabajos interactivos, por lotes y de procesamiento de streaming, todos trabajando en los mismos datos para entregar un producto final. La posibilidad de soportar ambos modos de procesamiento, por lotes y de flujos continuos es una de las principales razones por las que los usuarios eligen a Spark como motor distribuido de procesamiento.

Tanto *Spark Streaming* y *Structured Streaming* se basan en las capacidades centrales de *Spark Core* y comparten muchas de las características de bajo nivel en términos de computación distribuida, almacenamiento en caché en memoria e interacciones en cluster.

3.8.1. Gestión de datos en Memoria

Spark ofrece almacenamiento en memoria de los segmentos de un conjunto de datos, que deben cargarse inicialmente desde una fuente de datos. La fuente de datos puede ser un sistema de archivos distribuido u otro medio de almacenamiento. La forma de almacenamiento en memoria de Spark es análoga a la operación de almacenamiento en caché de datos. En las siguientes subsecciones se describen aspectos importantes derivados de la gestión de memoria como la recuperación de fallas, la ejecución perezosa, la latencia y el rendimiento.

3.8.1.1 Recuperación de Fallas

Debido a que Spark conoce exactamente qué fuente de datos se usó para ingerir los datos, como así también conoce todas las operaciones que se realizan sobre el flujo de datos, en caso de ocurrir una falla puede reconstituir el segmento de datos perdidos que estaba en un ejecutor bloqueado, aun cuando haya que hacerlo desde cero. Obviamente, esto va más rápido si esa reconstitución (recuperación, en la jerga de Spark), no necesita ser totalmente desde cero. Entonces, Spark ofrece un mecanismo de replicación, bastante similar a los sistemas de archivo distribuidos.

3.8.1.2 Evaluación Perezosa

El procesamiento de streaming en Spark sigue la filosofía de ejecución perezosa. Una buena parte de las operaciones que se pueden definir sobre los valores en el almacenamiento de Spark tienen una ejecución diferida, y es la ejecución de una operación de salida la que activará la ejecución real de computación en un cluster de Spark.

Hay que hacer mención especial a la siguiente situación, si un programa consta de una serie de operaciones lineales, con la anterior alimentando a la siguiente, los resultados intermedios desaparecen justo después de que el siguiente paso haya consumido su entrada.

3.8.1.3 Latencia y Rendimiento

Spark Streaming, utiliza internamente microlotes. Para ello genera una porción de elementos en un intervalo fijo, y cuando transcurre ese intervalo, comienza a procesar los datos recopilados durante el último intervalo. *Structured Streaming* adopta un enfoque ligeramente diferente en el sentido de que hará que el intervalo en cuestión sea lo más pequeño posible (el tiempo de procesamiento del último microlote) y propondrá, en algunos casos, también un modo de procesamiento continuo. Sin embargo, hoy en día, el funcionamiento por microlotes sigue siendo el modo de ejecución interno dominante del procesamiento de streaming con *Apache Spark*.

Una consecuencia de emplear microlotes es que el procesamiento de un elemento en particular se retrasa al menos el tiempo de duración de un microlote. En primer lugar, los microlotes crean una latencia de referencia. Los expertos deliberan sobre cuán pequeña es posible hacer esta latencia, aunque aproximadamente un segundo es un número relativamente común para el límite inferior.

Para muchas aplicaciones, una latencia en el espacio de unos segundos es suficiente, las siguientes problemáticas ejemplifican esta situación:

- Tener un tablero que actualice los indicadores de rendimiento de un sitio web en los últimos minutos.
- Extraer trending topics más recientes en una red social.
- Calcular las tendencias de consumo de energía de un grupo de hogares.

Si bien la latencia de un microlote es aceptable para muchos casos de procesamiento de streaming, puede ocurrir que el caso a resolver requiera ser procesado lo antes posible, es decir sin tener latencia. Para esos casos existen otros motores de streaming que pueden acelerar el seguimiento de algunos elementos que tienen prioridad, lo que garantiza una respuesta más rápida para ellos. Si su tiempo de respuesta es esencial para estos elementos específicos, *Apache Flink* o *Apache Storm* podrían ser una mejor opción. Pero si solo está interesado en un procesamiento rápido en promedio, Spark es una alternativa interesante.

Spark es un framework que, si bien hace concesiones en cuanto a latencia, está optimizado para construir una canalización de análisis de datos con agilidad, incluyendo creación rápida de prototipos. También puede proporcionar rendimiento estable en tiempo de ejecución aun en condiciones adversas que impliquen recuperación de fallas.

3.8.2. Transformaciones y Agregaciones

Para realizar el procesamiento de datos Spark adopta el enfoque declarativo de programación funcional, se declaran las transformaciones y agregaciones que operan en los flujos de datos, asumiendo que son flujos inmutables. Esto se aplica al procesamiento por lotes y Spark lo hace extensivo al procesamiento de streaming. Una piedra angular de Spark es que no debería tener que cambiar el código de su consulta cuando realiza un procesamiento por lotes o de streaming.

Debido a que no es posible mutar elementos en Spark, se emplean transformaciones para expresar cómo procesar el contenido de un flujo para obtener un flujo de datos derivado. Esto asegura que cualquier flujo de datos pueda rastrearse hasta sus entradas mediante la secuencia de transformaciones y agregaciones aplicadas. Como consecuencia, cualquier proceso particular en un cluster de Spark puede reconstituir el contenido del flujo de datos utilizando solo el código del programa y los datos de entrada, lo que hace que el cálculo sea inequívoco, trazable, reproducible y tolerante a fallas.

Spark hace un uso extensivo de transformaciones y agregaciones. Las transformaciones son cálculos que se expresan de la misma manera para cada elemento del flujo. Por ejemplo, crear un flujo derivado que duplique cada elemento de su flujo de entrada corresponde a una transformación. Las agregaciones, por otro lado, producen resultados que dependen de muchos elementos y potencialmente de todos los elementos del flujo observados hasta ahora. Por ejemplo, recopilar los cinco números más grandes de un flujo de entrada corresponde a una agregación. Calcular el valor promedio de alguna lectura cada 10 minutos también es un ejemplo de agregación.

Dicho de otra manera, las transformaciones expresan dependencias estrechas (narrow) ya que, para producir un elemento de la salida, solo se necesita un único elemento de entrada. Mientras que las agregaciones representan dependencias amplias (wide), donde para producir un elemento de salida es necesario observar muchos elementos del flujo de entrada.

Esta distinción es útil. Nos permite visualizar una forma de expresar funciones básicas que produce resultados utilizando funciones de orden superior.

Aunque *Spark Streaming* y *Structured Streaming* tienen formas distintas de representar un flujo de datos, las Api con las que operan son de naturaleza similar. Ambos se presentan bajo la forma de una serie de operaciones aplicadas a flujos de entrada inmutables y producen un flujo de salida, desencadenado por una operación de salida (sumidero de datos).

3.8.3. Semántica de Entrega de Datos

La computación distribuida tiene sus propios desafíos, ya que a veces incluye no solo fallas de nodos individuales, sino que también es posible encontrar situaciones como particiones de red, en las que algunas partes de nuestro cluster no pueden comunicarse con otras partes del mismo.

Spark ha sido diseñado utilizando una arquitectura de controlador/ejecutor. Una máquina concreta, el controlador, tiene la tarea de realizar un seguimiento de la progresión del trabajo junto con los envíos de trabajo de usuario, y el cálculo de ese programa se produce en los ejecutores a medida que llegan los datos.

Sin embargo, si las particiones de red separan alguna parte del cluster, el controlador podría realizar seguimiento solo de una parte de los ejecutores. En la otra sección de la partición, encontraremos nodos que son totalmente capaces de funcionar, pero que simplemente no podrán dar cuenta del procedimiento de cálculo al controlador. Esto crea un caso interesante en el que esos nodos "zombies" no reciben nuevas tareas, pero bien podrían estar en el proceso de completar algún fragmento de cómputo que se les asignó previamente. Al desconocer la partición, informarán sus resultados como cualquier otro ejecutor. Y debido a que este informe de resultados a veces no pasa por el controlador (por temor a convertir al controlador en un cuello de botella), el informe de esos resultados zombis podría tener éxito.

Debido a que el controlador, el único punto de contabilidad, no sabe que esos ejecutores zombis todavía están funcionando y reportando resultados, reprogramará las mismas tareas que los ejecutores perdidos tuvieron que realizar en nuevos nodos. Esto crea un problema de doble respuesta en el que las máquinas zombis perdidas a través de la partición y las máquinas que soportan las tareas reprogramadas informan los mismos resultados. Esto tiene consecuencias reales, un ejemplo de cálculo de flujo que mencionamos anteriormente son las tareas de enrutamiento para transacciones financieras. Un doble retiro, en ese contexto, o una doble orden de compra de acciones, podría tener tremendas consecuencias.

No es solo el problema antes mencionado el que origina diferentes semánticas de procesamiento. Otra razón importante es que cuando la salida de una aplicación de procesamiento de flujo y el punto de control de estado no se pueden completar en una operación atómica, se dañarán los datos si ocurre una falla entre el punto de control y la salida. Por lo tanto, estos desafíos han llevado a una distinción entre procesamiento al menos una vez y procesamiento como máximo una vez:

- *Al menos una vez*. Este procesamiento asegura que cada elemento de un flujo ha sido procesado una o más veces.
- *Como máximo una vez*. Este procesamiento garantiza que cada elemento del flujo se procese una vez o menos.
- *Exactamente una vez*. Esta es la combinación de *al menos una vez* y *como máximo una vez*.

El procesamiento al menos una vez implica asegurar de que se haya tratado cada parte de los datos iniciales, se trata de la falla del nodo a la que se refirió anteriormente. Como mencionamos, cuando un proceso de transmisión sufre una falla parcial en la que se deben reemplazar algunos nodos o se deben volver a calcular algunos datos, debemos volver a procesar las unidades de cómputo perdidas mientras se mantiene la ingesta de datos. Ese requisito significa que, si no se respeta al menos una vez el procesamiento, existe la posibilidad de que, bajo ciertas condiciones, se pierdan datos.

La noción antisimétrica se denomina procesamiento como máximo una vez. En este caso los sistemas de procesamiento garantizan que los nodos zombis que repiten los mismos resultados que un nodo reprogramado se tratan de manera coherente, para ello se debe hacer un seguimiento de los datos y del conjunto de resultados. Al realizar un seguimiento de los datos y sobre sus resultados, se puede descartar resultados repetidos, lo que genera garantías de procesamiento de una vez como máximo. La forma en que se logra esto se basa en la noción de idempotencia aplicada a la *última milla* de la recepción de resultados. La idempotencia califica una función de tal manera que si se la aplica dos veces (o más) a cualquier dato, se obtendrá el mismo resultado que la primera vez. Esto se puede lograr haciendo un seguimiento de los datos para los que se está informando un resultado y teniendo un sistema de contabilidad en la salida de nuestro proceso de transmisión.

3.8.3.1. Microlotes: una aplicación de procesamiento masivo síncrono

Spark Streaming, el modelo más maduro de procesamiento de streaming en Spark, se aproxima a lo que se denomina un sistema de paralelismo síncrono masivo (Bulk Synchronous Parallelism o BSP). La esencia de BSP es incluir:

- Una distribución dividida del trabajo asíncrono.
- Una barrera sincrónica, que entra a intervalos fijos.

La división es la idea principal, donde cada uno de los pasos sucesivos del trabajo a realizar en la transmisión se separa en una cantidad de fragmentos paralelos que son aproximadamente proporcionales a la cantidad de ejecutores disponibles. Cada ejecutor recibe su propio fragmento de trabajo y trabaja por separado hasta que llega el segundo fragmento. Un recurso en particular tiene la tarea de realizar un seguimiento del progreso de la computación. Con *Spark Streaming*, este es un punto de sincronización en el "controlador" que permite que el trabajo avance al siguiente paso. En esos pasos programados, todos los ejecutores del cluster están ejecutando el mismo cómputo. Tenga en cuenta que lo que se transmite en este proceso de programación son las funciones que describen el procesamiento que el usuario desea ejecutar en los datos. Los datos ya están en los ejecutores, y la mayoría de las veces se entregan directamente a estos recursos durante la vida útil del cluster.

Este enfoque fue denominado *estilo de transferencia de funciones* por Heather Miller en 2016 [7] e implica pasar funciones seguras de forma asíncrona a datos distribuidos, estacionarios e inmutables en un contenedor sin estado, y usar combinadores perezosos para eliminar estructuras de datos intermedias.

La frecuencia con la que se programan más rondas de procesamiento de datos está dictada por un intervalo de tiempo. Este intervalo de tiempo es de una duración arbitraria que se mide en tiempo de procesamiento por lotes; es decir, como una observación de tiempo de "reloj" en el cluster.

Para el procesamiento de streaming, se prefiere implementar barreras a intervalos pequeños y fijos que se aproximan mejor a la noción de procesamiento de datos en tiempo real.

3.8.3.2. Procesamiento continuo o de un registro a la vez

En el procesamiento de un registro a la vez, por el contrario, se analiza todo el cálculo tal como lo describen las funciones especificadas por el usuario y se implementa como canalizaciones utilizando los recursos del cluster. Luego, lo único que queda es hacer fluir los datos a través de los diversos recursos, siguiendo la canalización prescrita y se aplican las operaciones uno a uno sobre los registros de datos que se van recibiendo.

Los sistemas de procesamiento que funcionan de acuerdo con este paradigma incluyen *Apache Flink*, *Naiad*, *Storm* e *IBM Streams*. Esto no significa necesariamente que esos sistemas sean incapaces de procesar microlotes, sino que caracteriza su modo de operación principal o más nativo.

3.8.3.3. Procesamiento en microlotes vs. procesamiento continuo o de uno a la vez

A pesar de su mayor latencia, los sistemas de microlotes ofrecen importantes ventajas:

- Son capaces de adaptarse a los límites de la barrera de sincronización. Esa adaptación podría representar la tarea de recuperarse de una falla, si se ha demostrado que varios ejecutores se vuelven deficientes o pierden datos. La sincronización periódica también puede brindar la posibilidad de agregar o eliminar nodos ejecutores, lo que da la posibilidad de aumentar o reducir nuestros recursos según la carga observada del cluster, en su fuente de datos.
- A veces, a nuestros sistemas BSP les resulta más fácil proporcionar una gran consistencia porque sus determinaciones de lotes, que indican el comienzo y el final de un lote de datos en particular, son deterministas y están registradas. Por lo tanto, cualquier tipo de cálculo se puede rehacer y producir los mismos resultados la segunda vez.
- Tener datos disponibles como un conjunto que se puede inspeccionar al comienzo del microlote permite realizar optimizaciones eficientes que pueden proporcionar ideas sobre la forma de realizar los cálculos. Al aprovechar esto en cada microlote, se puede considerar el caso específico en lugar del procesamiento general, que se aplica para todas las entradas posibles. Por ejemplo, se podría tomar una muestra o calcular una medida estadística antes de decidir procesar o descartar cada microlote.

3.8.3.4. Uniendo Microbatch y procesamiento continuo

El matrimonio entre el microprocesamiento por lotes y el procesamiento de un registro a la vez, como se implementa en sistemas como *Apache Flink* o *Naiad*, sigue siendo un tema de investigación.

Aunque no resuelve todos los problemas, *Structured Streaming* está respaldado por una implementación principal que se basa en microlotes, no expone esa elección a nivel de Api, lo que permite una evolución que es independiente de un intervalo de lotes fijo. De hecho, el modelo de ejecución interno predeterminado de *Structured Streaming* es el de microlotes con un intervalo de lotes dinámico. Actualmente *Structured Streaming* también está implementando procesamiento continuo para algunos operadores.

3.8.3.5. Intervalo de lote dinámico

El intervalo de lote dinámico es la noción de que el recálculo de datos en un *DataFrame* o *Dataset* de streaming consiste en una actualización de los datos existentes con los nuevos elementos vistos a través del flujo. Esta actualización se produce en función de un disparador y la base habitual de esto sería la duración del tiempo. Esa duración de tiempo todavía se determina en función de una señal de reloj mundial fija que se espera se sincronice dentro de todo el cluster y que representa una única fuente de tiempo síncrona que comparten todos los ejecutores.

Sin embargo, este desencadenante puede ser *tan a menudo como sea posible*. Esta declaración es simplemente la idea de que se debe iniciar un nuevo lote tan pronto como se haya procesado el anterior, dada una duración inicial razonable para el primer lote. Esto significa que el sistema lanzará lotes con la mayor frecuencia posible. En esta situación, la latencia que se puede observar

es más cercana a la del procesamiento de un elemento a la vez. La idea aquí es que los microlotes producidos por este sistema convergerán al tamaño manejable más pequeño, haciendo que el flujo fluya más rápido a través de los cálculos sucesivos del ejecutor. Tan pronto como se produzca ese resultado, el controlador Spark iniciará y programará un nuevo lote.

3.8.3.6. Modelo de procesamiento de Structured Streaming

Structured Streaming es una evolución de *Spark Streaming* que incorpora mejoras y optimizaciones como el uso del optimizador *Catalyst* y el procesamiento de lenguaje *SQL*, esto lo convierte en la opción que el equipo de desarrollo de Apache Spark recomienda para el procesamiento de streaming. Por esta razón, en esta sección se profundiza en su modelo de procesamiento.

Los pasos principales en el procesamiento de streaming son los siguientes:

1. Cuando el controlador Spark activa un nuevo lote, el procesamiento comienza con la actualización de la cuenta de datos leídos de una fuente de datos, en particular, obteniendo compensaciones de datos para el comienzo y el final del último lote.
2. A esto le sigue la planificación lógica, la construcción de pasos sucesivos que se ejecutarán sobre los datos, seguida de la planificación de consultas (optimización intrapaso).
3. Y luego el lanzamiento y la programación del cálculo real, al agregar un nuevo lote de datos, para luego actualizar la consulta que se está procesando.

Desde el punto de vista del modelo de cómputo, veremos que la Api de *Structured Streaming* es significativamente diferente de *Spark Streaming*. A pesar de las diferencias internas entre las dos formas de procesamiento de streaming, ambas Api están construidas para ser utilizadas de manera muy similar. Esto facilita el aprendizaje y la rápida implementación, como así también facilita el hecho de poder migrar de manera simple entre un enfoque y el otro.

3.8.3.7. La desaparición del intervalo de lote

A continuación, se explica qué significan los lotes de *Structured Streaming* y su impacto con respecto a las operaciones. En *Structured Streaming*, el intervalo de lote que se está usando ya no es un presupuesto de cómputo. Con *Spark Streaming*, la idea era que, si producimos datos cada dos minutos y se hace pasar datos a la memoria de Spark cada dos minutos, se debería producir los resultados del cálculo en ese lote de datos en al menos dos minutos, para borrar la memoria de nuestro cluster para el siguiente microlote. Idealmente, tantos datos fluyen hacia afuera como hacia adentro, y el uso de la memoria colectiva de nuestro cluster permanece estable.

En *Structured Streaming*, sin esta sincronización de tiempo fijo, la capacidad para ver los problemas de rendimiento en el cluster es más compleja. En un cluster inestable, es decir, un cluster incapaz de borrar los datos al terminar de realizar los cálculos tan rápido como arriban los datos nuevos verá tiempos de procesamiento por lotes cada vez mayores, con un crecimiento acelerado. En ese contexto, controlar este tiempo de procesamiento por lotes es fundamental. Sin embargo, si se cuenta con un cluster que tiene el tamaño correcto con respecto al rendimiento de nuestros datos, hay muchas ventajas para tener una actualización lo más frecuente posible. En particular, deberíamos esperar ver resultados muy frecuentes de nuestro cluster con una granularidad más alta de lo que se suele observar con un tiempo de intervalo de lote conservador.

