Ingesta de datos con Kafka y Spark Streaming

Fernanda Maldonado
Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
2021

***Resumen*—La Administración Nacional de Usinas y Trasmisiones Eléctricas (UTE) ha cambiado su matriz energética y con el objetivo de poder hacer un uso más eficiente de la energía eléctrica, se ha instalado y se continúa instalando medidores inteligentes en todo Uruguay. Estos medidores almacenan gran cantidad de datos y resulta imprescindible para la empresa poder contar diariamente con estas medidas. Por consiguiente, el problema que se trata en este documento es mejorar el proceso actual de ingesta diaria de medidas. Para ello, se hizo un estudio de las herramientas que existen en la actualidad y se evaluó cuál se adapta mejor a este escenario. Se diseñó una nueva solución y se construyó un prototipo utilizando Kafka y Spark Streaming. Luego de las pruebas realizadas, los resultados son buenos e impulsan una nueva forma de realizar la ingesta de datos.**

I. Introducción

En los últimos años, Uruguay ha transformado su matriz energética, variando su forma de generación y apostando a energías renovables como la energía eólica. Este cambio ha sido toda una revolución para la empresa y ha traído consigo muchos cambios en su negocio. Un ejemplo, es la adquisición de medidores inteligentes, que ya se han instalado en un 33 % de los clientes. Mediante éstos, es posible hacer un uso más eficiente de la energía y permiten ser gestionados de forma remota (sin necesidad de que un lector asista al lugar). Estos medidores, a diferencia de los tradicionales, son capaces de alamcenar el consumo del cliente cada 15 minutos, lo cual implica que incrementa significativamente el volumen de datos a gestionar. Y en consecuencia, resulta sumamente importante para la empresa, disponer de estos datos de manera rápida y sin pérdidas. Para lograrlo, actualmente se utiliza la herramienta Sqoop, la cual permite extraer las medidas. El problema es que en escenarios donde el sistema recolector de medidas falla o ante fallas en la red de comunicación de los medidores, ambos traen como efecto que se acumulen muchas medidas a extraer y esto provoca que Sqoop funcione lento o incluso, no funcione. Por lo tanto, el objetivo de este trabajo es diseñar una solución que mitigue esas dificultades. Para lograrlo, primero se investiga acerca de las tecnologías disponibles en la actualidad y se estudia su funcionamiento para ver si son aplicables a este escenario. Se eligen Kafka y Spark Streaming, y se realiza un prototipo para validar la integración. No fue posible por temas de infraestructura poder hacer una prueba que simule mejor el escenario real. De todas formas, se logra aprender sobre las tecnologías y qué factores son importantes a la hora de implementar una solución usando estas tecnologías. El resto del documento se organiza de la siguiente forma, en la Sección II se presentan los principales conceptos que serán utilizados en el documento. Luego, en la Sección III se describe el mecanismo para realizar la ingesta de datos y sus inconvenientes. Además, se presentan los factores determinantes para elegir las herramientas a utilizar, cómo funcionan las nuevas herramientas, se presenta la nueva solución y el diseño del prototipo. Posteriormente, en la Sección IV se describen las pruebas realizadas. Seguidamente, en la sección V se detalla brevemente puntos a tener en cuenta para la implantación de la solución en un entorno productivo. Y por último, en la Sección VI se presenta las conclusiones y trabajo futuro.

II. Conceptos básicos

- Medida: consumo integrado cuarto-horario (15 minutal) para una magnitud. Ejemplos de magnitudes: Energía Activa Entrante, Energía Activa Saliente, entre otras.
- Registro: representa la lectura que indica un medidor para una magnitud en un determinado momento. Existe 3 tipos de registros: diarios (a una fecha y hora determinada), parcial (al momento de un reseteo manual) y cierre (al momento de un reseteo automático).
- Evento: suceso detectado por el medidor en un determinado momento. Ejemplos: corte de energía, reestablecimiento de energía, ausencia de tensión, entre otros.
- Medidor inteligente: dispositivo eléctrico con capacidad de registrar hasta 45 días de medidas, registros y eventos, que suceden en un predio donde UTE suministra servicio eléctrico.
- HeadEnd System (HES): sistema encargado de gestionar la recolección de los datos de los medidores, verificar su conectividad en la red y ofrecer un mecanismo de acceso seguro a los medidores para realizar actualizaciones de configuración y de firmware.
- Meter Data Management (MDM): sistema encargado de centralizar y gestionar todo el histórico de todos los datos enviados por los medidores inteligentes. Se compone de diferentes módulos encargados de realizar cálculos, estimaciones y validaciones de los datos.

III. Problema

III-A. Descripción

La ingesta de datos de un sistema suele ser un problema no menor sobre todo cuando se manejan grandes volúmenes de datos. MDM es un sistema que a diario recibe en el orden

de cientos de millones de mediciones, yendo en constante crecimiento debido a que por el momento solamente 33 % de los clientes poseen medidores inteligentes. Cada medidor inteligente envía sus datos a una base de datos relacional del HES y es necesario tener un mecanismo para hacer que dichas medidas lleguen al sistema MDM.

III-B. Solución actual y sus inconvenientes

Para resolver el problema, actualmente se utiliza la herramienta Sqoop. Con Sqoop es posible importar datos desde una base relacional al sistema de archivos distribuido Hadoop Distributed File System (HDFS). Para realizar esta importación, se define una consulta SQL sobre los datos que interesan y Sqoop realiza un proceso MapReduce, produciendo como salida un conjunto de archivos que contienen el resultado de la consulta SQL. Luego, con otro programa MapReduce es que se leen dichos archivos, se validan los datos y se cargan las medidas en una base de datos no relacional HBase de MDM.

En la Figura 1 se ilustran los pasos para realizar la ingesta de datos.

Si bien Sqoop logra resolver el problema, es una herramienta que presenta inconvenientes ante escenarios adversos como puede ser la caída del sistema HES o algún problema sobre la red de comunicación de los medidores. Lo que provocan ambos escenarios es que se acumulen muchos datos para extraer y eso hace que Sqoop empiece a demorar varias horas o peor aún, que termine en error la extracción. Ante estos casos, siempre se debe particionar el tiempo a extraer en al menos 6 horas. Sin embargo, dado que el volumen a extraer crece día a día, es probable que en un futuro el intervalo de 6 horas no sea suficiente (no sea posible de extraer) y se tenga que seguir reduciéndolo.