3.8.4. Fuentes de Datos

En *Structured Streaming*, una fuente es una abstracción que nos permite consumir datos de un productor de datos de streaming. El concepto detrás de la interfaz de origen es que los datos de streaming constituyen un flujo continuo de eventos a lo largo del tiempo y se puede ver como una secuencia, indexada con un contador que se incrementa monótonamente.

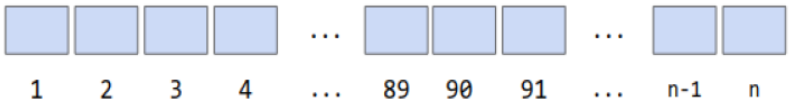

Ilustración 19 - Flujo visto como secuencia indexada de eventos

Las compensaciones, como se muestra en la ilustración 19, se utilizan para solicitar datos de la fuente externa y para indicar qué datos ya se han consumido. *Structured Streaming* sabe cuándo hay datos para procesar preguntando por el desplazamiento actual del sistema externo y comparándolo con el último desplazamiento que ha procesado. Los datos a procesar se solicitan obteniendo un lote entre dos compensaciones de inicio y fin. Cuando un lote ha sido procesado se debe informar a su fuente que estos han sido correctamente procesados mediante el envío de una confirmación (commit). El contrato de origen garantiza que todos los datos con una compensación menor o igual a la compensación confirmada han sido procesados y que las solicitudes posteriores contendrán solo compensaciones mayores que la compensación confirmada. Dadas estas garantías, las fuentes pueden optar por descartar los datos procesados para liberar recursos del sistema.

Este proceso se repite constantemente, lo que garantiza la adquisición de nuevos datos de streaming. Para recuperarse de una falla eventual, las compensaciones a menudo se controlan en el almacenamiento externo.

Además de la interacción basada en compensación, las fuentes deben cumplir dos requisitos para ser confiables:

- Deben poder reproducirse en el mismo orden.
- Deben proporcionar un esquema.

Al igual que se puede rebobinar una serie o película que se está mirando para ver una parte que no se pudo ver por una distracción, las fuentes de datos deben ofrecer la capacidad de reproducir una parte de la transmisión que ya se solicitó, pero que no se confirmó con un *commit*. Esto se hace llamando a *getBatch* con el rango de compensación que se quiere recibir nuevamente.

En *Structured Streaming*, la capacidad de ser reproducible es la capacidad de solicitar una parte del stream que ya había sido solicitada pero aún no confirmada.

Se considera que una fuente de datos es confiable cuando esta puede producir un rango de compensación no confirmado, incluso después de una falla total del proceso de transmisión. En este proceso de recuperación de fallas, las compensaciones se restauran desde su último punto de control conocido y se solicitan nuevamente desde la fuente. Esto requiere que el sistema de transmisión que respalda la implementación de origen almacene datos de forma segura fuera del proceso de transmisión. Al exigir la capacidad de reproducción de la fuente, *Structured Streaming* delega la responsabilidad de recuperación a la fuente misma. Esto implica que solo las fuentes

confiables trabajan con *Structured Streaming* para crear sólidas garantías de entrega de extremo a extremo.

Usando el criterio de si una fuente es confiable o no, las fuentes de streaming pueden clasificarse en:

- Confiables: entre ellas se encuentran la fuente de archivo y la fuente de Kafka.
- No Confiables: fuente de socket y fuente de tasa (rate).

Las fuentes no confiables generalmente se utilizan en entornos de desarrollo o de testing, aunque también pueden ser utilizadas en aquellos casos donde se puede tolerar perdida de datos en un sistema de producción.

Una característica definitoria de las Api estructuradas de Spark es que se basan en la información del esquema para manejar los datos. A diferencia del procesamiento de cadenas opacas o blobs de matriz de bytes, el trabajo basado en esquema proporciona información sobre cómo se configuran los datos en términos de campos y tipos. Esta información se emplea para hacer optimizaciones en diferentes niveles de la pila, desde la planificación de consultas hasta la representación binaria interna de los datos, el almacenamiento y el acceso a ellos. En *Structured Streaming*, se reutiliza la Api de Spark SQL para crear definición de esquemas.

3.8.4.1. Uso de las fuentes de datos

Las fuentes de datos no se crean directamente. En su lugar, *SparkSession* proporciona un método de creación, *readStream*, que expone la Api para especificar el tipo de fuente de streaming, su formato por medio del método *format*, y su configuración. Cada implementación de fuente de datos provee diferentes opciones ajustables por parámetros.

Internamente, la llamada a *readStream* crea una instancia de *DataStreamBuilder*. Esta instancia se encarga de administrar las diferentes opciones que se proporcionan en la llamada al método constructor de la misma.

Invocar al método *load* en esta instancia de *DataStreamBuilder* valida las opciones proporcionadas al constructor y, si todo funciona, devuelve un *StreamingDataFrame*.

La carga de una fuente de streaming es perezosa. Lo que se obtiene es una representación de la secuencia, incorporada en la instancia del *StreamingDataFrame*, que se puede usar para expresar la serie de transformaciones a aplicar para implementar la lógica de negocios específica. La creación de un *StreamingDataFrame* no da como resultado que se consuma o procese ningún dato hasta que se materializa la transmisión, y para que esto suceda se requiere una consulta.

3.8.4.2. Fuentes de datos disponibles

Las fuentes de streaming disponibles a partir de la versión 2.4 de Spark son:

- Json, orco, parquet, csv, texto, archivo de texto. Estas fuentes son basadas en archivos. La funcionalidad básica de Spark con estas fuentes de datos es monitorear una ruta (una carpeta) en un sistema de archivos y consumir archivos colocados en ella en forma atómica. Si esto no ocurre Spark procesará los archivos parcialmente escritos antes de que haya terminado su escritura. En sistemas de archivos que muestran escrituras parciales, como archivos locales o HDFS, esto se hace mejor escribiendo el archivo en un directorio externo y moviéndolo al directorio de entrada cuando termine su escritura.

Los archivos encontrados serán analizados por el formateador especificado. Por ejemplo, si se proporciona json, se utilizará el lector json de Spark para procesar los archivos, utilizando la información de esquema proporcionada.

- Kafka: en este caso se crea un consumidor capaz de recuperar datos de *Apache Kafka*. Apache Kafka es un sistema distribuido de publicación y suscripción para flujos de datos. Kafka le permite publicar y suscribirse a flujos de registros como lo haría con una cola de mensajes: estos se almacenan como flujos de registros con tolerancia a fallas. Kafka trabaja como un búfer distribuido.
  Kafka le permite almacenar secuencias de registros en categorías que se denominan temas. Cada registro en Kafka consta de una clave, un valor y una marca de tiempo. Los temas consisten en secuencias inmutables de registros para los cuales la posición de un registro en una secuencia se denomina desplazamiento. Leer datos se llama suscribirse a un tema y escribir datos es tan simple como publicar en un tema.
- Socket: la fuente de socket le permite enviar datos a sus streams a través de sockets TCP. Para iniciar uno, se debe especificar un host y un número de puerto desde el que leer los datos. Spark abrirá una nueva conexión TCP para leer desde esa dirección. La fuente de socket generalmente no se usa en ambientes de producción porque el socket se encuentra en el controlador y no proporciona garantías de tolerancia a fallas de extremo a extremo.
- Rate: genera un flujo de filas a la velocidad dada por la opción rowsPerSecond. Está pensado principalmente como fuente de prueba.

3.8.5. Sumideros de datos

Así como las fuentes permiten obtener datos en *Structured Streaming*, los sumideros especifican el destino para el conjunto de resultados de ese flujo. Los sumideros y el motor de ejecución también son responsables de realizar un seguimiento fiable del progreso exacto del procesamiento de datos.

Conceptualmente, los sumideros son la abstracción que representa la manera de producir datos salientes. *Structured Streaming* viene con varias fuentes de salida integradas y define una Api que permite crear sumideros personalizados para otros sistemas que no son compatibles de forma nativa.

3.8.5.1. Comprensión de los sumideros

Los sumideros sirven como adaptadores de salida entre la representación de datos internos en *Structured Streaming* y los sistemas externos. Los sumideros se materializan como una tabla de resultados donde cada elemento saliente es un registro que conforma dicha tabla. Proporcionan una ruta de escritura para los datos resultantes del procesamiento de secuencias. Además, también deben cerrar el ciclo de entrega de datos confiables.

Para asegurar la entrega de datos fiables de extremo a extremo, los sumideros deben proporcionar una operación de escritura idempotente. Idempotente significa que el resultado de ejecutar la operación dos o más veces es igual a ejecutar la operación una vez. Al recuperarse de una falla, Spark puede volver a procesar algunos datos que se procesaron parcialmente en el momento en que ocurrió la falla. En el lado de la fuente, esto se consigue usando la funcionalidad de reproducción. Las fuentes confiables deben proporcionar un medio para reproducir datos no confirmados, en función de una compensación determinada. Del mismo modo, los sumideros deben proporcionar los medios para eliminar los registros duplicados antes de que se escriban en la fuente externa.

La combinación de una fuente reproducible y un sumidero idempotente es lo que otorga a *Structured Streaming* su semántica de entrega de datos exactamente una vez. Los sumideros que no pueden implementar el requisito idempotente darán como resultado garantías de entrega de extremo a extremo de, como máximo, *al menos una vez*. Los sumideros que no pueden recuperarse de la falla del proceso de transmisión se consideran poco confiables porque pueden perder datos.

De la misma forma que se clasificó a las fuentes de datos, los sumideros de streaming pueden clasificarse en:

- Confiables: se consideran confiables a los sumideros que brindan una semántica de entrega de datos bien definida y son tolerantes a fallas durante el proceso de transmisión. Este tipo de sumidero se considera apto para ambientes de producción.
- No Confiables: se considera no confiables a los sumideros que no proporcionan tolerancia a fallas. Por esto se los desaconseja para ambientes de producción. Su utilización generalmente es durante el desarrollo y testing de aplicaciones de streaming.

3.8.5.2. Sumideros Disponibles

Dentro de los sumideros considerados confiables, se encuentran:

- Sumidero de Archivo: encausan los datos de salida hacia uno o más archivos en un directorio en el sistema de archivos. Admite los mismos formatos de archivo que la fuente de archivo, y sus opciones son: JSON, Parquet, ORCO, valores separados por comas (CSV) y texto. Los sistemas de archivos escalables, confiables y distribuidos, como HDFS o almacenes de objetos como Amazon Simple Storage Service (Amazon S3), permiten almacenar grandes conjuntos de datos como archivos en formatos arbitrarios. Cuando se ejecuta en modo local, en tiempo de exploración o desarrollo, es posible usar el sistema de archivos local para este sumidero.
- Sumidero Kafka: este sumidero escribe datos en *Apache Kafka*, manteniendo efectivamente los datos "en movimiento". Esta es una opción interesante para integrar los resultados de nuestro proceso con otros frameworks de streaming que se basan en Kafka como la columna vertebral de gestión de datos.

Los siguientes sumideros se proporcionan para admitir la interacción y experimentación con *Structured Streaming* y se consideran no confiables:

- Sumidero en Memoria: este sumidero crea una tabla temporal con los resultados de procesar el flujo. La tabla resultante se puede consultar dentro del mismo proceso de máquina virtual Java (JVM), lo que permite consultas en el cluster para acceder a los resultados del proceso de transmisión.
- Sumidero de consola: se encarga de imprimir los resultados del procesamiento de streaming en la consola. Esto es útil en etapas de desarrollo para inspeccionar visualmente los resultados de la transmisión.

3.8.5.3. Modos de salida

La salida de una consulta en *Structured Streaming* genera la tabla de resultados. La tabla de resultados contiene los resultados de la consulta, a partir de la cual se extraen datos para un almacén de datos externo.

Se denomina modo de salida a la manera en la que los registros van incorporándose a la tabla de resultados. El concepto es el mismo que los modos de guardado en un *DataFrame* estático. *Structured Streaming* tiene tres modos de salida:

- Modo Adición o Agregación: en el almacenamiento externo se escriben solo las filas nuevas anexadas en la tabla de resultados desde el último activador. Esto solo es aplicable a las consultas en las que no se esperan cambios en las filas existentes de la tabla de resultados. Este es el modo de comportamiento predeterminado. Este modo garantiza que cada fila se generará solo una vez (suponiendo un sumidero tolerante a fallas).
- Modo Completo: en el almacenamiento externo se escribe toda la tabla de resultados actualizada. El conector de almacenamiento decide cómo controlar la escritura de toda la tabla. Esto es útil cuando está trabajando con algunos datos con estado para los cuales se espera que todas las filas cambien con el tiempo o el receptor que está escribiendo no admite actualizaciones de nivel de fila.
- Modo Actualización (disponible desde Spark 2.1.1): en el almacenamiento externo se escriben solo las filas que se actualizaron en la tabla de resultados desde el último activador. Difiere del modo Completo en que el modo Actualización solo genera las filas que han cambiado desde el último activador. Si la consulta no contiene agregaciones, es equivalente al modo Adición. Naturalmente, los sumideros deben admitir actualizaciones de nivel de fila para admitir este modo.

Para controlar cuándo se envían los datos a nuestro receptor, se configura un activador. De forma predeterminada, *Structured Streaming* iniciará los datos tan pronto como el activador anterior complete el procesamiento. Puede usar activadores para asegurarse de no sobrecargar su receptor de salida con demasiadas actualizaciones o para intentar controlar el tamaño de los archivos en la salida. Actualmente, hay un tipo de activador periódico, basado en el tiempo de procesamiento, así como un activador "único" para ejecutar un paso de procesamiento una vez. Es probable que se agreguen más activadores en el futuro.

El activador de tiempo de procesamiento, simplemente especifica una duración como una cadena (también puede usar *Duration* en Scala o *TimeUnit* en Java).

3.8.6. Procesamiento sin estado y con estado

El procesamiento con estado es necesario cuando necesita usar o actualizar información intermedia (de estado) durante períodos de tiempo más largos (ya sea en un enfoque de microlote o de registro a la vez). Esto puede suceder cuando usa el tiempo del evento o cuando se realiza una agregación en una clave, ya sea que involucre el tiempo del evento o no.

Cuando se necesita realizar operaciones con estado, Spark se encarga de ocultar la complejidad inherente. Por ejemplo, cuando se especifica una agrupación, *Structured Streaming* mantiene y actualiza la información y el usuario solo necesita especificar la lógica. Al realizar una operación con estado, Spark almacena la información intermedia en un almacén de estado. La implementación del almacén de estado actual de Spark es un almacén de estado en memoria que se vuelve tolerante a fallas al almacenar el estado intermedio en el directorio del punto de control.

Hay momentos en los que se necesita un control detallado sobre qué estado debe almacenarse, cómo se actualiza y cuándo debe eliminarse, ya sea explícitamente o mediante un tiempo de espera. Esto se denomina procesamiento con estado arbitrario (o personalizado) y Spark permite almacenar esencialmente cualquier información que desee en el transcurso del procesamiento del

flujo. Esto proporciona una gran flexibilidad y potencia, y permite manejar con bastante facilidad lógica de negocios compleja.

Para poder implementar el procesamiento con estado es necesario considerar la utilización de la hora del evento, ya que como se explicó anteriormente, diversos factores pueden incidir en la transmisión y recepción de eventos. *Structured Streaming* se basa en *Spark SQL* y admite *java.sql.Timestamp* como tipo de marca de tiempo. Para otros tipos base, primero debemos convertir el valor a Timestamp antes de poder usarlo para el procesamiento de hora del evento.

Los eventos pueden llegar tarde o incluso no llegar nunca. ¿Qué tan tarde es demasiado tarde? ¿Por cuánto tiempo mantenemos agregaciones parciales antes de considerarlas completas? Para responder a estas preguntas, se introdujo el concepto de marcas de agua en *Structured Streaming*. Una marca de agua es un umbral de tiempo que dicta cuánto tiempo se espera los eventos antes de declarar que ya es demasiado tarde. Los eventos que se consideran retrasados más allá de la marca de agua se descartarán.

3.8.6.1. Agregación de ventanas basadas en el tiempo

En *Structured Streaming* desde la perspectiva de la Api, las agregaciones de ventanas se declaran utilizando una función de ventana como criterio de agrupación. La función de ventana debe aplicarse al campo que queremos usar como hora del evento.

La operación más simple es simplemente contar el número de ocurrencias de un evento en una ventana de tiempo dada. Cuando el activador se dispare se actualiza la tabla de resultados (según el modo de salida), que operará con los datos recibidos desde el último activador. El fragmento de código de la ilustración 20 muestra la definición de ventana de 10 minutos sin superposición entre ellas, esto quiere decir que cada evento cae en una única ventana.

```
# in Python
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes")).count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

Ilustración 20 – Definición de Ventana giratoria

La agregación de la ilustración implica que los recuentos se actualizarán en tiempo real, lo que significa que, si se agregan nuevos eventos en sentido temporal ascendente a nuestro sistema, *Structured Streaming* actualizará esos recuentos en consecuencia. El código fuente mostrado emplea el modo de salida completo, en este caso Spark generará la tabla de resultados completa independientemente de si hemos visto el conjunto de datos completo.

3.8.6.2. Agregación de ventanas deslizantes

Una alternativa disponible en *Structured Streaming* es utilizar ventanas de tiempo deslizantes. Esto se emplea en caso de que se requiera obtener recuentos personalizados donde la activación de cada ventana sea independiente del tiempo de cada ventana.

En el ejemplo incluido en la ilustración 21, se observan ventanas de 10 minutos, comenzando cada una de ella cada cinco minutos. Por lo tanto, cada evento se dividirá en dos ventanas diferentes.

```
# in Python
from pyspark.sql.functions import window, col
withEventTime.groupBy(window(col("event_time"), "10 minutes", "5 minutes"))\
  .count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

Ilustración 21 – Definición de Ventana deslizante

3.8.6.3. Agregación con marcas de agua

Los ejemplos anteriores no dicen nada respecto a cuanto hay que esperar por cada uno de los datos en caso de que estos lleguen con retrasos, y en una situación así Spark necesitará almacenar esos datos intermedios para siempre, porque nunca se especificó una marca de agua.

La marca de agua permite indicar a Spark el tiempo máximo a esperar en cada ventana definida, esto permite que una vez transcurrido el tiempo de la marca definida los recuentos se consideren definitivos.

En la Api DStreams, no había una forma sólida de manejar los datos atrasados, de esta manera si ocurría un evento en un momento determinado, pero no llegaba al sistema de procesamiento cuando comenzaba el lote para una ventana determinada, este evento tardío aparecía en otros lotes de procesamiento. La transmisión estructurada soluciona esto. En la ilustración 22 se ejemplifica el uso de marca de agua de 30 minutos.

```
# in Python
from pyspark.sql.functions import window, col
withEventTime\
  .withWatermark("event_time", "30 minutes")\
  .groupBy(window(col("event_time"), "10 minutes", "5 minutes"))\
  .count()\
  .writeStream\
  .queryName("pyevents_per_window")\
  .format("memory")\
  .outputMode("complete")\
  .start()
```

Ilustración 22 – Ventana deslizante con marca de agua

Al hacer esto, le indicamos a Spark que debe ignorar cualquier evento que ocurra más de 10 minutos de "tiempo de evento" después de un evento anterior. Por el contrario, esto también establece que esperamos ver todos los eventos en 10 minutos. Después de eso, Spark debería eliminar el estado intermedio y, según el modo de salida, hacer algo con el resultado. Es importante comprender que se necesita especificar marcas de agua porque si no se hiciera, se necesitaría mantener todas nuestras ventanas para siempre, esperando que se actualicen para siempre. Esto nos lleva a la pregunta central cuando se trabaja con el tiempo del evento: ¿qué tan tarde se espera recibir los datos? La respuesta a esta pregunta será la marca de agua que se debe configurar.

Especificar una marca de agua le permite liberar objetos de la memoria, lo que permite que su transmisión continúe ejecutándose durante mucho tiempo sin comprometer el uso de memoria ni los cálculos ya obtenidos.

3.8.6.4. Ventanas de sesión

En la versión 3.2 de Spark se incluye un nuevo tipo de ventana, las ventanas de sesión. Estas tienen una característica diferente en comparación con los tipos de ventanas descriptas anteriormente. La principal característica que poseen es que son de tamaño dinámico. Cuando se define una ventana de sesión debe definirse un tiempo específico de espera.

Una vez definida, una ventana de sesión, se inicia cuando se recibe el primer elemento. Mientras se reciben nuevos elementos dentro del tiempo de espera definido, continua el procesamiento de la misma. Cada nuevo elemento entrante extiende el tiempo de espera y la ventana continua su procesamiento. Una vez que se cumple el tiempo de espera especificado y no se reciben nuevos elementos, la ventana de sesión se cierra.

Funciona de manera similar a una sesión en un sitio web que tiene un tiempo de espera por sesión, si inicia sesión en un sitio web y no muestra ninguna actividad durante un tiempo x, el sitio web forzará el cierre de sesión. El tiempo de espera de la sesión se extiende cada vez que la sesión muestra actividad, es decir, cuando se recibe un nuevo elemento.

La ilustración 23 muestra la diferencia entre los tipos de ventanas que se pueden definir en Spark. Allí se observa en primera instancia las ventanas giratorias sin superposición y de tamaño fijo. A continuación, se observan ventanas deslizantes de tamaño fijo y con superposición. Por último, se muestra ventanas de sesión que tienen un tiempo de inactividad definido.

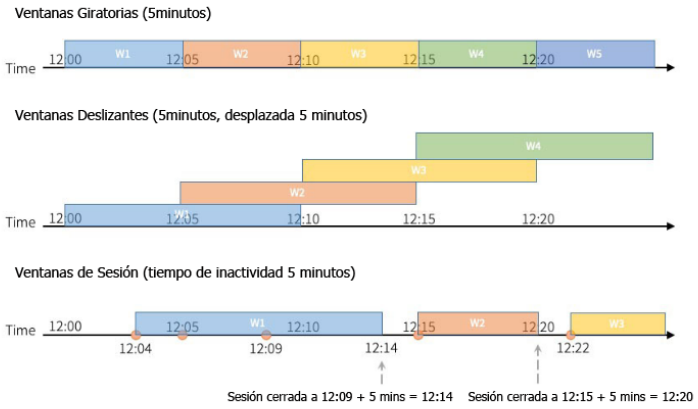

Ilustración 23 – Diferencias entre tipos de ventanas

Si bien anteriormente se podía implementar el manejo de sesiones en Spark, para hacerlo se requería el uso de *flatMapGroupsWithState* lo que complica el diseño de la lógica de procesamiento. Otro problema derivado del uso de *flatMapGroupsWithState* es que ya no está disponible en Python por lo que quienes trabajan con este lenguaje tienen que elaborar sus consultas a través de Java/Scala.

La ilustración 24 muestra un ejemplo de uso de ventana de sesión. Se puede observar la simplicidad del uso de la nueva funcionalidad *session_window*.

```
# session window in Python
windowedCountsDF = \
        eventsDF \
                .withWatermark("eventTime", "10 minutes") \
                .groupBy("deviceId", session_window("eventTime", "5 minutes")) \
                .count()
```

Ilustración 24 – Definición de Ventana de Sesión

4. Apache Flink

En este capítulo se desarrolla Apache Flink como motor de procesamiento de streaming. En los últimos años este framework ha aumentado su popularidad, lo que lo constituye como una de las opciones elegidas cuando se necesita un motor de procesamiento distribuido de streaming.

4.1. Introducción

En el año 2010, se inició el proyecto de investigación "Stratosphere: Information Management on the Cloud" dirigido por Volker Markl y financiado por la Fundación Alemana de Investigación, como una colaboración de varias universidades europeas: Technical University Berlin, Humboldt-Universität zu Berlín y Hasso-Plattner-Institut Potsdam.

Flink comenzó como una bifurcación del motor de ejecución distribuida Stratosphere y se convirtió en un proyecto de Apache Incubator en marzo de 2014. Debido al creciente interés en el procesamiento de streaming, en diciembre de 2014, Flink fue aceptado como un Top Level Project de Apache Foundation. Hoy en día el framework está soportado y desarrollado por la start-up Ververica (anteriormente Data Artisans), una empresa fundada por los creadores originales de Apache Flink.

Apache Flink es un framework distribuido de procesamiento de flujos de datos ilimitados y limitados, de código abierto. Originalmente no deriva de un framework de procesamiento por lotes, sino que por el contrario nació como un proyecto específico para procesamiento nativo de streaming. Con el tiempo fue mutando para no solo trabajar con flujos sino también permitir el trabajo por lotes.

Flink ha sido diseñado para ejecutarse en todos los entornos distribuidos, es decir clusters de computadoras, lo que le brinda la capacidad de realizar cálculos a la velocidad de la memoria y en cualquier escala.

Se basa en Api intuitivas y expresivas para implementar aplicaciones de procesamiento de streaming con estado.

4.1.1. Escalabilidad

Flink está diseñado para ejecutar aplicaciones de streaming con estado, a cualquier escala. La idea central consiste en dividir las aplicaciones en tareas simples que se pueden distribuir y paralelizar, ejecutándose simultáneamente en un cluster. Por esta razón, una aplicación puede aprovechar cantidades prácticamente ilimitadas de CPU, memoria principal, disco y entrada/salida de red.

Además, Flink mantiene fácilmente un estado de aplicación muy grande. Su algoritmo de puntos de control asíncrono e incremental garantiza un impacto mínimo en las latencias de procesamiento y al mismo tiempo, garantiza la coherencia de estado con semántica de procesamiento exactamente una vez.

4.1.2. Procesamiento en memoria

Las aplicaciones con estado en Flink están diseñadas para realizar procesamiento en memoria, lo cual brinda un altísimo rendimiento. Además de la altísima velocidad, el procesamiento en memoria brinda un beneficio adicional, ya que el estado de las tareas siempre se mantiene en memoria, y solo si el tamaño del estado excede la memoria disponible, emplea estructuras de datos en disco de acceso eficiente.

Por lo tanto, cuando se ejecutan tareas se realizan todos los cálculos accediendo al estado local, a menudo en memoria, lo que produce latencias de procesamiento muy bajas.

Flink garantiza la consistencia del estado exactamente una vez en caso de fallas mediante la verificación periódica y asíncrona del estado local en memoria o en almacenamiento persistente.

4.2. Arquitectura de Flink

Desde el punto de vista arquitectónico Apache Flink se construye sobre un núcleo central conocido como *Runtime Distributed Streaming Dataflow*. El núcleo reúne la funcionalidad básica que Flink proporciona, tanto para procesamiento de streams como para procesamiento por lotes. En el núcleo se encuentran las capacidades para procesamiento distribuido, gestión de memoria, tolerancia a fallas.

Sobre el núcleo se sitúan dos Api, una para para gestionar el procesamiento por lotes denominada *DataSetAPI* y otra para el procesamiento de streaming conocida como *DataStreamAPI*. Esta capa es muy importante ya que es la que permite la interacción entre los usuarios y el núcleo central.

Sobre la capa de Api se sitúan librerías específicas que permiten el trabajo con Machine Learning, procesamiento de grafos por medio de Gelly y manejo de tablas SQL. Estas librerías son las básicas, pero existe una creciente comunidad de usuarios y colaboradores que extienden las funcionalidades que Flink proporciona.

En la ilustración 25 se observa la arquitectura de Apache Flink donde el núcleo ocupa el lugar central y sirve de cimiento para las Api y librerías.

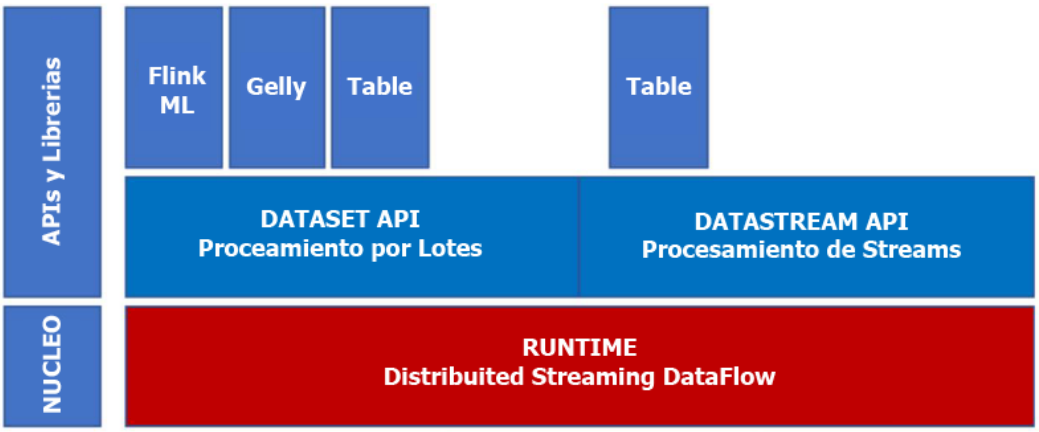

Ilustración 25 – Arquitectura de Apache Flink

4.3. Despliegue de Flink

La filosofía que sigue Flink es especializarse en el cómputo distribuido, paralelo y con estado, por lo que no se enfoca directamente de detalles específicos del despliegue como el almacenamiento. En su lugar se enfoca en permitir la interacción con otros protagonistas de ecosistemas distribuidos que brindan esa funcionalidad.

Hay tres formas de implementar *Flink*:

- Modo Local: esta manera de implementación permite la instalación y operación en una sola computadora corriendo sobre la *Java Virtual Machine*. Si bien en modo local se permite el funcionamiento e interacción con Flink, su principal característica, proporcionar procesamiento distribuido está condicionada. El modo local es ampliamente utilizado en ambientes de aprendizaje, de desarrollo y de testeo de aplicaciones.
- Modo Cluster: en la implementación en modo cluster se proporcionan varias alternativas, entre ellas:
    - Modo independiente.
    - Yarn.
    - Apache Mesos.
    - Apache Tez.
- Modo Cloud: este modo está desarrollado para permitir cómputo distribuido en la nube, y sus principales alternativas son:
    - Google GCE (*Google Compute Engine*).
    - Amazon EC2.
    - IBM Docker Cloud.

Apache Flink tiene múltiples opciones desde donde se puede leer y escribir datos. A continuación, se muestra una lista básica de opciones de almacenamiento de datos:

- Hadoop HDFS.
- Sistemas de archivo locales.
- Amazon S3.
- Motores de Bases de datos SQL:
    - MySQL.
    - Oracle.
    - MS SQLServer.
- MongoDB.
- HBase.
- Apache Kafka.
- Apache Flume.

La ilustración 26 resume los aspectos de despliegue y gestión de almacenamiento de *Apache Flink*.

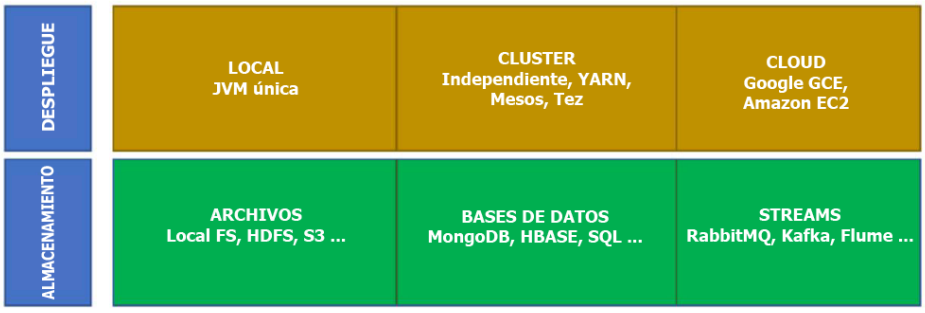

Ilustración 26 – Alternativas de despliegue de Apache Flink

4.4. Api de Lenguajes de Flink

Es importante mencionar que las Api para los lenguajes de programación soportados aún sigue en pleno desarrollo. Por esta situación, existen Api de lenguajes que están más desarrollados, como es el caso de Java y Scala. La Api de Python esta aun en desarrollo, por lo que existe funcionalidad disponible para Java y Scala que todavía no ha sido desarrollada para Python.

A continuación, se resumen los lenguajes soportados por la Api:

- *Java* y *Scala*: Flink está escrito en Java y en Scala. Esto los constituye como las opciones más desarrolladas y probadas.
- *Python*: el creciente uso de Python en comunidades de Big Data favoreció el desarrollo de la Api para este lenguaje.
- *SQL*: Flink admite la interacción en lenguaje SQL.

4.5. Niveles de Abstracción en Flink

Flink tiene distintos niveles de abstracción para el desarrollo de aplicaciones. La ilustración 27 muestra los distintos niveles de abstracción. En la parte inferior se encuentra el bloque de construcción de bajo nivel, este se encarga de brindar las funciones básicas para manejar flujos, estados, hora del evento entre otras características. Por encima de este se sitúan las core Api, el DSL declarativo y el nivel superior de más alto nivel SQL.

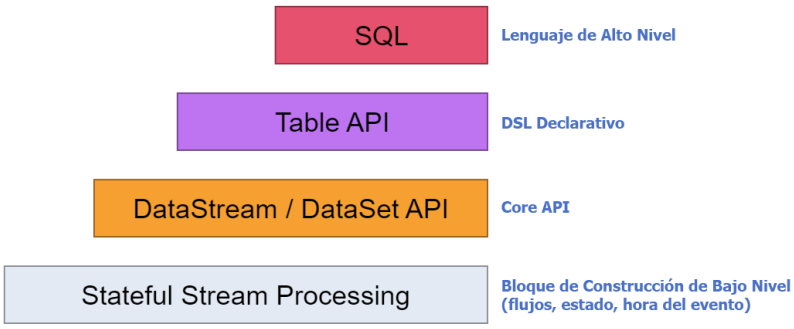

Ilustración 27 – Niveles de Abstracción en Flink

4.5.1. SQL

El nivel de abstracción más alto que ofrece Flink es *SQL*. El lenguaje nativo de motores de bases de datos relacionales es un mecanismo de alto nivel que permite expresar el manejo de datos de manera transparente sin exponer los mecanismos de implementación, por lo que constituye una poderosa herramienta disponible en Flink. El soporte SQL de Apache Flink está basado en *Apache Calcite* el cual implementa el estándar SQL.

Esta abstracción es similar a *Table Api* tanto en semántica como en expresividad, pero representa los programas como expresiones de consulta SQL. La abstracción *SQL* interactúa estrechamente con *Table Api*, y las consultas *SQL* se pueden ejecutar sobre tablas definidas en *Table Api*.

4.5.2. Table Api

Table Api es un DSL declarativo centrado en tablas, que pueden cambiar de forma dinámica (al representar flujos). *Table Api* sigue el modelo relacional (extendido) donde las tablas tienen un esquema adjunto (similar a las tablas de bases de datos relacionales) y la Api ofrece operaciones similares, como seleccionar, proyectar, unir, agrupar, agregar, etc.

Los programas desarrollados con *Table Api* definen de forma declarativa que operación lógica debe realizarse en lugar de especificar exactamente como se ve el código de la operación. Aunque *Table Api* es extensible por varios tipos de funciones definidas por el usuario, es menos expresiva que las *Core Api*, y más conciso de usar. Esto se traduce en menos cantidad de líneas de código a escribir.

Los programas de *Table Api* antes de ser ejecutados pasan por un optimizador que aplica reglas de optimización antes de realizar la ejecución.

4.5.3. DataStream / DataSet Api

Las Api *DataStream* para flujos ilimitados y *DataSet* para conjuntos de datos limitados constituyen las abstracciones principales de Flink. En la práctica, muchas aplicaciones no necesitan ser escritas usando abstracciones de bajo nivel, en cambio, se puede programar contra las Api principales dependiendo si la tarea a realizar es sobre procesamiento de streaming o por lotes.

Estas Api fluidas ofrecen los componentes básicos comunes para el procesamiento de datos, como diversas formas de transformaciones, uniones, agregaciones, ventanas, estados, etc. especificados por el usuario. Los tipos de datos procesados en estas Api se representan como clases en los respectivos lenguajes de programación.

La función de proceso de bajo nivel se integra con la Api de *DataStream*, lo que hace posible utilizar la abstracción de bajo nivel según sea necesario. La Api de *DataSet* ofrece primitivas adicionales en conjuntos de datos limitados, como bucles/iteraciones.

4.5.4. Stateful Stream Processing

La abstracción de nivel más bajo simplemente ofrece un procesamiento de flujo oportuno y con estado. Está embebida en la Api de *DataStream* a través de *ProcessFunction*. *ProcessFunction* es el bloque de construcción de más bajo nivel para el procesamiento de streaming y se encarga de proporcionar acceso a los componentes básicos de todas las aplicaciones de streaming (acíclicas):

- Eventos (elementos de flujo).
- Estado (tolerante a fallas, consistente, solo en flujo con clave).
- Temporizadores (tiempo de evento y tiempo de procesamiento, solo en transmisión codificada).

Por todo eso, *Stateful Stream Processing* permite a los usuarios procesar libremente eventos de uno o más flujos, proporciona un estado consistente y tolerante a fallas. Además, los usuarios pueden registrar el tiempo de evento y las devoluciones de llamada del tiempo de procesamiento, lo que permite que los programas realicen cálculos sofisticados.

4.6. Funcionamiento de Flink

En toda configuración de Flink están presentes cuatro componentes diferentes que trabajan de manera coordinada para ejecutar aplicaciones de streaming y por lotes. Estos componentes son *JobManager*, *ResourceManager*, *TaskManager* y *Dispatcher*. Debido a que Flink se implementa en Java y Scala, todos los componentes se ejecutan en la Máquina Virtual Java (JVM).

JobManager es el proceso que orquesta y controla la ejecución de una aplicación, cada aplicación está controlada por un *JobManager* diferente. El JobManager recibe una aplicación para su ejecución. La aplicación consta de un *JobGraph*, un grafo de flujo de datos lógico y un archivo JAR que agrupa todas las clases, bibliotecas y otros recursos necesarios. *JobManager* convierte *JobGraph* en un grafo de flujo de datos físico llamado *ExecutionGraph*, que consta de tareas que se pueden ejecutar en paralelo. *JobManager* solicita los recursos necesarios (ranuras de TaskManager) para ejecutar las tareas desde *ResourceManager*. Una vez que recibe suficientes ranuras de *TaskManager*, distribuye las tareas de *ExecutionGraph* a los *TaskManagers* que las ejecutan. Durante la ejecución, *JobManager* es responsable de todas las acciones que requieren una coordinación central, como la coordinación de los puntos de control.