III-C. Herramientas para una nueva solución

Dado que Sqoop no es una herramienta pensada para hacer streaming, este trabajo propone una solución utilizando herramientas que sí hayan sido pensadas para ello. Para tomar la decisión de qué herramientas utilizar, los factores que fueron determinantes son:

Figura 1: Ingesta de datos con Sqoop

- **Latencia**: dependiendo qué tan baja latencia se necesite, existen diferentes opciones de herramientas. En este contexto, la latencia no es crítica. Tomando en cuenta que este no es un sistema crítico como puede llegar a ser un sistema de monitoreo de pacientes en cuidados intensivos o de un avión, se entiende que no es necesaria una baja latencia. En MDM no se tomarán decisiones en base a las medidas en tiempo real, sino que serán almacenadas para un posterior análisis.
- **Procesamiento**: según cómo se quieran procesar los datos, también existen distintas alternativas. Hay dos formas de ir procesando, una consiste en ir procesando dato a dato (a medida que se van recibiendo) y otra consiste en ir procesando en batches (colecciones de datos). En este escenario, el procesamiento será en batches, dado que no aportaría valor procesar cada dato de manera individual y tampoco resulta relevante procesarlo lo más cercano posible al momento en que ocurre.
- **Integración**: es necesario buscar herramientas que sean fáciles de integrar con los sistemas HES y MDM, debido a que éstos ya se encuentran productivos.
- **Volumen**: es importante que las herramientas estén pensadas para manejar grandes volúmenes de datos. Actualmente se reciben 10 GB diarios en medidas. Pero se debe tomar en cuenta que a futuro se prevee que la totalidad de clientes del país posea un medidor inteligente y además se debe considerar que la solución se podría extender a otras entidades como registros y eventos. Tomando en cuenta todo esto, se espera recibir a futuro un total de 60 GB diarios.
- **Escalabilidad horizontal**: considerando la posibilidad de extender la solución, la escalibilidad de la herramienta es un aspecto muy valioso ya que permite aumentar la capacidad de procesamiento sin mayores dificultades, simplemente agregando nuevos servidores.
- **Tolerancia a fallas**: tomando en cuenta que se manejarán grandes volúmenes de datos y se trabajará en entornos distribuidos, resulta fundamental que las herramientas elegidas tengan mecanismos para ser tolerantes a fallas, ya que eso permite a un sistema seguir funcionando correctamente a pesar de que falle uno o varios de sus componentes.

En base a los factores se tomó la decisión de utilizar Kafka y Spark Streaming.

III-D. Kafka

Kafka es un sistema de mensajería distribuido basado en el modelo productor-consumidor. Centraliza la comunicación entre productores y consumidores de los datos. A su vez, fue diseñado para ser rápido y capaz que manejar grandes volúmenes de datos debido a su capacidad de ejecutar de forma distribuida. La unidad de dato dentro de Kafka se llama mensaje.

En la práctica, se suele montar un cluster de Kafka y para entender su funcionamiento, se definen los conceptos:

- Kafka brokers: son los mediadores entre los que publican los mensajes y los que consumen los mensajes. Un conjunto de brokers son los que conforman el cluster de Kafka.
- Productor Kafka: aplicación encargada de publicar mensajes. Se puede tener 1 o más.
- Consumidor Kafka: aplicación encargada de consumir mensajes. Se puede tener 1 o más.
- Kafka topics: los mensajes se organizan en topics. Tanto al momento de escribir/leer se hace hacia/desde un topic específico.
- Partición: los topics son divididos en una o más particiones que serán distribuidas entre los brokers. Esto permite que varios consumidores puedan leer de un mismo topic en paralelo. Cada mensaje dentro de una partición posee un identificador llamado offset, que define un orden entre todos los mensajes que va recibiendo la partición. Los consumidores pueden leer a partir del offset que deseen. Cada mensaje en un cluster de Kafka puede ser identificado de forma única mediante la tupla: topic, partición y offset dentro de la partición. Para saber cómo se debe elegir el número de particiones ver Apéndice VII-B. Para saber cómo se asigna el mensaje a una partición ver Apéndice VII-C.
- ZooKeeper: encargado de controlar el estado del cluster. Actúa como un repositorio de configuraciones, manteniendo toda la metadata del cluster. Envía notificaciones a los brokers ante creación/eliminación de un topic, caída/levantamiento de un broker y es el responsable de elegir un broker lider para las distintas particiones.

Un cluster de Kafka se dice que es altamente escalable ya que para aumentar su capacidad de procesamiento basta con agregar más servidores como brokers. Para saber cómo determinar el número de brokers, ver Apéndice VII-D. Por lo tanto, su flexibilidad para escalar hacen que teniendo la infraestructura adecuada, Kafka pueda manejar cualquier cantidad de datos. Existen en la actualidad muchas empresas que lo utilizan, ej.: LinkedIn que mantiene 100 clusters con más de 4000 brokers que sirven a 100000 particiones y 7 millones de réplicas, capaz de manejar alrededor de 7 billones de mensajes por día. [Lee(2019)]

Como todo sistema distribuído compuesto de varios nodos, es inevitable que los nodos fallen. Pero Kafka, es tolerante a fallas, esto quiere decir que si alguno de sus servidores falla, los otros servidores se harán cargo de sus trabajo automáticamente para garantizar que no haya interrupciones en el servicio y que además no haya pérdida de datos. Para poder asegurar la disponibilidad y durabilidad de los mensajes cuando un servidor falla, lo hace mediante réplicas. Como se dijo anteriormente, los mensajes se organizan en topics, cada topic posee una o varias particiones, y cada partición puede tener una o más réplicas. Las réplicas se almacenan en los brokers y cada broker puede almacenar hasta miles de réplicas pertenecientes a distintos topics y particiones. Existen dos tipos de réplicas:

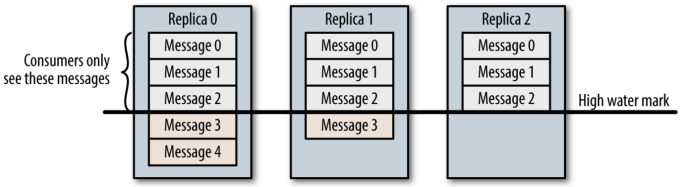

Figura 2: Los consumidores de Kafka solamente pueden leer mensajes que se encuentren replicados en todas las réplicas.
* Ilustración de [Neha Narkhede(2017)]

- Líder: cada partición tiene una única réplica designada como líder. Todos los pedidos de escritura y lectura que llegan a una partición son atendidos por el líder para garantizar la consistencia.
- Seguidor: el resto de las réplicas de una partición que no son líder, son seguidoras. Las particiones seguidoras solamente replican los mensajes del líder y su tarea es estar al día con todo lo que tiene el líder. En el caso de que una réplica líder falle, una de las particiones seguidoras se convertirá en el nuevo líder.

El líder también es responsable de saber cuáles seguidoras están al día con respecto a él (conoce al detalle qué mensajes ya fueron replicados en cada réplica). Las seguidoras siempre intentan estar al día con el líder a medida que van llegando nuevos mensajes, pero pueden quedar desincronizadas por problemas en la red o por una falla en su broker.

Un detalle interesante es que no todos los mensajes que están presentes en el líder están disponibles para los consumidores. Las aplicaciones consumidoras pueden leer solamente los mensajes que se encuentren sincronizados en todas las réplicas (ver Figura 2). El motivo detrás de este comportamiento es que si el líder por alguna razón falla y otra réplica toma su lugar, estos mensajes no existirán en Kafka. Por lo que si se permite a los clientes leer mensajes que solamente existen en el líder, podría dar lugar a inconsistencias.

Otro aspecto importante es que Kafka permite configurar por cuánto tiempo interesa que los mensajes permanezcan en Kafka antes de ser eliminados. Esto permite que los consumidores no tengan la necesidad de estar siempre trabajando. Los mensajes se graban en disco por cierto tiempo y se puede tener la tranquilidad de que si los consumidores no están trabajando o fallan, cuando retomen se pondrán al día y no se perderá ningún mensaje. O sino, también ofrece la posibilidad de guardar los datos hasta alcanzar cierto tamaño. Ambas configuraciones se puede realizar según el topic.

Otro detalle a resaltar es la forma en la que se paraleliza el conusmo de datos de un topic específico. Sucede frecuentemente que los consumidores de Kafka realizan operaciones de alta latencia, como puede ser: escribir en un sistema de archivos. Ante estos escenarios, puede suceder que un solo consumidor no logre quedar al día con el/los productor/es de los mensajes, ya que escriben a una velocidad más rápida de la que un consumidor puede leer y procesar, ahí es donde entra el concepto de grupos de consumidores. Este concepto

permite que múltiples consumidores puedan leer de un mismo topic, dividiendo los datos entre ellos, logrando que miembros del mismo grupo no reciban los mismos datos (ver Apéndice VII-A para más información).

Por último, se destaca que Kafka incluye la herramienta Kafka Connect[1] que permite transferir datos de manera escalable y confiable entre Kafka y otros sistemas. Provee una forma simple de definir conectores. Un conector posibilita por ejemplo escuchar cambios en una base de datos y replicarlos dentro de Kafka.

III-E. Spark Streaming

Spark Streaming es una extensión de la API core de Spark, la cual permite crear aplicaciones dedicadas a procesar streaming de datos y que además sean tolerantes a fallas. Soporta varios lenguajes de programación: Java, Scala y Python. Asimismo, es capaz de leer/enviar los datos desde/hacia diferentes fuentes (ej.: consumir datos desde Kafka y almacenarlos en HDFS). Ofrece dos maneras de procesar los datos: modo batch y modo streaming (cada dato individualmente).

Adicionalmente, Spark está pensando para ejecutar en entornos distribuidos. Se definen algunos de los conceptos básicos que serán de utilidad para entender su funcionamiento:

- Programa driver: proceso que ejecuta la función main() de la aplicación y encargada de crear el contexto de Spark.
- Ejecutor: proceso lanzado por una aplicación, que ejecuta tareas y mantiene los datos en memoria o en disco. Cada aplicación tiene sus propios ejecutores.

Para asegurar la tolerancia a fallas y que no haya pérdida de datos, Spark Streaming permite guardar el estado de la aplicación en un directorio llamado checkpoint. Ante una falla en el programa driver, es posible reiniciar y retomar desde el último checkpoint. Cuando un programa driver falla, todos sus ejecutores son eliminados junto con los datos que tenían en memoria. Para ejemplificar la idea, se toma en cuenta el escenario presentado en este documento, un ejecutor tendrá datos que son recibidos desde Kafka y los mismos estarán en memoria hasta que su procesamiento haya sido completado. Para poder recuperar estos datos en memoria ante una falla, es necesario utilizar los Write Ahead Logs (WALs). Estos logs son almacenados en disco y contendrán una copia de lo que se recibe desde Kafka. Por lo tanto, cada ejecutor como primer paso debería escribir en los WALs y ante una falla, sabrá que podrá recuperar los datos desde allí. Es buena práctica que luego de haber escrito sobre los WALs, se avise a Kafka que se han recibido correctamente los datos [Das(2015)]. Esto ayudará a que ante una falla, todo dato que haya sido recibido y se encuentre en memoria pero no escrito en WALs, se pueda recuperar dado que Kafka tiene conocimiento de hasta dónde se logró almacenar (ver Apéndice VII-E).