Flink presenta múltiples *ResourceManagers* para diferentes entornos y proveedores de recursos, como YARN, Mesos, Kubernetes e implementaciones independientes. *ResourceManager* es responsable de administrar las ranuras de *TaskManager*, la unidad de procesamiento de recursos de Flink. Cuando un *JobManager* solicita ranuras de *TaskManager*, *ResourceManager* le indica a un *TaskManager* con ranuras inactivas que se las ofrezca al *JobManager*. Si *ResourceManager* no tiene suficientes ranuras para cumplir con la solicitud de *JobManager*, *ResourceManager* puede hablar con un proveedor de recursos para aprovisionar contenedores en los que se inician los procesos de *TaskManager*. El *ResourceManager* también se encarga de finalizar los *TaskManagers* inactivos para liberar recursos informáticos.

Los *TaskManager* son los procesos de trabajo de Flink. Por lo general, hay varios de ellos que se ejecutan en una configuración de Flink. Cada *TaskManager* proporciona un cierto número de ranuras. El número de ranuras limita el número de tareas que puede ejecutar un *TaskManager*. Una vez iniciado, un administrador de tareas registra sus ranuras en el *ResourceManager*. Cuando lo indica el *ResourceManager*, el *TaskManager* ofrece uno o más de sus espacios a un JobManager. El *JobManager* puede entonces asignar tareas a las ranuras para ejecutarlas. Durante la ejecución, un *TaskManager* intercambia datos con otros *TaskManager* que ejecutan tareas de la misma aplicación.

Dispatcher corre entre la ejecución de trabajos y proporciona una interfaz REST para enviar aplicaciones para su ejecución. Una vez que se envía una aplicación para su ejecución, se inicia un *JobManager* y se entrega la aplicación. La interfaz REST permite que el *Dispatcher* sirva como punto de entrada HTTP a los clusters que están detrás de un firewall. El *Dispatcher* también ejecuta un panel web para proporcionar información sobre las ejecuciones de trabajos, dependiendo de cómo se envíe una aplicación para su ejecución.

La ilustración 28 es un esquema de alto nivel para visualizar las responsabilidades e interacciones de los componentes de una aplicación Flink.

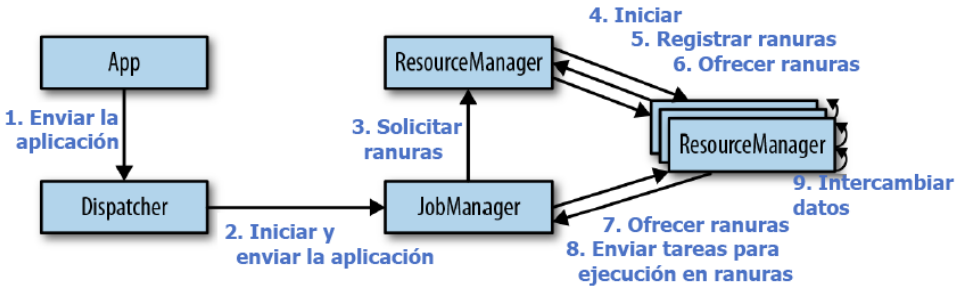

Ilustración 28 – Interacción de Componentes de una aplicación Flink

Según el entorno (YARN, Mesos, Kubernetes, cluster independiente) se pueden omitir algunos pasos, o los componentes se pueden ejecutar en el mismo proceso de JVM. Por ejemplo, en una configuración independiente (una configuración sin un proveedor de recursos), *ResourceManager* solo puede distribuir los espacios de los *TaskManagers* disponibles y no puede iniciar nuevos *TaskManagers* por sí solo.

El despliegue de aplicaciones Flink se puede realizar en estilo framework o en estilo librería. En el estilo framework las aplicaciones Flink se empaquetan en un archivo JAR y un cliente las envía a un servicio en ejecución. El servicio puede ser un *Dispatcher*, un *JobManager* o un *ResourceManager* de YARN. En cualquier caso, hay un servicio en ejecución que acepta la aplicación Flink y asegura su ejecución. Si la aplicación se envió a *JobManager*, inmediatamente comienza a ejecutar la aplicación. En cambio, si la solicitud se envió a un *Dispatcher* o YARN *ResourceManager*, activará un *JobManager* y entregará la aplicación, y recién *JobManager* comenzará a ejecutar la aplicación. El estilo framework es el enfoque tradicional para el despliegue de aplicaciones por medio de un cliente o un servicio en ejecución.

En el despliegue estilo librería, la aplicación Flink se incluye en una imagen de contenedor específica de la aplicación, como una imagen *Docker*. La imagen también incluye el código para ejecutar *JobManager* y *ResourceManager*. Cuando se inicia un contenedor desde la imagen, se inicia automáticamente *ResourceManager* y *JobManager,* y se envía el trabajo agrupado para su ejecución. Se utiliza una segunda imagen independiente del trabajo para implementar contenedores de *TaskManager*. Un contenedor que se inicia desde esta imagen inicia automáticamente un *TaskManager*, que se conecta al *ResourceManager* que registra sus ranuras.

Por lo general, un gestor de recursos externo como *Kubernetes* se encarga de iniciar las imágenes y se asegura de que los contenedores se reinicien en caso de falla. En este estilo de despliegue no hay un servicio Flink en ejecución, en su lugar, se incluye Flink como una librería junto con la aplicación en una imagen de contenedor. Este modo de despliegue es común para las arquitecturas de microservicios.

4.7. Procesamiento de Streaming en Flink

Es importante mencionar que desde sus orígenes el procesamiento de streaming constituyó el centro de atención del equipo de desarrollo de Apache Flink, por lo que este aparatado describe las funcionalidades centrales de Flink.

4.7.1. Esencia del procesamiento de streaming

Las aplicaciones en Flink se componen de flujos de datos de streaming, estos se inician cuando por medio de un operador se lee una fuente de datos. Una vez que se comienza a recibir el flujo de datos se puede aplicar operadores de transformación definidos por el usuario para manipular los datos. Esto conduce a obtener resultados, los que por medio de un operador de sumidero se encargan de escribir resultados en un medio de salida determinado. Estos flujos de datos forman grafos dirigidos que comienzan con una o más fuentes y terminan en uno o más sumideros.

En la ilustración 29 se puede observar una situación típica de procesamiento de streaming. Se señala el código fuente que corresponde a cada propósito, en este caso, leer fuente de datos, aplicar operadores de transformación y escribir datos salientes.

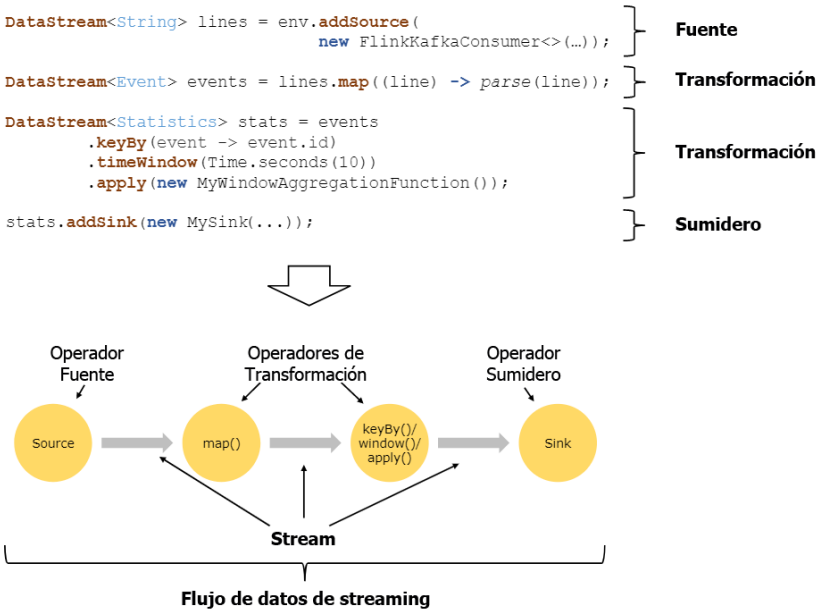

Ilustración 29 – Resumen de Procesamiento de streaming en Flink

Generalmente existe una correspondencia uno a uno entre las transformaciones en el programa y los operadores en el flujo de datos. A veces, sin embargo, una transformación puede constar de múltiples operadores.

Una aplicación puede consumir datos en tiempo real de fuentes de streaming, como colas de mensajes o registros distribuidos, como Apache Kafka o Kinesis. Pero Flink también puede consumir datos históricos acotados de una variedad de fuentes de datos. De manera similar, los flujos de resultados producidos por una aplicación se pueden enviar a una amplia variedad de sistemas que se pueden conectar como sumideros.

4.7.2. Procesamiento paralelo

Los programas en Flink son inherentemente paralelos y distribuidos. Durante la ejecución, un flujo de streaming tiene una o más particiones y cada operador tiene una o más subtareas. Las subtareas del operador son independientes entre sí y se ejecutan en diferentes subprocesos y posiblemente en diferentes máquinas o contenedores.

El número de subtareas del operador es el paralelismo de ese operador en particular. Diferentes operadores del mismo programa pueden tener diferentes niveles de paralelismo.

Los flujos pueden transportar datos entre dos operadores siguiendo dos patrones:

- Patrón uno a uno (one-to-one o reenvío).
- Patrón de redistribución (redistributing).

La ilustración 30 muestra el código de una aplicación Flink en una vista condensada, e incluye la vista paralelizada de procesamiento que el motor de procesamiento de streaming realiza.

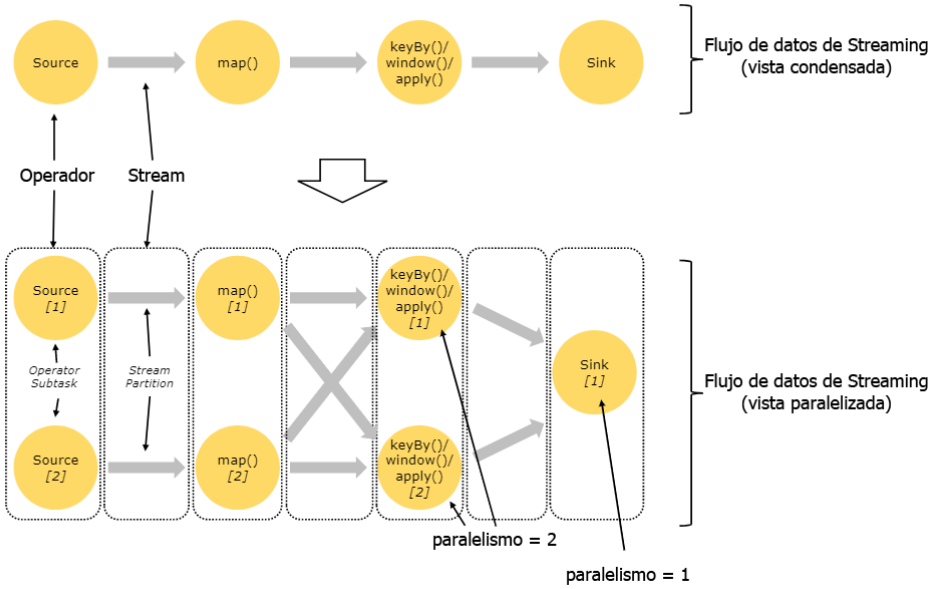

Ilustración 30 – Vista condensada y paralelizada de procesamiento

Los flujos que siguen el patrón uno a uno (por ejemplo, entre los operadores Fuente y map()) conservan la partición y el orden de los elementos. Eso significa que la subtarea[1] del operador map() verá los mismos elementos, en el mismo orden en que fueron producidos por la subtarea[1] del operador fuente.

Los flujos que siguen el patrón de redistribución (como entre map() y keyBy/window, así como entre keyBy/window y Sink) cambian la partición de los flujos. Cada subtarea del operador envía datos a diferentes subtareas de destino, según la transformación seleccionada. Algunos ejemplos son keyBy() (que vuelve a particionar mediante el hash de la clave), broadcast() o rebalance() (que vuelve a particionar aleatoriamente). En un intercambio de redistribución, el orden entre los elementos solo se conserva dentro de cada par de subtareas de envío y recepción (por ejemplo, subtarea[1] de map() y subtarea[2] de keyBy/window). Entonces, por ejemplo, la redistribución entre keyBy/window y los operadores sumidero que se muestran introducen el no determinismo con respecto al orden en que los resultados agregados para diferentes claves llegan al sumidero.

4.7.3. Procesamiento con estado

Flink ha sido diseñado para poder soportar el manejo de estado en sus operaciones. Esto significa que la forma en que se maneja un evento puede depender del efecto acumulado de todos los eventos anteriores. El estado se puede usar para algo simple, como contar eventos por minuto

para mostrar en un tablero, o para algo más complejo, como funciones informáticas para un modelo de detección de fraude.

Una aplicación Flink se ejecuta en paralelo en un cluster distribuido, en ese escenario, las diversas instancias paralelas de un operador determinado se ejecutarán de forma independiente, en subprocesos separados y, en general, se ejecutarán en diferentes máquinas.

El conjunto de instancias paralelas de un operador con estado es efectivamente un almacén de clave-valor fragmentado. Cada instancia paralela es responsable de manejar eventos para un grupo específico de claves, y el estado de esas claves se mantiene localmente.

La ilustración 31 que se incluye a continuación, muestra un trabajo que se ejecuta con un paralelismo de dos en los primeros tres operadores, y termina en un sumidero que tiene un paralelismo de uno. El tercer operador tiene estado y puede ver que se está produciendo una mezcla de red completamente conectada entre el segundo y el tercer operador. Esto se hace para dividir la transmisión por alguna clave, de modo que todos los eventos que deben procesarse juntos puedan ser procesados juntos.

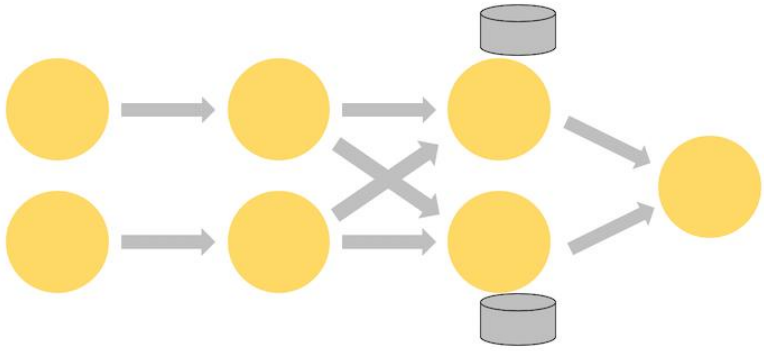

Ilustración 31 – Ejecución de trabajo en paralelo

En cuanto a la gestión del estado, siempre se accede al estado localmente, lo que ayuda a las aplicaciones Flink a lograr un alto rendimiento y una baja latencia. Se puede optar por mantener el estado en JVM o, si este es demasiado grande, en estructuras de datos en disco organizadas de manera eficiente, como se puede observar en la ilustración 32, donde las tareas pueden acceder al estado en memoria o hacerlo por medio de almacenamiento duradero eficiente en disco.

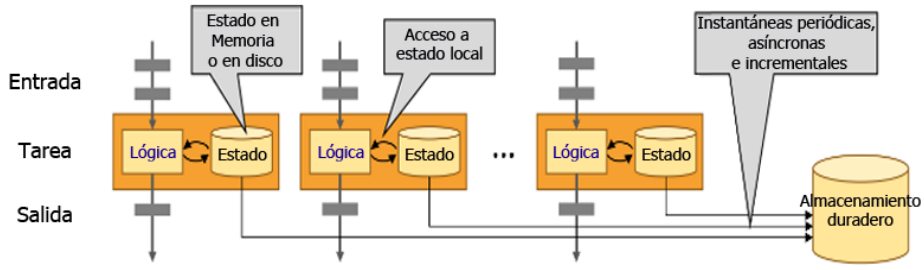

Ilustración 32 – Uso de almacén de estados

4.7.4. Tolerancia a Fallas

Flink es capaz de proporcionar una semántica *exactamente una vez* tolerante a fallas, a través de una combinación de instantáneas de estado y reproducción del streaming. Estas instantáneas

capturan el estado completo de la canalización distribuida, registrando las compensaciones en las colas de entrada, así como el estado en todo el grafo de trabajo que resultó de haber ingerido los datos hasta ese punto. Cuando ocurre una falla, las fuentes se rebobinan, se restaura el estado y se reanuda el procesamiento. Las instantáneas de estado se pueden capturar de forma asíncrona, de esta forma no se afecta el procesamiento en curso.

Flink periódicamente toma instantáneas persistentes de todo el estado en cada operador y copia estas instantáneas en almacenamiento duradero, como por ejemplo en un sistema de archivos distribuido. En caso de falla, Flink puede restaurar el estado completo de su aplicación y reanudar el procesamiento como si nada hubiera salido mal.

La ubicación donde se almacenan estas instantáneas se define mediante el *JobManager CheckpointStorage*. Hay dos implementaciones del *CheckpointStorage* disponibles, una de ellas conserva sus instantáneas de estado en un sistema de archivos distribuido, mientras que la otra usa el conjunto de JobManagers en JVM.

Para grandes aplicaciones se recomienda el almacén de estado duradero en disco, ya que admite un tamaño de estado muy grande, y esto es altamente durable. Para pequeñas aplicaciones o para entornos de prueba se puede optar por el *JobManager CheckpointStorage* que trabajan a nivel de JVM.

Flink utiliza una variante del algoritmo de Chandy-Lamport conocida como instantánea de barrera asíncrona, que en la terminología de Flink se la conoce con el nombre de instantáneas de estado. Cuando el *TaskManager* recibe la indicación del *CheckpointCoordinator* (parte del *JobManager*) para comenzar un checkpoint, todas las fuentes registran sus compensaciones e insertan barreras de checkpoint numeradas en sus flujos. Estas barreras fluyen a través del grafo de trabajo, indicando la parte del flujo, antes y después de cada checkpoint.

La ilustración 33 permite visualizar como los puntos de control (*checkpoint*) van segmentando el flujo de streams para tener un control completo respecto a que eventos se procesan. En este caso, el checkpoint N contendrá el estado de cada operador que resultó de haber consumido todos los eventos antes de la barrera del checkpoint N y ninguno de los eventos posteriores.

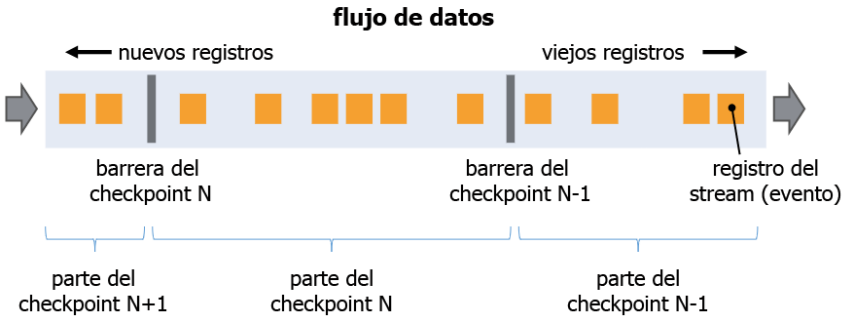

Ilustración 33 – Barreras y puntos de control (checkpoints)

Los backends de estado en Flink utilizan un mecanismo de copia en escritura para permitir que el procesamiento de secuencias continúe sin obstáculos mientras las versiones anteriores del estado se capturan de forma asíncrona. Solo cuando las instantáneas se hayan persistido de forma duradera, estas versiones anteriores del estado serán recolectadas por el *GarbageCollector*.