Otro punto interesante es que Spark Streaming se puede integrar sin problemas con cualquier otro componente de

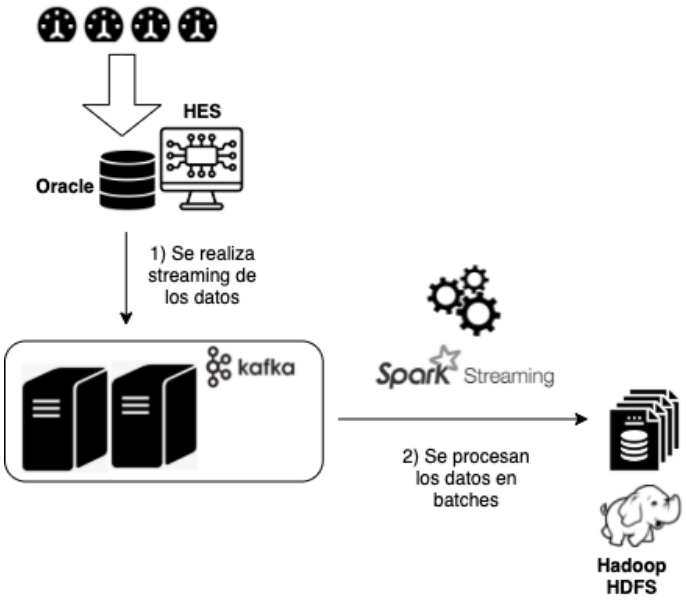

Figura 3: Ingesta de datos con Kafka y Spark Streaming

Spark como MLib (para machine learning) y SparkSQL (permite tratar al streaming de los datos como una tabla y hacer consultas SQL) lo cual abre la posibilidad a otros escenarios que pueden ser de utilidad para la empresa.

Y por último, un detalle que parece trivial pero se quiere dejar en claro es que con Spark Streaming se puede hacer transformaciones sobre los datos previo a ser almacenados.

III-F. Diagrama de la nueva solución

Luego de estudiar las características del problema a resolver y de la investigación sobre las capacidades y funcionamiento de cada herramienta es que se diseñó una nueva solución utilizando Kafka y Spark Streaming. Se presenta en la Figura 3 un diagrama de la nueva solución.

III-G. Diseño del prototipo

Para poder validar el diseño y la integración de las herramientas, se realizó un prototipo. Para el armado se utilizó la plataforma Docker, que permite crear, probar e implementar aplicaciones de forma rápida. Con Docker se crean contenedores que contienen todas las configuraciones necesarias para que el software en cuestión pueda ejecutar correcamente. La gran ventaja de Docker es que asegura que el contenedor puede ejecutar en cualquier otro equipo ya que encapsula en el contenedor todas las configuraciones y librerías para que nuestra aplicación pueda funcionar. Adicionalmente, Docker ofrece un repositorio llamado Docker Hub[2] donde se puede encontrar una gran variedad de contenedores ya listos para usar, lo cual agiliza mucho el desarrollo y evita tener que realizar tareas repetitivas de instalación y configuración.

Se presenta en Figura 4 la arquitectura del prototipo.

[1]https://kafka.apache.org/documentation/#connect

[2]https://hub.docker.com/

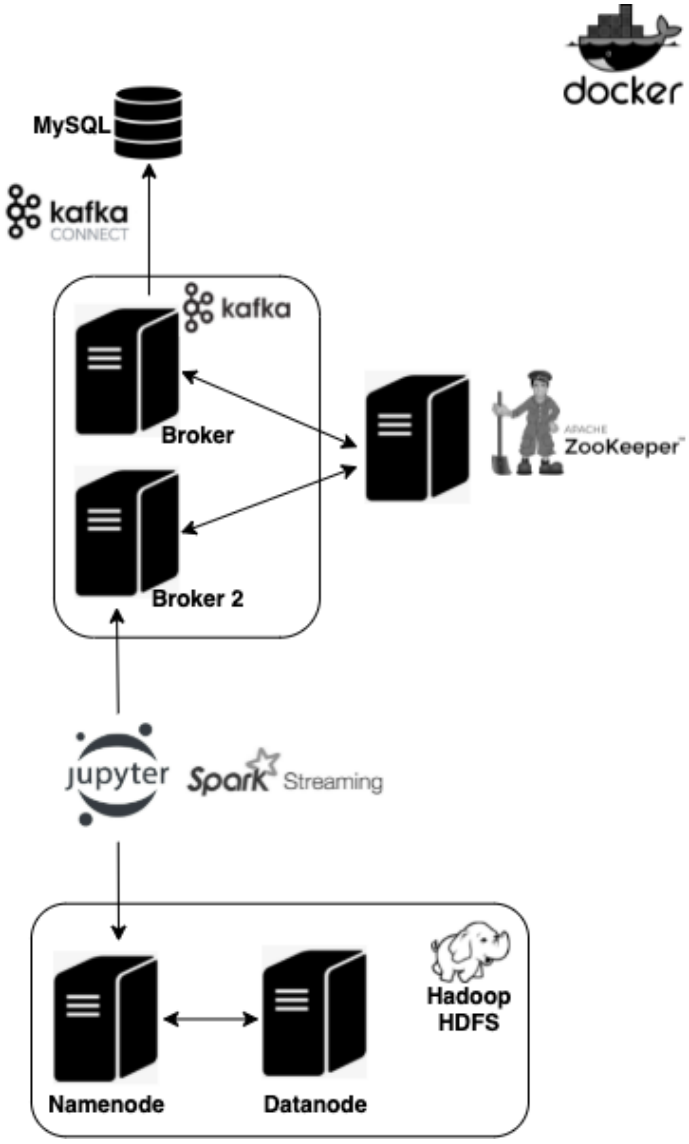

Figura 4: Prototipo usando Docker

Para la solución se utilizaron 7 contenedores:

- Contenedor MySQL: el cual se utiliza en representación de la base Oracle del HES. [MyS(2021)] La misma contiene una única tabla MEASURE con las columnas:
  - METER - identificador del medidor.
  - MEASURE_DATE - fecha de la medida.
  - PERIOD - periodo de la medida (al ser 15 minutal puede ir de 1 a 96).
  - MAGNITUDE - representa una magnitud.
  - VALUE - valor de la medida.
  - TS - momento en el que se creó el registro en la base de datos.
- Contenedor Broker: encargado de atender al productor y/o consumidor de los datos. Además de su función de broker, tendrá un rol extra que será funcionar como conector entre la base MySQL y Kafka. [Kaf(2021)]
- Contenedor Broker 2: encargado de atender al productor y/o consumidor de los datos. Broker y Broker 2 conforman el cluster de Kafka.
- Contenedor Zookeeper: su tarea es controlar el estado de los brokers y mantenerlos informados de los cambios que ocurren. [Zoo(2019)]
- Contenedor Jupyter: en el cual se tienen las librerías necesarias para ejecutar Spark Streaming utilizando como lenguaje de programación Python. Esta aplicación es la que consume los mensajes de Kafka. [Jup(2021)] [Spa()]
- Contenedor Namenode: para montar un sistema HDFS es necesario disponer de al menos un namenode (maestro) y un datanode (esclavo). El namenode recibe los pedidos para almacenar los datos. [Nam(2020)]
- Contenedor Datanode: es el nodo encargado de persistir los datos. Recibe las solicitudes de parte del namenode. [Dat(2020)]

Para poder centralizar la ejecución de todos los contenedores en simultáneo, se utilizó la herramienta Docker Compose[3]. Mediante un archivo de configuración, es posible describir todas las indicaciones necesarias para levantar e interconectar todos los nodos. Todo el código se encuentra disponible aquí [4].

IV. Experimentación

Para probar la nueva solución, se levantó el prototipo en una sola máquina con estas características:

Sistema operativo: macOS Big Sur versión 11.4
Procesador: 2.7 GHz Intel Core i5 de dos núcleos
Memoria: 8 GB 1867 MHz DDR3

En un principio fue costoso hacer que todos los contenedores se conecten entre sí, teniendo que ajustar varios archivos de configuración. Los blogs [Kaf(2019)] y [Con(2020)] fueron de mucha utilidad para entender cómo configurar bien los brokers y sus puertos. Para conectar la base MySQL con Kafka, resultó de utilidad el blog [Goswami()] que explica detalladamente los pasos a seguir y fue necesario buscar un conector especial para la versión de MySQL utilizada. Este conector es utilizado por Kafka Connect para poder comunicarse.

Se configuró Kafka Connect para que busque nuevos datos en la base cada 10 segundos. El modo en el que opera es timestamp, a través de la columna TS. Esto significa que usa la columna TS para detectar nuevas filas o cambios en las filas.

Luego, se creó un único topic *measure* con una sola partición y sin réplica.

Se definió que en el consumidor la duración del batch sea baja (30 segundos), esto quiere decir que cada 30 segundos se irán armando batches con todos los datos recibidos.

Las pruebas consistieron básicamente en:

[3]https://docs.docker.com/compose/install/
[4]https://gitlab.fing.edu.uy/dalila.fernanda.maldonado/bdnr/-/tree/master

1. Conectarse al nodo que contiene la base de datos y ejecutar comandos sql de tipo insert sobre la tabla MEASURE.
2. Ejecutar comando para visualizar mensajes dentro del topic.
3. Verificar que el consumidor reciba correctamente los datos.
4. Verificar que los datos queden correctamente almacenados en el sistema de archivos.

Durante las pruebas, se sufrieron varios inconvenientes debido a la poca capacidad de procesamiento para simular tantos servidores. Inicialmente se creó el topic measure con una réplica pero esto trajo como consecuencia que algunos contenedores quedaran sin responder, provocando que el consumidor no pueda leer los mensajes de Kafka. Por lo tanto, al quedar sin réplica, es el contenedor broker quien contiene el topic con su única partición. Si bien el consumidor puede comunicarse con cualquier broker del cluster que esté disponible, durante las pruebas se forzó a que se comunicara siempre con el broker 2 para corroborar que la comunicación entre brokers esté funcionando correctamente, ya que el broker 2 no es quien almacena los mensajes.

Otro inconveniente que se tuvo es que si se activa para almacenar los datos que procesa el consumidor en HDFS y además el consumidor activa su directorio checkpoint para recuperarse ante fallas, todo el entorno queda inestable por la poca capacidad de procesamiento. Por lo tanto, tuvo que probarse la escritura en HDFS y la activación de checkpoints por separado.

Se controló que si la aplicación consumidora no está levantada y se insertan nuevos mensajes en Kafka, cuando se levanta la aplicación logra ponerse al día a partir del punto correcto.

También se verificó que si se insertan registros en la base MySQL y no está levantado Kafka Connect, cuando se levanta el conector se pone al día a partir del punto correcto.

No es posible hacer una comparación de tiempos con la solución actual debido a que la realidad simulada en este prototipo difere mucho de la real y que además durante la prueba se tuvo que limitar muchas de las configuraciones:

- Cantidad de brokers.
- Cantidad de particiones de un topic.
- Factor de replicación de una partición.
- Tiempo de retención de los mensajes en Kafka.
- Cantidad de ejecutores de Spark.
- Activación de checkpoint y WALs en la aplicación consumidora.
- Carga baja de inserts sobre la la base MySQL (no se pudo simular una carga que se asemeje más a la realidad del HES).

Sin duda que cada uno de los puntos mencionados anteriormente tienen un impacto en la performance de la solución y resulta fundamental experimentar un poco más para encontrar el balance adecuado para este escenario antes de hacer una comparación con la solución actual.

V. Puntos a tener en cuenta para la implantación de la solución en un escenario real

Dado que es un tema muy crítico para la empresa, resulta indispensable armar una buena estrategia para implantar un cambio en la ingesta de datos. Por lo tanto, será primordial probar la solución en un ambiente con mayores prestaciones, para poder experimentar en mayor profundidad algunos aspectos (descriptos en la Sección IV). Además, la nueva solución con respecto a la actual, sería mas compleja de monitorear ya que con la actual simplemente se ejecuta una query y se observa si el archivo final tiene la cantidad esperada. Y con la nueva solución, pueden llegar a existir distintos puntos de falla como puede ser el conector Kafka Connect, problemas en el cluster de Kafka o en la aplicación consumidora. Sin embargo, un punto importante a destacar es que esta solución podría convivir con la actual, de manera que esto facilitaría la entrada en producción y permitiría chequear varios elementos, tales como:

- Verificar que el cluster de Kafka haya sido correctamente dimensionado (cantidad de brokers).
- Verificar que las particiones y réplicas estén funcionando correctamente.
- Confirmar si los tiempos de recolección de los datos son aceptables.
- Asegurar que no hayan datos faltantes (comparando resultado de solución actual versus nueva solución).
- Analizar si la base de datos del HES está siendo muy penalizada por Kafka Connect.
- Analizar si el uso de los WALs ocupa mucho espacio en disco o penaliza mucho los tiempos del consumidor. Y así definir si es conveniente tener que replicar todos los datos de Kafka en estos archivos o si se puede asumir el riesgo y en el peor escenario siempre recuperar desde Kafka.

VI. Conclusiones y trabajo futuro

Como conclusión se puede decir que se cumplió en gran parte con el objetivo planteado de diseñar una solución basado en otras herramientas para hacer la ingesta de datos a MDM. Si bien se cree que esta nueva solución mitigará los problemas con Sqoop, no fue posible asegurarlo debido a que la solución no pudo ser probada bajo condiciones similares. De todas formas, se cree que no presentará los mismos inconvenientes dado que se utilizaron herramientas que estan pensadas para procesar datos en streaming. Ambas son muy usadas en la actualidad, y en particular se tiene referencia de Kafka con aplicaciones en miles de empresas [Com()].

Se logró identificar qué factores son determinantes a la hora de buscar herramientas para la ingesta de datos: latencia, procesamiento, integración, volumen, escalabilidad horizontal y tolerancia a fallas.

Adicionalmente, se adquirió buen conocimiento acerca de cómo funciona Kafka y Spark Streaming. Dado que son herramientas muy usadas, existe una buena comunidad con grandes aportes y en particular el libro acerca de Kafka

[Neha Narkhede(2017)] logra explicar con bastante claridad todos los aspectos técnicos.

Si bien con las pruebas solamente se pudo probar la integración entre todas las herramientas, aún así fue de mucha utilidad para entender todo el funcionamiento y aprender aspectos técnicos que se deben tener en cuenta a la hora de configurar el entorno. También se aprendió acerca de qué puntos son importantes al momento de elegir la cantidad de brokers, número de particiones y para qué sirven los grupos de consumidores.

Y por otro lado, el saber que Kafka es muy utilizado y posee aplicaciones en miles de compañías da cierta garantía de que se está yendo por el camino correcto y la solución debería poder aplicarse a este escenario sin mayores inconvenientes.

Con respecto a trabajo futuro, se plantean varias posibilidades. En primer lugar, extender esta solución a las otras entidades del HES como son registros y eventos. En particular para eventos, se podría estudiar la posibilidad de usar SparkSQL ya que esta herramienta permite hacer consultas SQL sobre streaming de datos que puedan ser de utilidad en este contexto (ejemplo: generar alertas cuando se reportan más de x cantidad de eventos de corte para una misma zona). Y como segundo lugar, se puede buscar herramientas que ayuden a monitorear el cluster de Kafka, ya que como se dijo en la Sección V, esta nueva solución es mucho más compleja de controlar con respecto a la solución actual. Sería de gran utilidad poder visualizar este tipo de información:

- Total de particiones líderes.
- Total de particiones seguidoras.
- Distribuición de los líderes en los brokers.
- Métricas acerca de cuánto tiempo están demorando los mensajes entre el líder y sus seguidoras.
- Tamaño de las particiones.
- Cantidad de mensajes recibidos por segundo por topic.
- Cantidad de mensajes recibidos por segundo en cada broker por topic.
- Uso de CPU y Memoria en brokers.
- Cantidad de mensajes generados versus Cantidad mensajes consumidos por las aplicaciones.

Sería bueno disponer de una herramienta que permita visualizar estas métricas para poder controlar que el cluster está funcionando adecuadamente y poder anticiparse a problemas que puedan surgir.

REFERENCIAS

[Com()] Companies using Kafka. https://kafka.apache.org/powered-by. Online; accessed 21 July 2021.

[Spa()] Spark Streaming + Kafka Integration Guide (Kafka broker version 0.8.2.1 or higher). https://spark.apache.org/docs/2.4.5/streaming-kafka-0-8-integration.html. Online; accessed 21 July 2021.

[Kaf(2019)] Kafka Listeners – Explained. https://www.confluent.io/blog/kafka-listeners-explained/?utm_source=github&utm_medium=rmoff&utm_campaign=ty.community.con.rmoff-listeners&utm_term=rmoff-devx, 2019. Online; accessed 21 July 2021.

[Zoo(2019)] Zookeeper Image. https://hub.docker.com/r/wurstmeister/zookeeper, 2019. Online; accessed 21 July 2021.

[Con(2020)] My Python/Java/Spring/Go/Whatever Client Won't Connect to My Apache Kafka Cluster in Docker/AWS/My Brother's Laptop. Please Help! https://www.confluent.io/blog/kafka-client-cannot-connect-to-broker-on-aws-on-docker-etc/?utm_source=github&utm_medium=rmoff&utm_campaign=ty.community.con.rmoff-listeners&utm_term=rmoff-devx, 2020. Online; accessed 21 July 2021.

[Dat(2020)] Hadoop datanode of a hadoop cluster Image. https://hub.docker.com/r/bde2020/hadoop-datanode, 2020. Online; accessed 21 July 2021.

[Nam(2020)] Hadoop namenode of a hadoop cluster Image. https://hub.docker.com/r/bde2020/hadoop-namenode, 2020. Online; accessed 21 July 2021.

[Jup(2021)] Jupyter Notebook with Python, Scala, R, Spark, Mesos Stack Image. https://hub.docker.com/r/jupyter/all-spark-notebook, 2021. Online; accessed 21 July 2021.

[Kaf(2021)] Multi-Broker Apache Kafka Image. https://hub.docker.com/r/wurstmeister/kafka, 2021. Online; accessed 21 July 2021.

[MyS(2021)] MySQL Image. https://hub.docker.com/_/mysql, 2021. Online; accessed 21 July 2021.