4.7.5. Semántica de procesamiento

Cuando las cosas van mal en una aplicación de procesamiento de streaming, es posible que se pierdan o se dupliquen resultados. En Flink se puede elegir la semántica de procesamiento para un cluster y para una aplicación.

Las alternativas de semántica de procesamiento que Flink brinda son:

- *Procesamiento como máximo una vez*: en este caso Flink no hace ningún esfuerzo por recuperarse de las fallas.
- *Procesamiento al menos una vez*: aquí Flink garantiza que no se pierda nada, pero pueden experimentarse resultados duplicados.
- *Procesamiento exactamente una vez*: en este escenario Flink garantiza que no se pierda nada ni que se duplique.

Dado que Flink se recupera de las fallas rebobinando y reproduciendo los flujos de datos de origen, cuando la semántica elegida es exactamente una vez, esto no significa que cada evento se procesará exactamente una vez. En cambio, significa que cada evento afectará el estado administrado por Flink exactamente una vez.

La alineación de barreras solo es necesaria para proporcionar garantías de exactamente una vez. Si no necesita esto, puede obtener algo más de rendimiento configurando Flink para usar el modo al menos una vez, que tiene el efecto de deshabilitar la alineación de barrera.

El caso de que se requiera la más alta precisión de procesamiento, Flink ofrece otra alternativa, procesamiento exactamente una vez de extremo a extremo. Esta es la alternativa que mayor exactitud ofrece. Para conseguir esta precisión deben cumplirse dos situaciones:

- Utilizar fuente de datos reproducibles.
- Utilizar sumideros idempotentes transaccionales.

4.7.6. Ventanas en Flink

Las ventanas para Flink constituyen la funcionalidad central del procesamiento de flujos infinitos. Las ventanas dividen el flujo en porciones de tamaño finito, sobre los cuales se puede aplicar cálculos.

En pocas palabras, se crea una ventana tan pronto como llega el primer elemento que debería pertenecer a esta ventana, y la ventana se elimina por completo cuando el tiempo (ya sea del evento o de procesamiento) pasa su marca de tiempo final más el especificado por el usuario como latencia permitida.

Además, cada ventana tendrá un activador y una función agregada. Esta función contiene el cálculo que se aplica al contenido de la ventana. El activador indica las condiciones que deben darse para aplicar la función. Una política de activación podría ser *cuando la cantidad de elementos en la ventana es más de 4* o *cuando la marca de agua pasa el final de la ventana*.

Las siguientes secciones describen las alternativas disponibles en Flink para la definición y uso de ventanas.

4.7.6.1. Ventanas con clave o sin clave

Inicialmente cuando se procesan agregaciones de ventana en Flink existen dos posibilidades a utilizar: ventanas sin clave y ventanas con clave.

En las ventanas sin clave, este tipo de ventana no codifica la transmisión. En este tipo de ventana el flujo original no se dividirá en múltiples flujos lógicos, y toda la lógica de procesamiento se realizará mediante una sola tarea, o sea, con un paralelismo de 1.

La ilustración 34 muestra la definición de ventana sin clave. Se puede observar parámetros opcionales indicados entre corchetes.

```
stream
       .windowAll(...)
      [.trigger(...)]
      [.evictor(...)]
      [.allowedLateness(...)]
      [.sideOutputLateData(...)]
       .reduce/aggregate/apply()
      [.getSideOutput(...)]
```

Ilustración 34 – Ventanas sin Clave

En las ventanas con clave, la ventana codifica la transmisión. En este caso, se divide el flujo infinito en flujos lógicos con clave.

En el caso de transmisiones con clave, cualquier atributo de sus eventos entrantes se puede usar como clave. Tener un flujo con clave permitirá que múltiples tareas realicen el cálculo en paralelo, ya que cada flujo con clave lógico se puede procesar independientemente del resto. Todos los elementos que hagan referencia a la misma clave serán enviados a la misma tarea paralela.

A continuación, la ilustración 35 ejemplifica la definición de una ventana con clave. Allí también se observan los parámetros opcionales que pueden utilizarse.

```
stream
       .keyBy(...)
       .window(...)
      [.trigger(...)]
      [.evictor(...)]
      [.allowedLateness(...)]
      [.sideOutputLateData(...)]
       .reduce/aggregate/apply()
      [.getSideOutput(...)]
```

Ilustración 35 – Ventanas con Clave

En los ejemplos anteriores se observa que al definir ventanas existen parámetros opcionales, entre ellos: *trigger*, *evictor*, *allowedLateness*.

Trigger: determina cuándo una ventana está lista para ser procesada por la función de ventana. Cada tipo de ventana tiene un valor predeterminado de *Trigger*.

La interfaz *Trigger* tiene cinco métodos que permiten reaccionar a diferentes eventos:

- *onElement()* se llama a este método para cada elemento que se agrega a una ventana.
- *onEventTime()* se llama a este método cuando se activa un temporizador de tiempo de evento registrado.
- *onProcessingTime()* se llama a este método cuando se activa un temporizador de tiempo de procesamiento registrado.
- *onMerge()* es relevante para activadores con estado y fusiona los estados de dos activadores cuando sus ventanas correspondientes se fusionan, por ejemplo, cuando se utilizan ventanas de sesión.
- *clear()* realiza cualquier acción necesaria al eliminar la ventana correspondiente.

Evictor: tiene la capacidad de eliminar elementos de una ventana después de que se activa el activador y antes y/o después de que se aplique la función de ventana. De forma predeterminada, todos los evictors preimplementados aplican su lógica antes de la función de ventana.

AllowedLateness: permite indicar el retraso permitido o marca de agua cuando se trabaja con ventanas con procesamiento con hora del evento. De forma predeterminada, los elementos tardíos se eliminan cuando la marca de agua se supera.

4.7.6.2. Ventanas giratorias

Las ventanas giratorias permiten procesar el streaming y asignan cada elemento a una ventana específica. Entre ventanas no hay superposición, esto indica que cada elemento recibido pertenece a una única ventana definida. Por ejemplo, si define una ventana con un tamaño de 5 minutos, se evaluará la ventana actual y se iniciará una nueva ventana cada cinco minutos.

La ilustración 36 ejemplifica la definición de ventanas giratorias. En el primer caso con procesamiento por hora del evento, en el segundo caso por tiempo de procesamiento y por último con procesamiento por hora del evento con un offset de 8 horas.

```
input = ...  # type: DataStream

# tumbling event-time windows
input \
    .key_by(<key selector>) \
    .window(TumblingEventTimeWindows.of(Time.seconds(5))) \
    .<windowed transformation>(<window function>)

# tumbling processing-time windows
input \
    .key_by(<key selector>) \
    .window(TumblingProcessingTimeWindows.of(Time.seconds(5))) \
    .<windowed transformation>(<window function>)

# daily tumbling event-time windows offset by -8 hours.
input \
    .key_by(<key selector>) \
    .window(TumblingEventTimeWindows.of(Time.days(1), Time.hours(-8))) \
    .<windowed transformation>(<window function>)
```

Ilustración 36 – Definición de Ventanas Giratorias

4.7.6.3. Ventanas deslizantes

Las ventanas deslizantes definen ventanas de un tamaño determinado, pero a diferencia de las giratorias, utilizan un parámetro adicional para especificar el deslizamiento de cada ventana. Este

deslizamiento indica el tiempo de superposición de una ventana con la siguiente. En este caso, los elementos que se procesan se pueden asignar a más de una ventana.

Los intervalos de tiempo se pueden especificar usando *Time.milliseconds(x)*, *Time.seconds(x)*, *Time.minutes(x)*, *Time.hours(x)*, etc.

La ilustración 37 ejemplifica la definición de ventanas deslizante, en el primer caso con procesamiento por hora del evento, luego usando tiempo de procesamiento y por último usando tiempo de procesamiento y el parámetro opcional offset con un valor de -8 horas.

```
input = ...  # type: DataStream

# sliding event-time windows
input \
    .key_by(<key selector>) \
    .window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5))) \
    .<windowed transformation>(<window function>)

# sliding processing-time windows
input \
    .key_by(<key selector>) \
    .window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5))) \
    .<windowed transformation>(<window function>)

# sliding processing-time windows offset by -8 hours
input \
    .key_by(<key selector>) \
    .window(SlidingProcessingTimeWindows.of(Time.hours(12), Time.hours(1), Time.hours(-8))) \
    .<windowed transformation>(<window function>)
```

Ilustración 37 – Definición de Ventanas Deslizantes

4.7.6.4. Ventanas de sesión

Las ventanas de sesión tienen la función de agrupar elementos por sesiones de actividad. Las ventanas de sesión no se superponen y no tienen una hora fija de inicio y finalización, a diferencia de las ventanas giratorias y deslizantes. En cambio, una ventana de sesión se cierra cuando no recibe elementos durante un cierto período de tiempo, es decir, cuando se produce un intervalo de inactividad.

Una ventana de sesión se puede configurar con un intervalo de sesión estático o con una función extractora de intervalos de sesión que define la duración del período de inactividad. Cuando expira este período, la sesión actual se cierra y los elementos subsiguientes se asignan a una nueva ventana de sesión.

En los fragmentos de código de la ilustración 38 se muestra la definición de ventanas de sesión.

```
input = ...  # type: DataStream

class MySessionWindowTimeGapExtractor(SessionWindowTimeGapExtractor):

    def extract(self, element: tuple) -> int:
        # determine and return session gap

# event-time session windows with static gap
input \
    .key_by(<key selector>) \
    .window(EventTimeSessionWindows.with_gap(Time.minutes(10))) \
    .<windowed transformation>(<window function>)

# event-time session windows with dynamic gap
input \
    .key_by(<key selector>) \
    .window(EventTimeSessionWindows.with_dynamic_gap(MySessionWindowTimeGapExtractor())) \
    .<windowed transformation>(<window function>)

# processing-time session windows with static gap
input \
    .key_by(<key selector>) \
    .window(ProcessingTimeSessionWindows.with_gap(Time.minutes(10))) \
    .<windowed transformation>(<window function>)

# processing-time session windows with dynamic gap
input \
    .key_by(<key selector>) \
    .window(DynamicProcessingTimeSessionWindows.with_dynamic_gap(MySessionWindowTimeGapExtractor())) \
    .<windowed transformation>(<window function>)
```

Ilustración 38 – Definición de Ventana de Sesión

Los espacios estáticos se pueden especificar utilizando *Time.milliseconds(x)*, *Time.seconds(x)*, *Time.minutes(x)*, etc. Los espacios dinámicos se especifican implementando la interfaz *SessionWindowTimeGapExtractor*.

Debido a que las ventanas de sesión no tienen un inicio y un final fijo, se evalúan de manera diferente que las ventanas giratorias y las deslizantes. Internamente, un operador de ventana de sesión crea una nueva ventana para cada registro que llega y fusiona las ventanas si están más cerca entre sí que el espacio definido. Para poder combinarse, un operador de ventana de sesión requiere un activador y una función de ventana de combinación, tales como *ReduceFunction*, *AggregateFunction* o *ProcessWindowFunction*.

4.7.6.4. Ventanas globales

Existe un tipo de ventana llamado global. Una ventana global asigna todos los elementos con la misma clave a la misma ventana global única. Este esquema de ventanas solo es útil si también especifica un activador personalizado. De lo contrario, no se realizará ningún cálculo, ya que la ventana global no tiene un final natural en el que podamos procesar los elementos agregados.

Este tipo de ventana se utiliza cuando se necesita disponer de una ventana única para todos los elementos de entrada que se vayan a recibir.

La ilustración 39 muestra el código necesario para definir una ventana global.

```
input = ...  # type: DataStream

input \
    .key_by(<key selector>) \
    .window(GlobalWindows.create()) \
    .<windowed transformation>(<window function>)
```

Ilustración 39 – Definición de Ventana Global

4.7.7. Funciones de Ventana

El objetivo de definir una ventana es poder aplicar un determinado procesamiento al flujo de streams recibidos. La primera tarea al definir una ventana en Flink es definir su tipo, posteriormente se debe especificar el cálculo a realizar definiendo la función de procesamiento. Esta función se utilizará para procesar los elementos de entrada de la ventana.

La función de ventana puede ser:

- *ReduceFunction*.
- *AggregateFunction*.
- *ProcessWindowFunction*.

Los dos primeros se pueden ejecutar de manera más eficiente porque Flink puede agregar de forma incremental los elementos a cada ventana a medida que estos llegan. Un *ProcessWindowFunction* obtiene un *Iterable* para todos los elementos contenidos en una ventana, e información adicional sobre la ventana.

Una transformación de ventana con *ProcessWindowFunction* no se puede ejecutar tan eficientemente como los otros casos porque Flink tiene que almacenar en búfer todos los elementos de una ventana internamente antes de invocar la función.

4.7.7.1. ReduceFunction

Una función *Reduce* especifica cómo se combinan dos elementos de entrada para producir un elemento de salida del mismo tipo. Flink usa la función *Reduce* para agregar incrementalmente los elementos de una ventana. En el ejemplo de la ilustración 40 se define una función *Reduce*.

```
input = ...  # type: DataStream

input \
    .key_by(<key selector>) \
    .window(<window assigner>) \
    .reduce(lambda v1, v2: (v1[0], v1[1] + v2[1]),
            output_type=Types.TUPLE([Types.STRING(), Types.LONG()]))
```

Ilustración 40 – Función Reduce

La definición de función *Reduce* del ejemplo anterior emplea una *función lambda* que suma los segundos campos de las tuplas recibidas, este procesamiento se realiza para todos los elementos en la ventana.

4.7.7.2. AggregateFunction

Una función de agregación (*Aggregate*) es una versión generalizada de una función *Reduce* que tiene tres tipos: un tipo de entrada (IN), un tipo de acumulador (ACC) y un tipo de salida (OUT).

El tipo de entrada es el tipo de los elementos de entrada en el stream y la función *Aggregate* tiene un método para agregar un elemento de entrada a un acumulador. La interfaz también tiene métodos para crear un acumulador inicial, para fusionar dos acumuladores en un acumulador y para extraer una salida (de tipo OUT) de un acumulador. Al igual que con la función *Reduce*, Flink agregará de forma incremental los elementos de entrada de una ventana a medida que lleguen.

La ilustración 41 define una función *Aggregate* que calcula el promedio del segundo campo de los elementos en la ventana.

```
class AverageAggregate(AggregateFunction):

    def create_accumulator(self) -> Tuple[int, int]:
        return 0, 0

    def add(self, value: Tuple[str, int], accumulator: Tuple[int, int]) -> Tuple[int, int]:
        return accumulator[0] + value[1], accumulator[1] + 1

    def get_result(self, accumulator: Tuple[int, int]) -> float:
        return accumulator[0] / accumulator[1]

    def merge(self, a: Tuple[int, int], b: Tuple[int, int]) -> Tuple[int, int]:
        return a[0] + b[0], a[1] + b[1]

input = ...  # type: DataStream

input \
    .key_by(<key selector>) \
    .window(<window assigner>) \
    .aggregate(AverageAggregate(),
               accumulator_type=Types.TUPLE([Types.LONG(), Types.LONG()]),
               output_type=Types.DOUBLE())
```

Ilustración 41 – Función Aggregate

4.7.7.3. ProcessWindowFunction

Una función *ProcessWindow* obtiene un *Iterable* que contiene todos los elementos de la ventana y un objeto *Context* con acceso a la información de tiempo y estado, esto le permite brindar más flexibilidad que otras funciones de ventana. Si bien esto brinda mayor flexibilidad, tiene un costo de rendimiento y genera mayor consumo de recursos, porque los elementos no se pueden agregar de forma incremental, sino que deben almacenarse en búfer internamente hasta que la ventana se considere lista para el procesamiento.

El código que se incluye en la ilustración 42 muestra la definición y uso de una función *ProcessWindow*. Esta función tiene como finalidad contar los elementos de la ventana. Además, la función agrega información sobre el procesamiento de la ventana a la salida.

```
input = ...  # type: DataStream

input \
    .key_by(lambda v: v[0]) \
    .window(TumblingEventTimeWindows.of(Time.minutes(5))) \
    .process(MyProcessWindowFunction())

# ...

class MyProcessWindowFunction(ProcessWindowFunction):

    def process(self, key: str, context: ProcessWindowFunction.Context,
                elements: Iterable[Tuple[str, int]]) -> Iterable[str]:
        count = 0
        for _ in elements:
            count += 1
        yield "Window: {} count: {}".format(context.window(), count)
```

Ilustración 42 – Función ProcessWindow

4.7.8. Fuentes de datos y sumideros

En la actualidad existen numerosos sistemas de almacenamiento de datos entre ellos sistemas de archivos, almacenes de objetos, bases de datos relacionales (basados en SQL), sistemas no SQL, almacenes clave-valor, índices de búsqueda, logs de eventos, colas de mensajes, etc. Apache Flink se enfoca en el procesamiento de streaming y procesamiento por lotes, por lo que debe cubrir una amplia diversidad de fuentes y sumideros de datos. Para ello proporciona una amplia librería de conectores que le permiten leer y escribir datos en sistemas externos. También brinda una Api para implementar conectores personalizados.

Poder leer o escribir datos en fuentes externas no es suficiente para un procesador de streaming, este necesita brindar garantías de consistencia y recuperación en caso de falla. Para poder proporcionar estos mecanismos es necesario integrar fuentes de datos y sumideros que se integren con el mecanismo de recuperación y los puntos de control (*checkpoint*) de Flink.

4.7.8.1. Consideraciones sobre las fuentes de datos

Para proporcionar coherencia de estado exactamente una vez, cada conector fuente debe poder establecer su posición de lectura, pudiendo marcar una posición cuando esta ya ha sido leída. Al tomar un punto de control, una fuente debe poder conservar sus posiciones de lectura y restaurar estas posiciones durante la recuperación. Si una aplicación consume datos de un origen que no puede almacenar y restablecer una posición de lectura, podría sufrir una pérdida de datos en caso de falla, y en este caso solo brindar garantías de consistencia como máximo una vez.

La combinación del mecanismo de punto de control, recuperación de Flink y fuentes reiniciables garantiza que una aplicación no perderá ningún dato. Sin embargo, la aplicación en este caso puede emitir resultados dos veces, porque todos los resultados que se han emitido después del último punto de control exitoso se emitirán nuevamente al realizar sobre la aplicación una recuperación de falla. Por lo tanto, las fuentes reiniciables y el mecanismo de recuperación de Flink no son suficientes para proporcionar garantías de una sola vez de extremo a extremo, incluso cuando el estado de la aplicación sea consistente exactamente una vez.

4.7.8.2. Consideraciones sobre los sumideros de datos

Para brindar garantías exactamente una vez de extremo a extremo se requiere conectores de sumidero que proporcionen escrituras idempotentes y escrituras transaccionales.

Una operación de escritura idempotente es aquella que, aunque se realice más de una vez, solamente dará como resultado la escritura una vez. A modo de ejemplo, insertar repetidamente el mismo par clave-valor en un hashmap es una operación idempotente porque la primera operación de inserción agrega el valor de la clave en el mapa y todas las inserciones siguientes no cambiarán el mapa porque ya contiene el par clave-valor. Las operaciones de escritura idempotentes son necesarias para las aplicaciones de streaming porque se pueden realizar varias veces sin cambiar los resultados. Por lo tanto, puede mitigar el efecto de los resultados repetidos causados por el mecanismo de puntos de control de Flink.

Como consecuencia, una aplicación que se basa en sumidero idempotente para lograr resultados exactamente una vez, debe garantizar que anula resultados ya escritos mientras se recupera de una falla.

El segundo enfoque para lograr una consistencia exactamente una vez de extremo a extremo se basa en escrituras transaccionales. La idea aquí es escribir solo esos resultados en un sistema de sumidero externo que se hayan calculado antes del último punto de control exitoso. Este comportamiento garantiza exactamente una vez de extremo a extremo porque, en caso de falla, la aplicación se restablece al último punto de control y no se emite ningún resultado al sistema receptor después de ese punto de control. Al escribir datos solo una vez que se completa un punto de control, el enfoque transaccional no sufre la inconsistencia de reproducción de las escrituras idempotentes. Sin embargo, esto agrega latencia, porque los resultados solo se vuelven visibles cuando se completa un punto de control.

4.7.8.3. Conectores incluidos

Flink proporciona un conjunto de conectores, tanto de fuentes y como de sumideros de datos. Las fuentes de datos predefinidas incluyen la lectura de archivos, directorios y sockets, y la ingesta de datos de colecciones e iteradores. Los sumideros de datos predefinidos admiten la escritura en archivos, en stdout y stderr, y en sockets.

A continuación, se mencionan los conectores de fuente de datos provistos por Flink:

- Apache Kafka.
- Amazon Kinesis Streams.
- Sistema de Archivos.
- RabbitMQ.
- Apache Nifi.
- Api de Twitter.
- Google PubSub.
- Netty.
- Apache ActiveMQ.

Los conectores de sumidero incluidos en Flink son:

- Apache Kafka.
- Apache Cassandra.
- Amazon Kinesis Streams.
- ElasticSearch.
- Sistema de Archivos.
- RabbitMQ.
- Apache Nifi.

- Google PubSub.
- Apache Flume.
- Redis.
- JDBC.
- Apache ActiveMQ.

Para poder usar estos conectores en una aplicación, generalmente se requieren componentes adicionales de terceros, por ejemplo, servidores para los almacenes de datos o colas de mensajes. Si bien los conectores de streaming enumerados en esta sección son parte del proyecto Flink y están incluidos en las source releases, no están incluidos en las distribuciones binarias.

Los conectores de fuentes y sumideros de datos constituyen uno de los aspectos de mayor importancia en la comunidad de Flink, por esta razón hay un gran compromiso comunitario en desarrollar nuevos conectores.

5. Trabajo Experimental

En este capítulo se describe el trabajo experimental desarrollado para realizar la comparación entre Apache Spark y Apache Flink.

5.1. Descripción del Trabajo Experimental

El trabajo consiste en desarrollar dos aplicaciones para el procesamiento de flujos de datos, ambas resolviendo el mismo problema. En la propuesta de trabajo final se planteó simular datos de streaming. Con el avance de la investigación se comenzó a experimentar accediendo a transmisiones de streaming en tiempo real, por lo que finalmente se pudo plasmar los resultados accediendo a datos reales de streaming, particularmente se accederá a flujos de texto de la red social Twitter.

Se construirá una aplicación que accederá a datos en tiempo real de tweets, filtrados por tema. Una vez recibidos los datos, los mismos deben ser preprocesados ya que los tweets pueden incluir urls, caracteres especiales e incluso emoticonos en su contenido. Los datos ya preprocesados serán origen de datos a dos aplicaciones, una de ellas realizará el procesamiento utilizando Apache Spark y la otra aplicación hará el procesamiento en Apache Flink. El caso de uso de dichas aplicaciones consiste en recibir el flujo de datos en tiempo real, dividir el mismo en palabras y realizar el conteo de cada palabra distinta, obteniendo como salida las palabras más utilizadas junto a su conteo, para un tema concreto.

Sobre las aplicaciones de procesamiento se realizarán pruebas que permitan medir el rendimiento de las mismas usando para en el caso de Spark ha herramienta *SparkTaskMetrics*. Por el lado de Apache Flink se utilizará la herramienta *Metrics*. Ambas herramientas de monitoreo son proporcionadas por cada uno de los frameworks. El uso de estas herramientas de monitoreo se complementará con la utilización de *JConsole*.

5.2. Hardware Utilizado

Todos los experimentos fueron realizados con el hardware que se detalla a continuación:

- Computadora Intel Core i7 11va generación, 16 GB memoria RAM, disco duro SSD NVME 1TB, placa red ethernet gigabit, placa video GeForce RTX 3070.
- Computadora Intel Core i5 7va generación, 8 GB memoria RAM, disco duro SSD 480GB, placa red ethernet gigabit, placa video Intel Graphics.
- Computadora Intel Core i5 6va generación, 8 GB memoria RAM, disco duro SSD 240GB, placa red ethernet gigabit, placa video Intel Graphics.
- Switch 5 bocas 10/100/1000.
- Conectividad internet FTTH 100Mbits.

Con el hardware detallado se procedió a armar un cluster, definiendo un nodo como gestionador del cluster y dos nodos como ejecutores. El gestionador del cluster implementa el pipeline de ingesta de datos, es decir se encarga de recibir el flujo continuo de datos y preprocesarlo. También en el gestionador se orquesta la ejecución del pipeline de procesamiento. Este pipeline es coordinado por el gestionador del cluster y el trabajo específico se ejecuta de manera distribuida en los nodos ejecutores.

5.3. Software Utilizado

Los componentes de software utilizados para realizar el desarrollo experimental son:

- Microsoft Windows 10 Professional.
- Apache Spark 3.2.0.
- Apache Flink 1.14.0.
- Apache Kafka 2.12.0.
- Java Virtual Machine 8 y 11.
- Phyton 3.7.
- Anaconda Navigator 2.1.0.
- Jupyter Notebook 6.3.0
- Spyder 4.2.5.
- Tweepy 4.10.0.
- KafkaPython 2.0.1.

5.4. Origen de Datos

Para la realización del presente Trabajo Final Integrador se experimentó con la utilización de un conjunto de datos de streaming en tiempo real. Se usó como fuente de datos Twitter, ya que provee una Api que permite el acceso a desarrolladores a un conjunto de funcionalidad por medio del protocolo seguro *Https*.

La funcionalidad a la que Twitter brinda acceso por medio de su Api incluye un mecanismo de autenticación seguro, acceso a tweets con metadatos, filtrado de tweets por temas, manejo de condiciones de excepción, posibilidad de acceso asíncrono, entre otras.

El acceso a la Api de Twitter devuelve los resultados en formato *JSON*. Estos datos serán decodificados para poder ser procesados.

5.4.1. Análisis del Origen de Datos

La red social Twitter se caracteriza porque de acuerdo a acontecimientos que ocurren, los usuarios reaccionan a éstos realizando publicaciones donde dan puntos de vista y opiniones sobre dichos acontecimientos. La temática seleccionada para la ingesta de datos se vincula al deporte futbol, uno de los deportes más populares en todo el mundo. Concretamente se incluyó las siguientes cadenas de búsqueda: “soccer”, “football”. La ingesta de datos se realizó durante la realización de partidos de UEFA Champions League, para obtener alta ingesta de datos.

Un análisis sobre la transmisión de la red social Twitter, muestra que para las cadenas de búsqueda ingresadas: “soccer”, “football” la tasa de entrada de registros es variable, con una media de 60 registros por segundo. En ciertos momentos la transmisión experimentó incrementos en la tasa de entrada que superan la media observada de 60 registros por segundo con picos de 120 a 150 registros, más del doble de la tasa media observada.

La ilustración 43 muestra la cantidad de registros recibidos durante la primera prueba realizada.

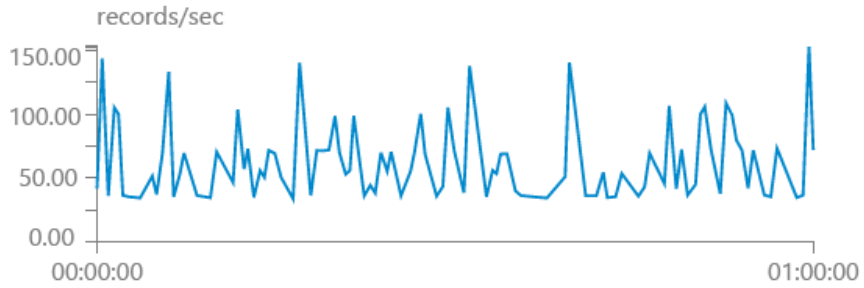

Ilustración 43 – Tasa de Entrada (Registros por segundo)

El tiempo total de prueba fue de 10hs. Durante las 5 pruebas realizadas se procesó 2201833 tweets, con una media de 220183 tweets por hora. La ilustración 44 resume duración de las pruebas y tweets recibidos durante las mismas.

Prueba	Duración	Tweets Recibidos
1	1:00 hs	226535
2	1:30 hs	327350
3	2:00 hs	431092
4	2:30 hs	551025
5	3:00 hs	665831

Ilustración 44 – Pruebas Realizadas

5.5. Pipelines de Procesamiento

En terminología específica de procesamiento de flujos, el procesamiento a realizar se denomina *stream processing pipeline* o tubería de procesamiento de flujos. La misma comienza con la conexión a la fuente de datos, posteriormente se describen las tareas de procesamiento a realizar, y por último se detalla el sumidero de datos salientes.

Para el caso de estudio se desarrollarán dos pipelines. El primer pipeline, será llamado de ingesta de datos, y tiene por objetivo obtener los datos en tiempo real, realizar el preprocesado de los datos recibidos y escribir los datos resultantes en un medio confiable que brinde tolerancia a fallas.

El segundo pipeline, llamado de procesamiento, leerá los datos que se van escribiendo en el medio confiable y hará el pasaje de los mismos al framework para su procesamiento. Como los frameworks de procesamiento son Apache Spark y Apache Flink se deben construir dos aplicaciones, donde cada una realiza el procesamiento en el framework correspondiente.

5.5.1. Pipeline de Ingesta de Datos

El pipeline de ingesta de datos tendrá como fuente de datos la red social Twitter, y se empleará la Api que dicha empresa proporciona a desarrolladores de software. La Api permite el acceso e interacción con la red social por medio de protocolo seguro *Https*. Específicamente para Python

existe una librería llamada Tweepy, la cual fue desarrollada para abstraer la complejidad de la Api de Twitter.

El primer paso para el acceso al flujo de datos en tiempo real es autenticar el usuario de conexión. Una vez realizada la autenticación, se establece la conexión a la fuente de datos, y los datos solicitados empiezan a fluir de manera ininterrumpida.

Es oportuno mencionar que las redes sociales se caracterizan por permitir que sus usuarios escriban libremente contenido, esto posibilita a los usuarios emplear todo tipo de caracteres incluyendo letras, números, signos, emoticonos, links a urls, etc cuando escriben tweets. Por esta razón es necesario llevar a cabo tareas de preprocesamiento de los datos.

Las tareas de preprocesamiento tienen por objetivo eliminar caracteres especiales, signos y urls. También se procede a eliminar caracteres en mayúscula como así también eliminar palabras vacías (stopwords). El resultante del preprocesamiento debe guardarse en un medio confiable, en este caso se optó por Apache Kafka.

Kafka es un proyecto de intermediación de mensajes de código abierto desarrollado por LinkedIn y donado a la Apache Software Foundation escrito en Java y Scala. El proyecto tiene como objetivo proporcionar una plataforma unificada, de alto rendimiento y de baja latencia para manipulación de fuentes de datos en tiempo real. Apache Kafka organiza los datos en temas y permite la publicación/suscripción a los mismos para escribir y para leer datos.

La ilustración 45 resume el pipeline de ingesta de datos.

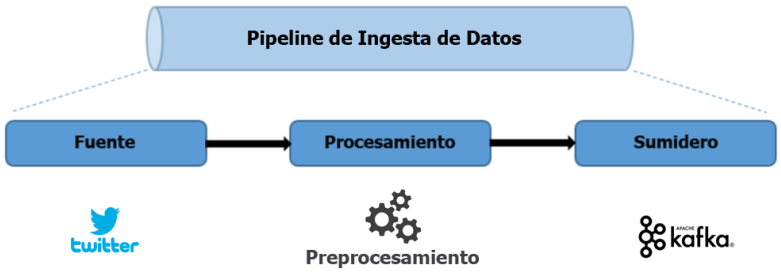

Ilustración 45 – Pipeline de Ingesta de Datos

5.5.2. Pipelines de Procesamiento

Debido a que se utilizará dos frameworks en la comparativa, se construirán dos pipelines de procesamiento. Cada uno de los cuales tendrá como fuente de datos Apache Kafka. El programa debe conectarse al servidor Kafka y suscribirse a un tema específico. De esta manera cuando el pipeline de ingesta reciba flujos de datos, estos podrán ser consumidos por el pipeline de procesamiento.

Una vez que se reciben los datos de Kafka, cada pipeline realizará el procesamiento en el framework correspondiente (Spark o Flink). El procesamiento consiste en dividir el flujo de datos recibido en palabras, realizar el conteo individual de palabras recibidas para obtener los totales por palabra.

Finalmente, los resultados obtenidos se escriben en consola para permitir la visualización de los mismos. La ilustración 46 resume el pipeline de procesamiento.

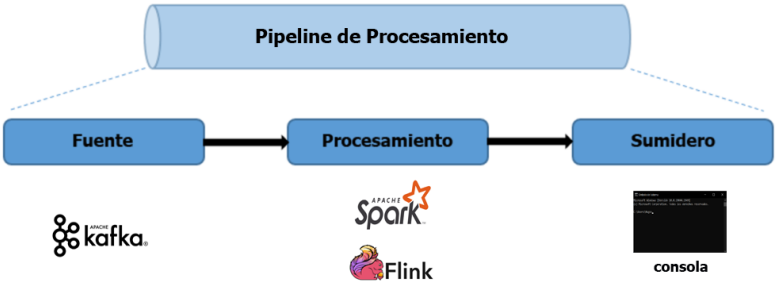

Ilustración 46 – Pipeline de Procesamiento

5.6. Código Fuente de Pipelines de Procesamiento

El código fuente de los pipelines de procesamiento se subió a un repositorio público de Git para que pueda ser accesible a quien esté interesado en los detalles de implementación. El link del repositorio es:

https://github.com/hugofajardo1/SparkFlinkComparison.git

A continuación, se detallan los archivos escritos en lenguaje Python y la finalidad de cada uno de ellos.

readTweetsWriteKafka.py

Este programa se conecta a twitter.com y descarga tweet de un tema específico. Preprocesa los tweets recibidos eliminando caracteres especiales, urls y stopwords. Finalmente, escribe los tweets ya preprocesados en un tema de Kafka. La ilustración 47 muestra el código fuente de la subclase MyStream de la librería Tweepy donde se leen los tweets, se llama a la función que realiza el preprocesamiento y se escribe en el productor de Kafka.

```
# Subclass Stream to read Tweets
class MyStream(tweepy.Stream):

    def __init__(self, *args):
        super().__init__(*args)

    def on_status(self, status):
        print(status.text)

    def on_data(self, data):
        try:
            msg = json.loads( data )
            clean_text = cleanText(msg['text'])
            future = producer.send(topic, clean_text.encode('utf-8'))
            producer.flush()
            return True
        except BaseException as e:
            print("Error on_data: %s" % str(e))
        return True

    def on_error(self, status):
        print(status)
        return True
```

Ilustración 47 - Lectura de Tweets y escritura en Kafka

sparkStreamingKafka.py

Este programa se conecta a *Apache Kafka* y lee el contenido del tema registrado. Luego procesa el contenido del tema en Spark con *Structured Streaming*. La ilustración 48 muestra el código fuente donde se define *SparkSession.* Luego se lee el contenido del tema de Kafka. Posteriormente se calculan los totales por palabra y se escriben los resultados en consola.

```
servidor_bootstrap = '192.168.0.2:9092'
topic = 'Twitter'

spark = SparkSession \
    .builder \
    .master("local[8]") \
    .appName("TwitterWordCount") \
    .config("spark.sql.streaming.metricsEnabled", True) \
    .config("spark.sql.shuffle.partitions", 5) \
    .getOrCreate()
spark.sparkContext.setLogLevel("ERROR")

lines = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", servidor_bootstrap) \
  .option("subscribe", topic) \
  .option("startingOffsets", "latest") \
  .load()

lines.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")

# Split the lines into words
words = lines.select(explode(split(lines.value, " ")).alias("word"))

# Generate running word count
wordCounts = words.groupBy("word").count().orderBy('count', ascending=False)

# Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Ilustración 48 - Procesamiento en Spark

flinkStreamingKafka.py

Este programa se conecta a *Apache Kafka* y lee el contenido del tema registrado. Luego procesa el contenido del tema en Flink. La ilustración 49 muestra el código fuente de la función TwitterProcessing, esta lee el tema de Kafka, calcula los totales por palabra y escribe los resultados en consola.