[Das(2015)] Tathagata Das. Improved Fault-tolerance and Zero Data Loss in Apache Spark Streaming. https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html, 2015. Online; accessed 21 July 2021.

[Goswami()] Gautam Goswami. Data Ingestion From RDBMS By Leveraging Confluent's JDBC Kafka Connector. https://dataview.in/data-ingestion-jdbc-kafka-connector/. Online; accessed 21 July 2021.

[Lee(2019)] Jon Lee. How LinkedIn customizes Apache Kafka for 7 trillion messages per day. https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages, 2019. Online; accessed 21 July 2021.

[Neha Narkhede(2017)] Gwen Shapira Todd Palino Neha Narkhede. *Kafka The Definitive Guide*. O'Reilly, Estados Unidos, 1 edition, 2017.

VII. APÉNDICE

VII-A. Grupos de consumidores Kafka sobre mismo topic

Dado que Kafka permite que varios productores puedan escribir sobre un mismo topic, resulta necesario también que varios consumidores puedan leer sobre un mismo topic. Si se estuviera limitado a un solo consumidor por topic podría convertirse en un problema si la velocidad a la que escriben los productores supera a la velocidad en la que un consumidor puede leer y procesar. En consecuencia, esto provocaría que la aplicación consumidora se retrase cada vez más y sea incapaz de estar al día con los mensajes entrantes. Cuando se quiere paralelizar la lectura de un mismo topic, se define un Grupo de consumidores. Cuando varios consumidores pertenecen al mismo grupo, cada consumidor recibe un subconjunto diferente de mensajes. Esto permite agilizar el proceso de consumo de los datos y también en caso de que falle un consumidor del grupo, el resto podrá tomar su trabajo. Es importante tener en cuenta la relación entre particiones y consumidores al momento de definir un grupo. En la Figura 5 se presenta el caso más sencillo. En este escenario el consumidor 1 obtendrá todos los mensajes del topic T1 leyendo todas las particiones. En la Figura 6 se agrega un consumidor más al grupo G1, por lo que se dividirán las particiones entre ellos. Posteriormente, en la Figura 7 se agregan dos consumidores más al grupo G1 y en consecuencia cada uno de ellos tendrá una sola partición asignada. Y por último, en la Figura 8 se agrega un quinto consumidor y en este caso éste quedará inactivo y no recibirá mensajes ya que no tiene partición de dónde consumir. Por lo tanto, nunca tiene sentido tener más consumidores que particiones.

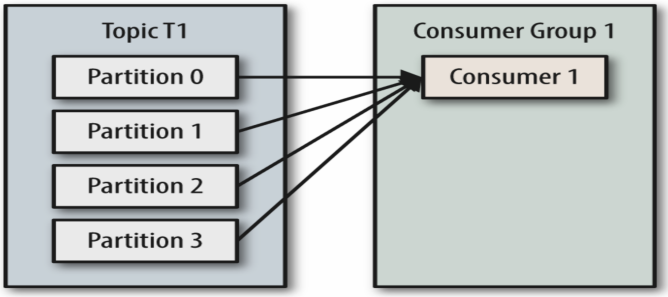

Figura 5: 4 Particiones y 1 Consumidor
* Ilustración de [Neha Narkhede(2017)]

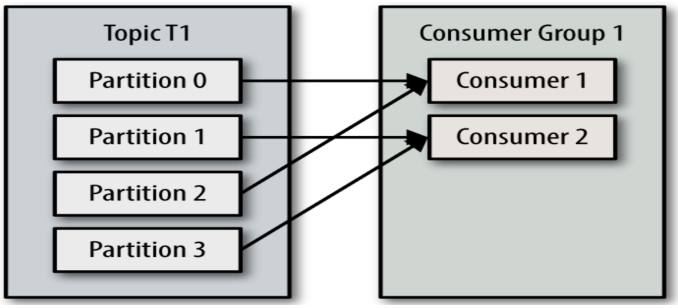

Figura 6: 4 Particiones y 2 Consumidores
* Ilustración de [Neha Narkhede(2017)]

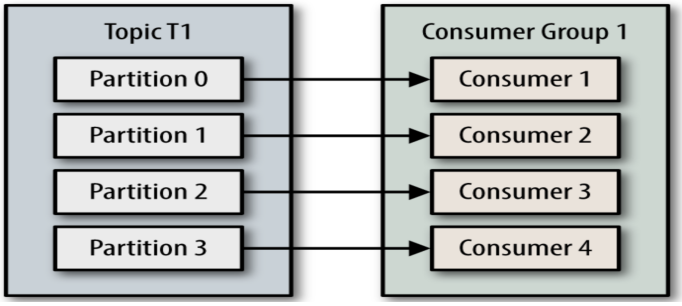

Figura 7: 4 Particiones y 4 Consumidores
* Ilustración de [Neha Narkhede(2017)]

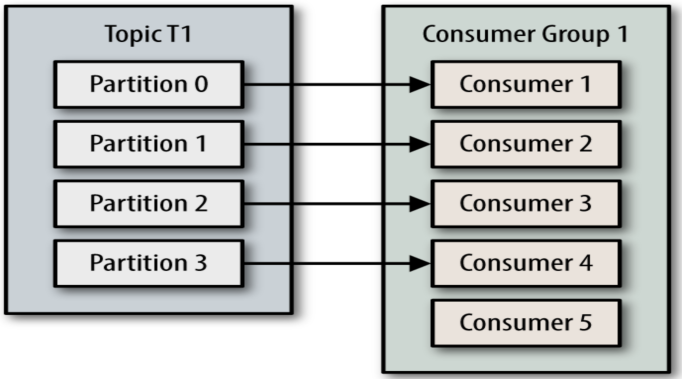

Figura 8: 4 Particiones y 5 Consumidores
* Ilustración de [Neha Narkhede(2017)]