```
def TwitterProcessing():

    env = StreamExecutionEnvironment.get_execution_environment()
    env.set_runtime_mode(RuntimeExecutionMode.STREAMING)
    # write all the data to one file
    env.set_parallelism(1)

    env.add_jars("file:///C:/kafka/kafka_2.12-3.2.0/libs/flink-sql-connector-kafka-1.15.2.jar")

    kafka_props = {'bootstrap.servers': '192.168.0.2:9092', 'group.id': 'Twitter'}

    deserialization_schema = SimpleStringSchema()

    kafka_consumer = FlinkKafkaConsumer('Twitter',  deserialization_schema, kafka_props)

    kafka_consumer.set_start_from_latest()

    ds = env.add_source(kafka_consumer)

    def split(line):
        yield from line.split()

    # compute word count
    ds = ds.flat_map(split) \
           .map(word_munge) \
           .map(lambda i: (i.lower(), 1), output_type=Types.TUPLE([Types.STRING(), Types.INT()])) \
           .key_by(lambda i: i[0]) \
           .reduce(lambda i, j: (i[0], i[1] + j[1])) \


    ds.print()
    print("_____")

    # submit for execution
    env.execute()
```

Ilustración 49 - Procesamiento en Flink

5.7. Herramientas de Monitorio y Medición

La naturaleza de procesamiento en tiempo real de las aplicaciones de streaming requiere un constante monitoreo, por esta razón *Apache Spark* y *Apache Flink* proporcionan herramientas de monitoreo. La razón para el constante monitoreo es que diversos factores pueden afectar a una aplicación de streaming cuando se ejecuta en un ambiente de producción. A continuación, se detallan algunos problemas que pueden presentarse:

- Variación en la tasa de entrada de datos.
- Limitaciones de recursos: entre ellos el procesador, memoria disponible, capacidad de almacenamiento.
- Problemas de conectividad.
- Caída de nodos del cluster.
- Recuperación en caso de falla.

Cuando ocurren inconvenientes como los antes mencionados, las herramientas de monitoreo permiten rápidamente detectar estas situaciones y enviar alertas para que los equipos de trabajo puedan tratar los inconvenientes.

Tanto Spark como Flink poseen varias maneras de proporcionar acceso a métricas de rendimiento, entre ellas: acceso vía servidor http, acceso vía Api Rest, conexión a herramientas externas de monitoreo.

Para realizar las mediciones de rendimiento de ambos frameworks se utilizará las herramientas de métricas que cada uno de ellos proporciona y se complementará con el uso de *JConsole*.

JConsole es una herramienta gráfica de monitoreo para supervisar Java Virtual Machine (JVM) y aplicaciones Java en una máquina local o remota. Esta herramienta emplea funciones subyacentes de JVM para proporcionar información sobre rendimiento y consumo de recursos de las aplicaciones que se ejecutan en la plataforma Java mediante la tecnología *Java Management Extensions* (JMX). *JConsole* viene como parte de Java Development Kit (JDK).

6. Evaluación Comparativa

El trabajo de revisión y comparación entre los frameworks de procesamiento de flujos no se centrará únicamente en evaluar performance, también incluirá diferentes aspectos que se consideran cuando se necesita decidir que framework de procesamiento se va a utilizar.

Por ello la evaluación comparativa incluye los siguientes ítems:

- Facilidad de instalación y despliegue.
- Fuentes admitidas para el intercambio de datos.
- Lenguajes de programación soportados.
- Documentación disponible.
- Evaluación de rendimiento.

6.1. Facilidad de Instalación y Despliegue

Para evaluar facilidad de instalación y de despliegue se tendrá en cuenta la documentación que cada framework proporciona en su sitio web.

Ambos frameworks proporcionan documentación en sus sitios web. También permiten el acceso a descargas, tanto de versiones binarias como de código fuente.

El acceso al sitio web de cada framework es el siguiente:

- Spark: https://spark.apache.org/
- Flink: https://flink.apache.org/

La documentación para realizar la instalación y despliegue permite que ambos frameworks se puedan instalar y configurar en diversas plataformas sin inconvenientes. También incluyen opciones de configuración a tener en cuenta cuando se necesite desplegar ambientes de producción.

6.2. Fuentes de Datos Admitidas

Respecto a fuentes de datos admitidas, Spark concentra sus esfuerzos en soportar diversos formatos de archivos (orc, json, csv, text, avro), por lo que las aplicaciones escritas en Spark deben leer y producir resultados en dichos formatos. Spark también admite el uso de sockets para la ingesta de datos, aunque en el caso de esta fuente de datos se aclara que no se brindan garantías de tolerancias a fallas. Una mención especial tiene la interacción de Spark con Apache Kafka, esta comunicación está soportada de manera directa y nativa, lo que lo constituye como uno de los mecanismos más elegidos a la hora de realizar ingesta de datos en tiempo real.

Flink por su parte no pone el foco en formatos de archivo admitidos, sino que se concentra en proporcionar conectores a diversas fuentes de datos. Para las siguientes fuentes de datos ya se proporcionan conectores con la versión descargada:

- Amazon Kinesis Streams.
- Apache ActiveMQ.
- Apache Kafka.
- Apache Nifi.
- Conectores JDBC.

- Google PubSub.
- RabbitMQ.

En caso de que se necesite usar una fuente de datos cuyo conector no ha sido proporcionado, Flink brinda la documentación necesaria para que se pueda implementar dicho conector. Otra consideración importante es que el uso de sockets en Flink está soportado solamente para Java y Scala, no así para Python.

Si bien los dos frameworks adoptan diferentes enfoques a la hora de realizar ingesta y producción de resultados, estas diferencias de enfoque no generan inconvenientes a la hora de escribir aplicaciones que necesiten consumir y producir resultados.

6.3. Lenguajes de Programación soportados

En cuanto a lenguajes de programación, ambos frameworks soportan el uso de Java y Scala, como lenguajes principales. El hecho de que sean considerados lenguajes principales implica que sus Api son las más desarrolladas.

Ambos lenguajes también soportan el uso de SQL, esto abre las puertas de la escritura de aplicaciones de procesamiento de flujos a un grupo muy grande de desarrolladores, que vienen del mundo de bases de datos relacionales.

El lenguaje Python es soportado por los dos frameworks. En el caso de Spark tiene desarrollada la Api de Python casi en su totalidad. Flink por su parte sigue desarrollando la misma, por lo que aún existe funcionalidad actualmente no soportada para Python.

Spark además soporta el uso de R, esto abre las puertas del procesamiento de flujos a quienes trabajan en el mundo del cálculo matemático y de las ciencias de datos.

6.4. Documentación Disponible

En este apartado, ambos frameworks brindan documentación con cada una de las versiones disponibles. También facilitan el acceso a casos de ejemplo básicos.

Cuando la complejidad de procesamiento aumenta la situación cambia, ya que no siempre brindan ejemplos completos. En estos casos se debe recurrir a comunidades de usuarios.

Si bien ambas comunidades de usuarios están en pleno crecimiento, la comunidad de Spark es actualmente más numerosa y proporciona mayor cantidad de documentación, de ejemplos y resolución de problemas que la comunidad de Flink.

6.5. Evaluación de Rendimiento

La evaluación de rendimiento pretende evaluar ambos frameworks bajo las mismas condiciones, tanto a nivel de configuración de cluster, tareas de procesamiento a realizar y conjunto de datos de entrada. Spark y Flink permiten llevar a cabo optimizaciones que permiten mejorar el rendimiento, en este caso no se realizarán optimizaciones, con esto se pretende evaluar el rendimiento con la configuración proporcionada al instalar y desplegar el cluster.

Para evaluar el rendimiento de las aplicaciones que implementan el pipeline de procesamiento se utilizará las herramientas de monitoreo que cada framework proporciona y se complementará con la utilización de la herramienta *JConsole*.

La evaluación de rendimiento cubrirá los siguientes ítems:

- Uso de CPU.
- Uso de memoria.
- Cantidad de hilos de ejecución utilizados.
- Cantidad de clases cargadas.
- Latencia.
- Duración total del ciclo de procesamiento.

6.5.1. Consideraciones Generales de la Evaluación de Rendimiento

Para realizar la evaluación de rendimiento se ejecutaron en paralelo la aplicación de procesamiento desarrollada en Spark y la aplicación construida en Flink. El entorno de ejecución es un cluster formado por un nodo principal y dos nodos ejecutores. La razón para realizar la evaluación en paralelo es proporcionar a cada aplicación el mismo conjunto de datos en tiempo real, y realizar las mediciones de manera simultánea.

La cantidad de pruebas realizadas en simultáneo fueron 5. En cada una de las pruebas que se realizó, el tiempo total de ingesta de datos fue la misma cantidad de tiempo de duración de cada prueba. Durante la realización de las pruebas, los datos en bruto recibidos fueron preprocesados. En la etapa de preprocesamiento no se eliminaron tweets, solamente se limitó a eliminar caracteres especiales, signos, urls y stopwords.

La prueba inicial tuvo una duración de 1 hora. En las siguientes pruebas se fue incrementando en media hora el tiempo de cada prueba realizada. Culminando la prueba número 5 con una duración de 3 horas.

Para que el lector tenga una perspectiva visual que facilite la comprensión de los resultados se adjuntarán gráficos de cada uno de los aspectos evaluados. Se pudo observar que cada uno de los frameworks, para un aspecto medido, obtiene valores de rendimiento similares en cada una de las cinco pruebas realizadas. Por esta razón se decidió seleccionar aleatoriamente una de las pruebas realizadas para generar los gráficos que se adjuntarán. Los gráficos reúnen información del procesamiento durante un tiempo de 1 hora.

6.5.2. Uso de CPU

Al momento de iniciar el procesamiento se observó que la aplicación Spark consume entre 5.5% y 6% del tiempo del procesador. Luego se observa una baja en el uso de CPU a valores medios de 1.5% con picos máximos escasos entre 7% y 10%. Los valores mínimos de uso del procesador registrados fueron entre 0.4% y 0.5%. La ilustración 50 muestra el uso de CPU de Spark.

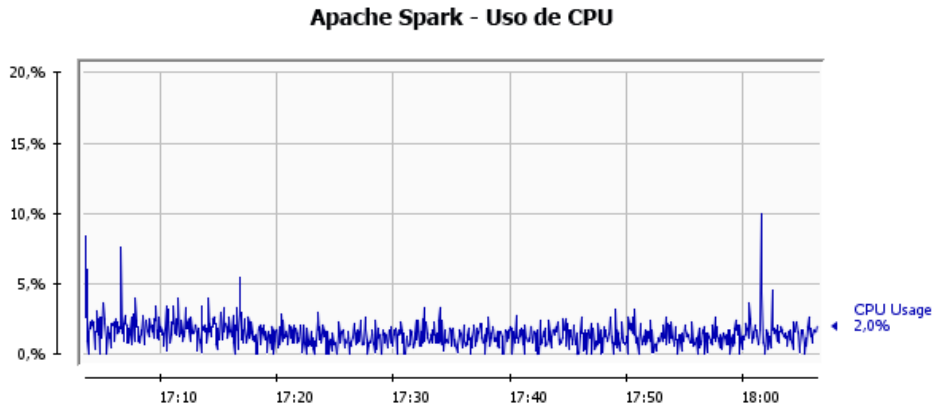

Ilustración 50 – Uso de CPU de Apache Spark

La aplicación Flink al momento de iniciar el procesamiento consume entre 0.6% y 0.8% del tiempo del procesador. Se pudo observar que estos valores varían muy poco a medida que la aplicación continúa en ejecución. La media observada de uso del procesador fue de 0.8% con picos máximos entre 2.8% y 3%, y valores mínimos registrados entre 0.2% y 0.3%. La ilustración 51 muestra el uso de CPU de Flink.

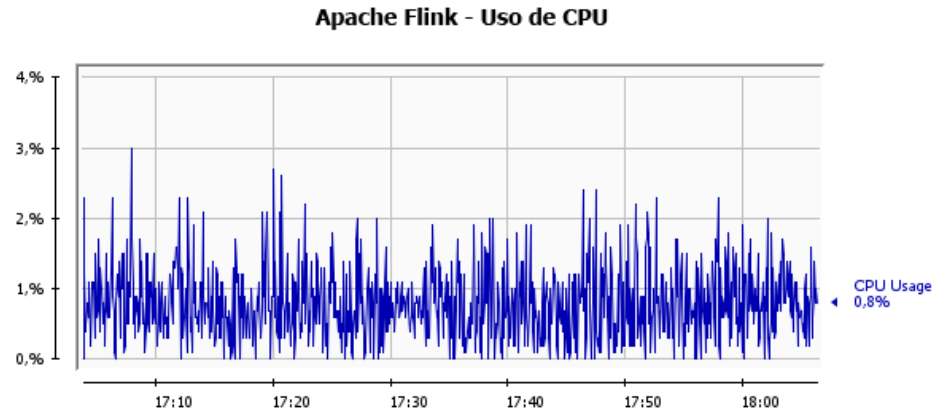

Ilustración 51 – Uso de CPU de Apache Flink

6.5.3. Uso de Memoria

El uso de memoria de la aplicación Spark inicialmente es entre 200Mb y 300Mb. Con el correr de la ejecución el uso de memoria baja y luego aumenta paulatinamente hasta alcanzar un valor máximo de 560Mb. En ese momento el consumo de memoria baja y se consigue el valor mínimo observado de 96Mb, y nuevamente vuelve a incrementar el uso de memoria con el correr de la ejecución hasta alcanzar nuevamente el valor máximo observado, y posteriormente vuelve a descender. Esta situación de incremento en el uso de memoria, posterior descenso, y nuevamente incremento, es cíclica y se observó en todas las pruebas realizadas. El consumo medio observado de memoria fue de 350Mb. La ilustración 52 muestra el uso de memoria de Spark.

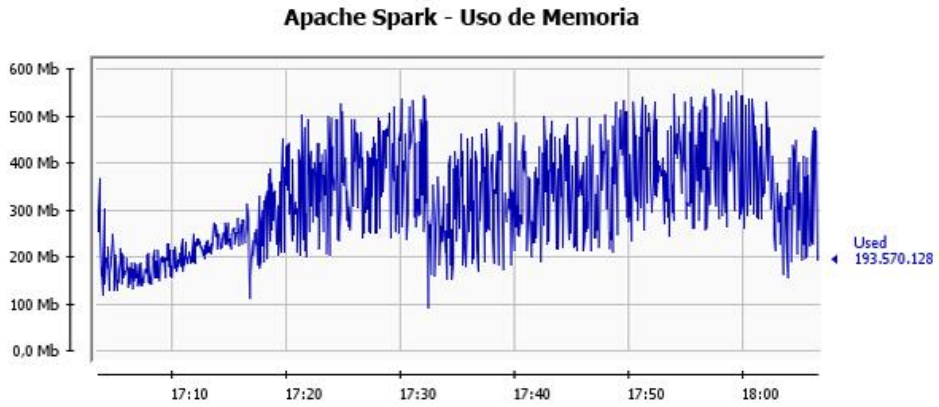

Ilustración 52 – Uso de Memoria de Apache Spark

El procesamiento de la aplicación Flink muestra inicialmente uso de memoria de entre 100Mb y 110Mb. Luego paulatinamente sube hasta valores máximos de entre 150Mb a 180Mb. Cuando alcanza esos valores máximos luego desciende abruptamente a 40Mb-50Mb. Inmediatamente después de producirse el descenso en el consumo de memoria a valores mínimos, el uso de memoria comienza a aumentar paulatinamente hasta alcanzar nuevamente valores máximos de entre 150Mb a 180Mb, cuando nuevamente se observa una baja a los valores mínimos observados. Esta situación de incremento en el uso de memoria, posterior descenso, y nuevamente incremento, es cíclica y se observó en todas las pruebas realizadas. El consumo medio observado de memoria fue de 120Mb. La ilustración 53 muestra el uso de memoria de Flink.

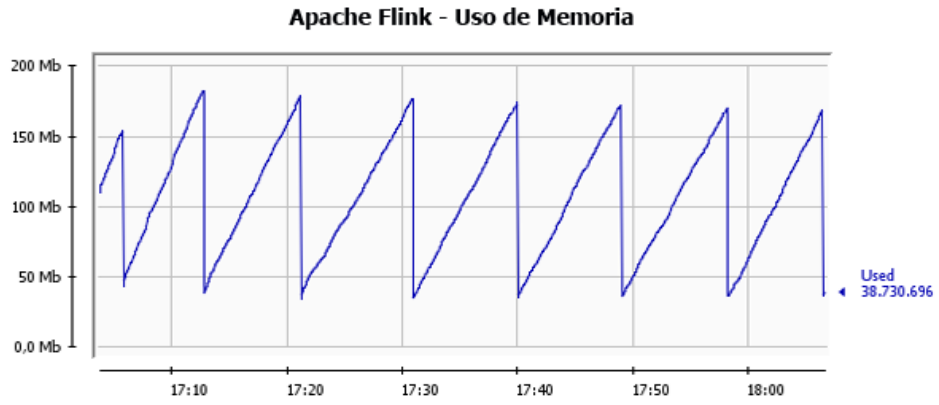

Ilustración 53 – Uso de Memoria de Apache Flink

En cuanto a cantidad total de memoria reservada, Spark reserva 950Mb en promedio por ejecución, mientras que Flink reserva 3.7Gb.

6.5.4. Cantidad de Hilos de Ejecución

Este apartado se encarga de monitorear la cantidad de subprocesos activos que cada aplicación y su framework de procesamiento emplean al ser ejecutados.

La aplicación Spark al iniciar el procesamiento abre 275 hilos de ejecución. Este número varía con el correr de la ejecución, en algunos momentos sube a 290 hilos y luego tiende a estabilizarse en una media de 277. La cantidad de hilos de ejecución solo bajó a 260-245 hilos en muy pocos

momentos. La ilustración 54 muestra la cantidad de hilos de ejecución de Spark durante el procesamiento.

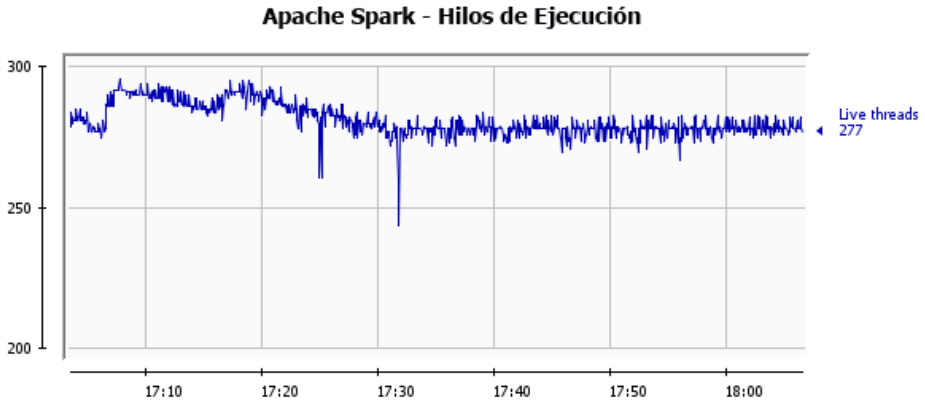

Ilustración 54 – Hilos de Ejecución de Apache Spark

La aplicación Flink cuando inicia el procesamiento emplea entre 60 y 70 hilos de ejecución. Pasados entre 2 y 3 minutos, la cantidad de subprocesos activos baja a valores entre 61 y 63 subprocesos y se estabiliza a lo largo de la ejecución. La ilustración 55 muestra la cantidad de hilos de ejecución de Flink a lo largo del tiempo.

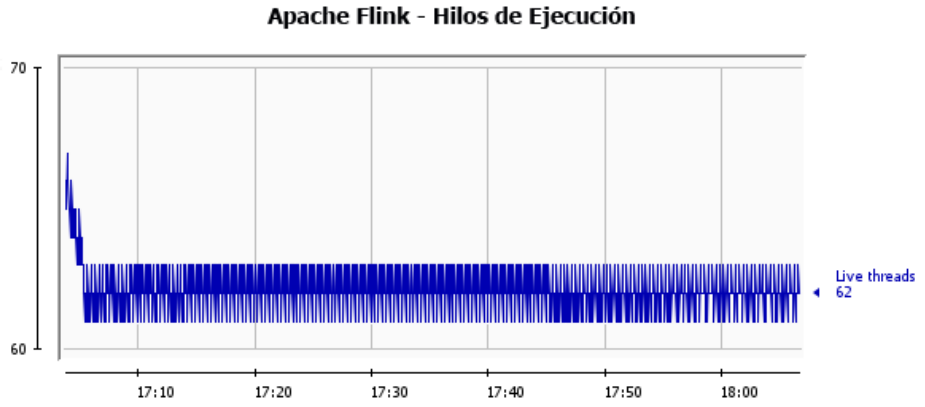

Ilustración 55 – Hilos de Ejecución de Apache Flink

6.5.5. Cantidad de Clases Cargadas

El apartado cantidad de clases cargadas permite monitorear el número de clases que cada aplicación y framework emplean al momento de ser ejecutados.

La aplicación Spark al momento de iniciar el procesamiento carga 20000 clases. Con el avance en la prueba, la cantidad de clases cargadas sube ligeramente a 20300 y luego 20500, posteriormente baja a 20470 y se mantiene estable en ese número. La ilustración 56 muestra la cantidad de clases cargadas de Spark.

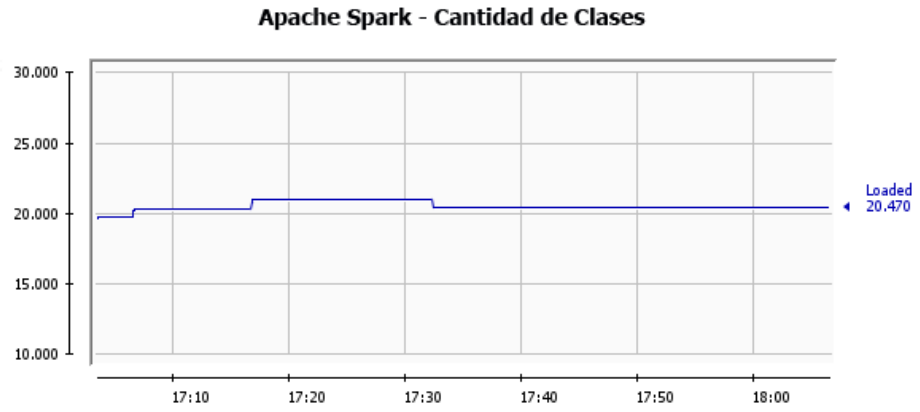

Ilustración 56 – Cantidad de Clases Cargadas de Apache Spark

La aplicación Flink por su parte carga 14200 clases al inicio del procesamiento. Luego comienza a subir lentamente la cantidad de clases cargadas hasta llegar a 14316. Cuando alcanza ese número, la cantidad de clases cargadas se estabiliza y a medida que la ejecución continua sigue una media de 14310 clases. La ilustración 57 muestra la cantidad de clases cargadas de Flink.

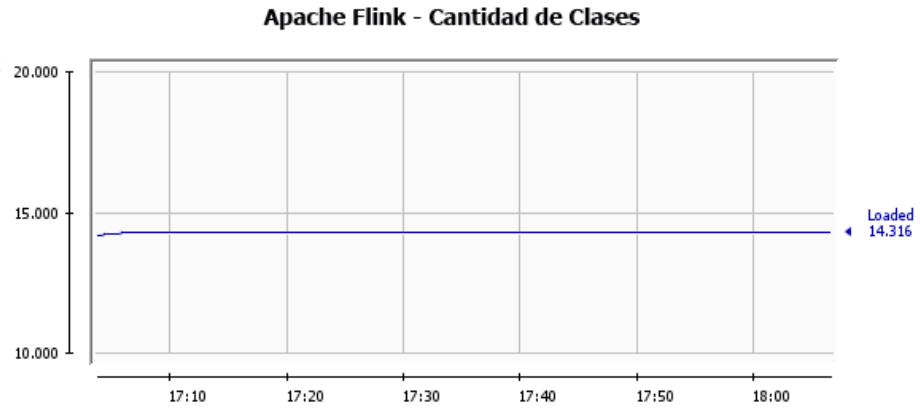

Ilustración 57 – Cantidad de Clases Cargadas de Apache Flink

6.5.6. Latencia

En secciones anteriores se indicó que, en sistemas de procesamiento de flujos, el término latencia se refiere al tiempo necesario para que el sistema reaccione ante la llegada de un evento o estímulo.

Los frameworks bajo estudio tienen diversos orígenes, Apache Spark comenzó como un motor de procesamiento distribuido por lotes y luego añadió capacidades de procesamiento de flujos. Spark divide el flujo de datos de entrada en pequeños microlotes, lo que genera un retraso en el procesamiento de cada estímulo recibido. Actualmente Spark ya admite el procesamiento continuo, sin emplear microlotes y con latencia teórica aproximada a 1ms. En este trabajo comparativo no se empleó ese enfoque de procesamiento, porque a la fecha de realización del presente trabajo, el modo continuo no admite el uso de agregaciones.

Apache Flink por el contrario nació como un motor de procesamiento de flujos, enfocando directamente su trabajo a responder de manera inmediata cuando se reciben datos de entrada, es decir con muy baja latencia.

La latencia de Spark inicialmente es de 1 segundo. Con el correr de la ejecución el framework ajusta el tamaño de cada microlote y la latencia baja considerablemente a 0.3 segundos, manteniéndose estable en ese registro. El registro mínimo de latencia fue de 0.2 segundos.

La aplicación Flink por su parte, al inicio de la ejecución presentó una latencia de entre 30 ms y 35 ms. A medida que la ejecución continúa, la latencia observada tiene variaciones mínimas con una media de 40ms. La mayor latencia observada por la aplicación Flink llegó a un valor de 51ms.

6.5.7. Duración Total del Ciclo de Procesamiento

La duración total del ciclo de procesamiento considera el tiempo de latencia e incluye también el tiempo que el framework tarda en producir resultados para un dato ingresado.

En Spark la duración total del ciclo de procesamiento mostró una media de 1.3 segundos al inicio de la ejecución, contabilizándose como la suma entre latencia y duración del procesamiento del primer microlote. A medida que la ejecución continua, Spark realiza ajustes del tamaño de cada microlote, y la duración total del ciclo de procesamiento baja hasta estabilizarse en una media de 0.5 segundos (latencia 0.3 s. + procesamiento 0.2 s.). El valor mínimo registrado fue de 0.4 segundos.

Flink por su parte al comienzo del procesamiento tiene una duración total del ciclo de procesamiento de 0.15 segundos. Debido a que Flink mantiene valores muy estables de latencia, se observó que no presenta variaciones considerables en la duración total del ciclo de procesamiento durante las pruebas realizadas. Esto permitió observar valores de duración total del ciclo de procesamiento siempre menores o iguales a 0.18 segundos, con una media de 0.13 segundos y un mínimo registrado de 0.1 segundo.

6.6. Resumen de la Evaluación

Después de realizar 10hs de pruebas, tanto Apache Spark como Apache Flink demostraron ser sólidas opciones para el procesamiento de flujos de datos a gran escala.

Ambas herramientas brindan la posibilidad de escalar soluciones cuando las necesidades de procesamiento aumentan, característica vital en el procesamiento de flujos. En este punto las herramientas de monitoreo que proporcionan se vuelven críticas, e incluso una práctica usual es desarrollar herramientas personalizadas de monitoreo, que permitan alertar cuando las necesidades de procesamiento varíen, para que el equipo de desarrollo pueda adaptar el tamaño del cluster a las necesidades concretas de procesamiento.

La escalabilidad que brindan Spark y Flink, combinada con entornos de Cloud Computing permite adaptar dinámicamente el tamaño del cluster a las necesidades concretas de procesamiento. Esto origina un ambiente ideal para que el procesamiento de flujos sea aplicable a todo tipo de problemas y a cualquier escala.

Para tener una mayor comprensión de los datos de rendimiento de cada una de las herramientas evaluadas, se adjunta la ilustración 58 que resume cada aspecto de la evaluación de rendimiento y los resultados de cada uno de los frameworks evaluados.

Aspecto	Apache Spark	Apache Flink
Facilidad de Instalación y Despliegue	Brinda documentación	Brinda documentación
Fuentes admitidas para el intercambio de datos	Se enfoca en diversos formatos de archivo	Se enfoca en conectores a diversas fuentes de datos
Lenguajes de programación soportados	Java, Scala, Python, SQL y R	Java, Scala, Python y SQL
Documentación disponible	Posee mayor Comunidad, documentación y ejemplos	Comunidad en crecimiento, menos documentación y ejemplos
Valor Medio de Uso de CPU	1.5%	0.8%
Valor Medio de Uso de memoria	350Mb	120Mb
Valor Medio de Hilos de ejecución utilizados	277 hilos	62 hilos
Valor Medio de Cantidad de Clases cargadas	20470 clases	14310 clases
Valor Medio de Latencia	0.3 segundos	40 milisegundos
Valor Medio de Duración Total del Ciclo de Procesamiento	0.5 segundos	0.13 segundos

Ilustración 58 –Resumen de la Evaluación Comparativa

7. Conclusiones y Trabajos Futuros

En el mundo del procesamiento de datos, la cantidad de dispositivos que producen información aumenta exponencialmente. Es decir, crece constantemente la cantidad de dispositivos con capacidad informática, entre ellos: televisores, automóviles conectados, teléfonos inteligentes, computadoras para bicicletas, relojes inteligentes, cámaras de vigilancia, termostatos, etc. Nos rodeamos de dispositivos destinados a producir registros de eventos, flujos de mensajes que representan las acciones e incidentes que forman parte de la historia del dispositivo en su contexto. A medida que interconectamos cada vez más esos dispositivos, se requieren mayores capacidades para analizar y procesar los registros de eventos generados. Este fenómeno abre la puerta a una increíble explosión de creatividad e innovación en el dominio del análisis de datos en tiempo real, con la condición de que encontremos una manera de hacer que este análisis sea manejable. En este mundo de dispositivos y registros de eventos agregados, el procesamiento de streaming ofrece una manera eficiente de realizar el análisis de flujos de datos.

El procesamiento de streaming resulta beneficioso en la mayoría de las situaciones en las que se generan datos nuevos y dinámicos de forma continua. Es apto para la mayoría de problemas y casos de uso de Big Data. En un principio, las aplicaciones pueden procesar transmisiones de datos para producir informes básicos y realizar acciones sencillas como respuesta. Con el tiempo, dichas aplicaciones realizan un análisis de datos más sofisticado, como la aplicación de algoritmos de aprendizaje automático y la extracción de información más exhaustiva.

En el mundo del procesamiento de streaming, en la actualidad, hay dos protagonistas principales *Apache Spark* y *Apache Flink*. Los frameworks bajo estudio tienen diversos orígenes, Spark comenzó como un motor de procesamiento distribuido por lotes y luego añadió capacidades de procesamiento de flujos. Spark divide el flujo de datos de entrada en pequeños microlotes, lo que genera un retraso en el procesamiento de cada estímulo recibido. Actualmente Spark ya admite el procesamiento continuo, sin emplear microlotes y con latencia teórica aproximada a 1ms. En este trabajo comparativo no se empleó ese enfoque de procesamiento, porque a la fecha de realización del presente trabajo, el modo continuo no admite el uso de agregaciones. Flink por el contrario nació como un motor de procesamiento de flujos, enfocando directamente su trabajo a responder de manera inmediata cuando se reciben datos de entrada, es decir con muy baja latencia.

En este trabajo se realiza una comparación entre los dos frameworks previamente mencionados. Habitualmente los trabajos comparativos entre herramientas de procesamiento se enfocan en realizar evaluación de rendimiento. En este estudio comparativo no solo se enfoca en evaluar rendimiento, sino también otros aspectos importantes, entre ellos: facilidad de instalación y despliegue, fuentes admitidas para el intercambio de datos, lenguajes de programación soportados y documentación disponible. Considerando todos estos aspectos, se espera proporcionar al lector una base sólida que ayude a la hora de elegir el framework de procesamiento de streaming. En las siguientes secciones se detallan las conclusiones de cada aspecto evaluado.

7.1. Facilidad de Instalación y Despliegue

El primer aspecto evaluado se refiere a facilidad de instalación y despliegue, aquí se tuvo en cuenta la documentación que cada framework proporciona en su sitio web. Ambos frameworks proporcionan documentación en sus sitios web.

La documentación para realizar la instalación y despliegue permite que ambos frameworks se puedan instalar y configurar en diversas plataformas sin inconvenientes.

7.2. Fuentes de Datos Admitidas para el Intercambio de Datos

Los dos frameworks adoptan diferentes enfoques a la hora de realizar ingesta y producción de resultados, Spark concentra sus esfuerzos en soportar diversos formatos de archivos, mientras que Flink se concentra en proporcionar conectores a diversas fuentes de datos.

A pesar de las diferencias de enfoque entre los dos frameworks, esto no genera inconvenientes a la hora de escribir aplicaciones que necesiten consumir y producir resultados.

7.3. Lenguajes de Programación Soportados

En cuanto a lenguajes de programación, ambos frameworks soportan el uso de:

- Java.
- Scala
- SQL.
- Python.

En cuanto al soporte de Python, la Api de Spark está más desarrollada que la de Flink. Spark además soporta el uso de R, esto permite a quienes trabajan en cálculo matemático y ciencias de datos realizar procesamiento de flujos.

7.4. Documentación Disponible

Los dos frameworks brindan documentación y casos de ejemplo básicos. Cuando la complejidad aumenta se debe recurrir a comunidades de usuarios. La comunidad de Spark es actualmente más numerosa y proporciona mayor cantidad de ejemplos, documentación y asistencia en la resolución de problemas que la comunidad de Flink.

Este apartado cobrar mayor importancia a medida que se avanza en implementación de pipelines de procesamiento complejos, al punto que puede convertirse en un cuello de botella en el desarrollo e implementación de aplicaciones de procesamiento de flujos en ambientes de producción.

7.5. Evaluación de Rendimiento

La evaluación de rendimiento se realizó bajo las mismas condiciones, tanto a nivel de configuración de cluster, tareas de procesamiento a realizar y conjunto de datos de entrada. Spark y Flink permiten llevar a cabo optimizaciones para mejorar el rendimiento, en este caso no se realizaron optimizaciones, porque lo que se buscó fue evaluar el rendimiento con la configuración proporcionada al instalar y desplegar el cluster. A continuación, se detallan los aspectos de rendimiento evaluados.

7.5.1. Uso de CPU

El uso de CPU por parte de Spark tuvo una media de 1.5% en las pruebas realizadas. Flink por su parte tuvo una media de 0.8%. Esta diferencia en el uso de CPU también se evidenció en valores máximos y mínimos de uso de CPU. Esto demuestra que en todas las pruebas realizadas Flink consume menos CPU que Spark.

7.5.2. Uso de Memoria

En el apartado uso de memoria, Spark tuvo un valor medio de 350Mb. Flink por su parte, tuvo un valor medio de uso de memoria de 120Mb. Esta diferencia en el uso de memoria también se evidenció en valores máximos y mínimos. En todas las pruebas realizadas Flink consumió menos memoria que Spark.

7.5.3. Cantidad de Hilos de Ejecución Utilizados

El indicador cantidad de hilos de ejecución, monitorea la cantidad de subprocesos activos que cada aplicación y su framework de procesamiento emplean al ser ejecutados.

Spark tuvo un valor medio de 277 hilos, mientras el valor medio de Flink fue de 62 hilos. Esta diferencia en cantidad de hilos también se evidenció en valores máximos y mínimos de cantidad de hilos utilizados. Esto evidencia que Flink emplea una menor cantidad de hilos que Spark en todas las pruebas realizadas.

7.5.4. Cantidad de Clases Cargadas

El apartado cantidad de clases cargadas monitorea el número de clases que cada aplicación y framework emplean al momento de ser ejecutados.

En este aspecto evaluado, Flink carga menor cantidad de clases que Spark. Esto se expresa en el valor medio observado de clases cargadas, como así también en los valores máximos y mínimos.

7.5.5. Latencia

En sistemas de procesamiento de flujos, el término latencia se refiere al tiempo necesario para que el sistema reaccione ante la llegada de un evento o estímulo.

La latencia media observada de Spark fue de 0.3 segundos. Flink por su parte registró una latencia media de 40 milisegundos. La latencia máxima registrada por Spark fue de 1 segundo, mientras que Flink registró 51 milisegundos de latencia máxima. Respecto a latencia mínima, Spark registró 0.2 segundos, mientras que Flink por su parte registró 30 milisegundos de latencia mínima. Estos datos indican que, durante las pruebas realizadas, Flink tuvo menor latencia que Spark.

7.5.6. Duración Total del Ciclo de Procesamiento

La duración total del ciclo de procesamiento considera la duración del tiempo de latencia y también la duración del tiempo que el framework tarda en producir resultados para un dato ingresado.

El valor medio observado en duración total del ciclo de procesamiento fue de 0.5 segundos para Spark, y de 0.13 segundos para Flink. Esta diferencia también se evidenció en los valores máximos y mínimos observados, por lo que, Flink presentó menor duración total del ciclo de procesamiento que Spark.

7.6. Consideraciones Finales

En situaciones donde se necesita procesar flujos de datos en tiempo real, decidir el framework de procesamiento es una tarea compleja. El presente estudio consideró diversos aspectos, no solo evaluación de rendimiento.

Si consideramos facilidad de instalación y despliegue, ambos frameworks proporcionan documentación que permite realizar estas tareas sin inconvenientes.

Respecto a fuentes de datos admitidas los frameworks evaluados enfocan sus esfuerzos desde puntos de vista diferentes, Spark se concentra en formatos de archivo, mientras Flink se enfoca en conectores de datos para brindar acceso a diversas fuentes. Aunque varíe el foco de cada framework, las dos opciones no presentan inconvenientes.

Si se considera lenguajes de programación soportados, ambos frameworks soportan Java, Scala, Python y SQL. Spark saca ventaja en este apartado, ya que tiene más desarrolladas las Api de Python que Flink y además soporta el uso de lenguaje R.

El apartado de documentación disponible favorece a Spark, ya que brinda más documentación y su comunidad de usuarios proporciona mayor cantidad de ejemplos y guías de resolución de problemas que la comunidad de Flink.

La evaluación de rendimiento realizada para el caso de estudio considerado favoreció a Flink, ya que durante las pruebas consumió menos tiempo de procesador, menos cantidad de memoria, utilizó menos cantidad de hilos de ejecución y cargó menos cantidad de clases. También aventaja a Spark por tener menor latencia y menor duración total del ciclo de procesamiento.

Si bien los resultados en las pruebas de rendimiento favorecen a Flink en el caso de estudio considerado, ambos frameworks son opciones eficientes en el procesamiento de streaming a toda escala. Particularmente, en caso de que se necesite realizar procesamiento con restricciones de tiempo críticas, Flink resulta ser más adecuada en base a los registros obtenidos en las pruebas.

7.7. Trabajos Futuros

Una característica que ambos frameworks anuncian es la posibilidad de disponer de algoritmos avanzados de Machine Learning. Esto abre las puertas a trabajos futuros, trabajos que complementen el presente estudio comparativo con análisis de algoritmos de Machine Learning provistos por Spark y por Flink.

También puede ser un trabajo futuro posible, integrar pipelines de procesamiento de flujos y pipelines de procesamiento por lotes, ya que en muchos casos las necesidades de procesamiento de streaming deben convivir con casos de uso de procesamiento por lotes.

Otro desafío interesante relacionado al presente trabajo, es la posibilidad de analizar el desempeño de las herramientas evaluadas en un caso real en un entorno de producción.

8. Bibliografía

[1] Apache Flink References https://flink.apache.org/

[2] Apache Spark References https://spark.apache.org/

[3] Chambers B., Zaharia M. (2018). “Spark. The Definitive Guide”. O’Reilly ISBN: 978-1-491-91221-8.

[4] Hueske F., Kalavri V. (2019). “Stream Processing with Apache Flink”. O’Reilly ISBN: 978-1-491-97429-2.

[5] Maas G., Garillot F. (2019). “Stream Processing with Apache Spark”. O’Reilly ISBN: 978-1-491-94424-0.

[6] Akidau T., Chernyak S., Lax R. (2018). “Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing”. O’Reilly ISBN: 978-1491983874.

[7] Miller H., Haller P., Müller N., Boullier J. “Function Passing: A Model for Typed, Distributed Functional Programming”. ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity, Onward! November 2016.

[8] Python Spark References https://spark.apache.org/docs/latest/api/python/index.html

[9] Python Flink References https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/python/overview/

[10] Saxena S., Gupta S. (2017). “Practical Real-time Data Processing and Analytics: Distributed Computing and Event Processing using Apache Spark, Flink, Storm, and Kafka”. Packt Ltd ISBN: 978-1-78728-120-2.

[11] Zaharia M., Chowdhury M., Franklin M.J., Shenker S., Stoica I. (2010). “Spark: Cluster computing with working sets”. In 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10).

[12] Fajardo H., Hasperué W. (2022). “Procesamiento de Flujo de Datos. Un caso de estudio: Análisis en tiempo real usando datos geolocalizados”. CACIC 2022. La Rioja, Argentina: Universidad Nacional de La Rioja.

[13] “What is streaming data?” References https://aws.amazon.com/streaming-data/

[14] “The Internals of Apache Spark” References https://books.japila.pl/apache-spark-internals/overview/