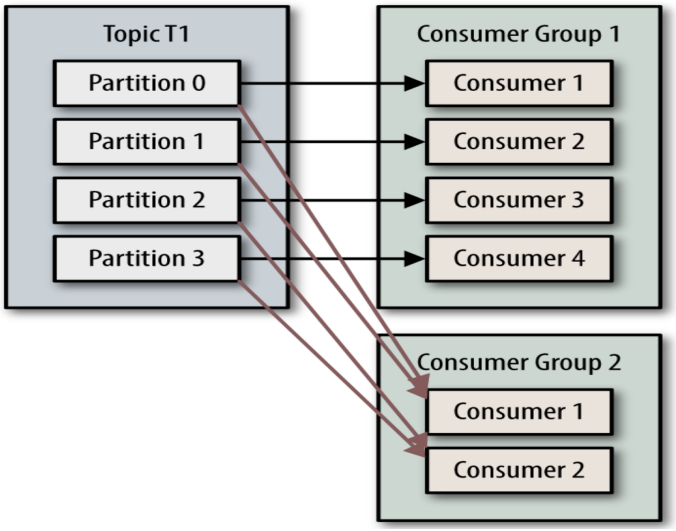

Figura 9: 4 Particiones y 2 Grupos de Consumidores
* Ilustración de [Neha Narkhede(2017)]

Para asegurar que una aplicación lea todos los mensajes de un topic, es necesario asegurar que la apliación tenga su propio Grupo de Consumidores. Si por ejemplo, se agrega un nuevo Grupo de Consumidores G2 (ver Figura 9), los consumidores de este grupo recibirán todos los mensajes del topic T1 independientemente de lo que el Grupo de Consumidores G1 esté haciendo. Ambos grupos recibirán todos los mensajes del topic T1. Esto resulta muy útil debido a que permite que aplicaciones con distintos fines puedan consumir los mismo datos sin interferir entre ellas.

VII-B. ¿Cómo elegir el número de Particiones de un Topic?

El número de particiones de un topic es configurable y existen varios factores a considerar cuando se intenta configurar:

- ¿Qué rendimiento se espera lograr para el topic? Es decir, ¿se espera escribir 100 KB por segundo o 1 GB por segundo?
- ¿Cuál es el rendimiento máximo que se espera lograr cuando se consume desde una sola partición? Si por ejemplo se sabe que el consumidor escribe los datos en una base de datos y esta base de datos nunca maneja más de 50 MB por segundo en cada hilo que escribe, entonces se sabe que estará limitado a 50 MB de rednimiento cuando se consume desde una partición.
- Se puede repetir el mismo ejercicio para estimar el máximo rendimiento por productor para una única partición, pero sabiendo que los productores son típicamente mucho más rápidos que los consumidores, es usual saltearse este paso.
- Si los mensajes enviados a las particiones van con claves, agregar particiones después puede resultar muy desafiante. Por lo tanto, es importante calcular el rendimiento basado en lo que se espera a futuro y no en el uso actual.

- Considerar el número de particiones que se colocarán en cada broker, el espacio disponible en disco y el ancho de banda por broker.
- Evitar sobreestimar, ya que cada partición usa memoria y otros recursos del broker y eso incrementará el tiempo para la elección de líderes.

Si por ejemplo se quiere ser capaz de escribir y leer 1 GB por segundo desde un topic y se sabe que cada consumidor puede solamente procesar 50 MB por segundo, se puede deducir que se necesitan al menos 20 particiones. De esta forma, se tendrán 20 consumidores leyendo el mismo topic a 1 GB por segundo.

Si a priori no se dispone de una estimación del rendimiento esperado, se sugiere limitar el tamaño de la partición en el disco a menos de 6 GB por día de retención, ya que a menudo da resultados satisfactorios.

VII-C. Asignación de mensajes a particiones

Los mensajes de Kafka son una pareja clave-valor, donde la clave puede ser nula. Cuando se especifica una clave, se usa para decidir a cuál de las particiones del topic, le corresponde guardar el mensaje. Todos los mensajes con una misma clave, irán a la misma partición. Sin embargo, si la clave es nula, el mensaje será enviado a alguna partición disponible al azar, utilizando un algoritmo de round-robin para balancear los mensajes a lo largo de las particiones.

VII-D. ¿Cuántos Brokers son necesarios?

No existe una fórmula que de la cantidad exacta pero existen diferentes elementos a considerar al momento de dimensionar un cluster para Kafka:

- ¿Cuánto espacio en disco se necesita para retener los mensajes durante el tiempo configurado? ¿Cuánto espacio en disco tendrá disponible cada broker? Ejemplo: si se precisa que el cluster sea capaz de retener 10 TB en datos y un broker puede retener 2 TB, entonces el tamaño mínimo para el cluster es de 5 brokers.
- ¿Cuál será el factor de replicación? Usar replicación siempre aumentará los requerimientos de almacenamiento, en al menos un 100 %. Siguiendo con el ejemplo, si consideramos que se replique al menos una vez, el tamaño mínimo del cluster sería de 10 brokers.
- ¿Qué capacidad tienen las interfaces de red? ¿Pueden manejar el tráfico esperado en caso de haber múltiples consumidores? ¿El tráfico será constante o existirán horas picos? Es decir, si la interfaz de red de un broker es usada en un 80 % de su capacidad durante horas picos, y hay dos consumidores de los datos, los consumidores no podrán estar al día al menos que hayan dos brokers. Adicionalmente, si el cluster tiene replicación, este funcionaría como un tercer consumidor de los datos que debe considerarse.

VII-E. Kafka Offsets

El offset define una posición dentro de una partición. Existen dos tipos de offsets:

- Current offset: Es un puntero al último mensaje enviado a un consumidor. De esta forma se logra que Kafka sepa cuál debe ser el próximo mensaje a enviar, evitando enviarle mensajes duplicados al consumidor.
- Committed offset: es un puntero al último mensaje que el consumidor confirmó que pudo procesar exitosamente. Sirve para evitar enviar el mismo mensaje a nuevos consumidores luego de un evento de rebalanceo de particiones (cuando se tiene un grupo de consumidores y se une un nuevo consumidor al grupo o falla un consumidor del grupo, es necesario hacer un rebalanceo para definir la asignación de particiones a cada consumidor del grupo). Estos offsets se almacenan en un topic interno de Kafka llamado __consumer_offsets.