Facultad de Ciencias

Diseño e implementación de una arquitectura distribuida para implantar un sistema de vigilancia de la calidad del aire

(Design and implementation of a distributed architecture for deploying an air quality monitoring system)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERIA INFORMÁTICA

Autor: David Gragera Iglesias

Director\es: Marta Elena Zorrilla Pantaleón

Ricardo Dintén Herrero

Agradecimientos

Me gustaría agradecer a mi familia y a mi pareja por haberme acompañado durante estos años en la realización del máster en ingeniería informática. También a aquellos compañeros y amigos con los que he compartido experiencias en la facultad.

Además de mi agradecimiento a mis dos tutores, Marta Elena Zorrilla Pantaleón y Ricardo Dintén Herrero, por haberme apoyado y ayudado durante la realización de este trabajo fin de máster. He sido recibido con gran apoyo en los periodos de mayores dudas y temores. Sin ellos no estaría donde me encuentro ahora mismo.

Gracias a todos.

Resumen

En este trabajo de fin de máster, se propone diseñar e implementar un sistema de vigilancia de calidad del aire siguiendo una arquitectura distribuida y escalable basada en tecnologías big data. El objetivo principal es refactorizar la aplicación actual desarrollada sobre una arquitectura centralizada, con el fin de permitir la ingesta de un mayor número de fuentes de datos en el futuro y visualizar las mediciones de la calidad del aire en tiempo real. Con el fin de lograr estos objetivos, se construye la solución basada en una arquitectura Lambda modificada que aprovecha tecnologías como Apache Spark y Apache Kafka.

Para evaluar la efectividad de la solución desarrollada, se llevan a cabo una serie de pruebas en la nube con diferentes configuraciones. Estas pruebas permitieron verificar que los requisitos especificados, entre otros, de rendimiento, se cumplieron satisfactoriamente. Estas mediciones procesadas se presentan a través de *dashboards*, así como las métricas de rendimiento del sistema implementado.

Palabras clave: procesamiento en tiempo real, IoT, arquitectura distribuida, computación en la nube.

Abstract

In this master's thesis, the proposal is to design and implement an air quality monitoring system following a distributed and scalable architecture based on big data technologies. The main objective is to refactor the current application developed on a centralized architecture to enable the ingestion of a larger number of data sources in the future and to visualize real-time air quality measurements. In order to achieve these goals, the solution is built based on a modified Lambda architecture that uses technologies as Apache Spark and Apache Kafka technologies.

To assess the effectiveness of the developed solution, a series of cloud-based tests with different configurations are conducted. These tests allowed us to verify that the specified requirements, including performance, were successfully met. Processed measurements are presented through dashboards, as well as performance metrics of the implemented system.

Keywords: real-time processing, IoT, distributed architecture, cloud computing.

Tabla de contenido

Índice de figuras

Índice de tablas

Índice de Cuadros

1. Introducción

Tradicionalmente, los proyectos para la ingesta, gestión y tratamiento de la información han sido desarrollados utilizado arquitecturas centralizadas, donde todo el procesamiento se realiza en el mismo servidor. Un caso real es el proyecto de vigilancia de la calidad del aire que desarrolla CIC y que va a ser utilizado en este TFM como caso de estudio para proponer una arquitectura distribuida alternativa utilizando tecnologías *big data*. Actualmente el servicio de telemetría ofrece medidas ambientales agregadas procedentes de tres fuentes de datos que emiten un caudal de unos 400 registros cada diez minutos. El interés del proyecto es escalar esta solución para que pueda tener una ingesta de un mayor volumen de datos (más fuentes) y ofrecer un cuadro de mandos que muestre las métricas al usuario en tiempo cuasi-real.

En el ámbito del *data streaming* existen, actualmente, dos arquitecturas predominantes: Lambda y Kappa. En el caso de ser implementadas sobre una plataforma distribuida y escalable, estas arquitecturas permiten satisfacer los requisitos para procesar grandes cantidades de datos de manera eficiente y en tiempos razonables, baste con agregar o reducir nodos de procesamiento.

Ligadas a estas arquitecturas, existen tecnologías que han sido ampliamente adoptadas en la industria para resolver problemas de procesamiento y análisis de grandes volúmenes de datos en tiempo real y en *batch*. Un ejemplo de estas tecnologías son: Apache Kafka, un sistema de gestión de colas de mensajes; y Apache Spark, un motor de procesamiento distribuido, que junto a otras han sido empleadas en este proyecto para implementar la plataforma de procesamiento de datos en tiempo real.

El objeto de este proyecto es por tanto diseñar e implementar una solución *big data* que permita un salto tecnológico que mejore el rendimiento y facilite el escalado del sistema para hacer frente al crecimiento del negocio, así como establecer un patrón para la construcción de aplicaciones de uso intenso de datos en la consultora informática.

La memoria se organizará del siguiente modo. En primer lugar, en la sección 2, se presentará una visión actual del proyecto, describiendo cada una de las partes que lo compone. En la sección 3 se realizará una exposición de las herramientas, *frameworks* y plataformas seleccionadas, indicando su finalidad en el proyecto. En la sección 4, se abordará el diseño de la solución, incluyendo los requisitos establecidos, así como la arquitectura propuesta y la organización de los componentes claves. En la sección 5 se describirá la implementación de estos componentes resaltando las partes de interés. En la sección 6, se expondrá el proceso de despliegue en la nube, así como el análisis y la discusión de las pruebas realizadas y los resultados obtenidos. Por último, en la sección 7 se presentarán las conclusiones del proyecto y se recogerán las posibles direcciones futuras de trabajo.

2. Sistema actual para el control de la calidad del aire

CIC ha desarrollado, en colaboración con ViewNext, el proyecto del sistema de vigilancia de la calidad del aire para el ayuntamiento de Madrid. Este se encuentra divido en distintos propósitos. En concreto, consta de tres aplicaciones desarrolladas en Java Enterprise Environment (JEE) que están alojadas en un servidor de aplicaciones Wildfly:

- Proyecto Web Service: Esta aplicación permite a los operadores realizar peticiones a través de puntos de entrada definidos. Entre estos, se incluye la recepción de mediciones enviadas de cada fuente de datos, como se muestra en la Figura 1. Estas mediciones se validan y se almacenan en la base de datos relacional.

- Proyecto Batch: Este componente lee las mediciones recogidas por el proyecto Web Service que se encuentran en la base de datos y aún no han sido procesadas. Aquí se llevan a cabo las operaciones de filtrado, recálculo y agregación de las mediciones, que posteriormente se insertan en la base de datos.

- Proyecto REST: Esta aplicación se encarga de proporcionar una comunicación efectiva entre el proyecto FRONT y la base de datos con el fin de poder obtener, transformar y mostrar información.

- Proyecto FRONT: Este último componente permite a los usuarios finales acceder a información relevante sobre la calidad del aire de la ciudad. Entre estas se disponen, a través de una interfaz gráfica, las mediciones recogidas, así como las alertas que se pueden generar al procesarse estos datos.

A pesar de la clara separación de aplicaciones, actualmente no se cuenta con un mecanismo para escalar horizontalmente ni distribuir el proyecto en distintos nodos. Por tanto, las aplicaciones están programadas bajo una estrategia monoprocesador. Esto puede ocasionar una pérdida de rendimiento cuando los datos superan las capacidades del servidor.

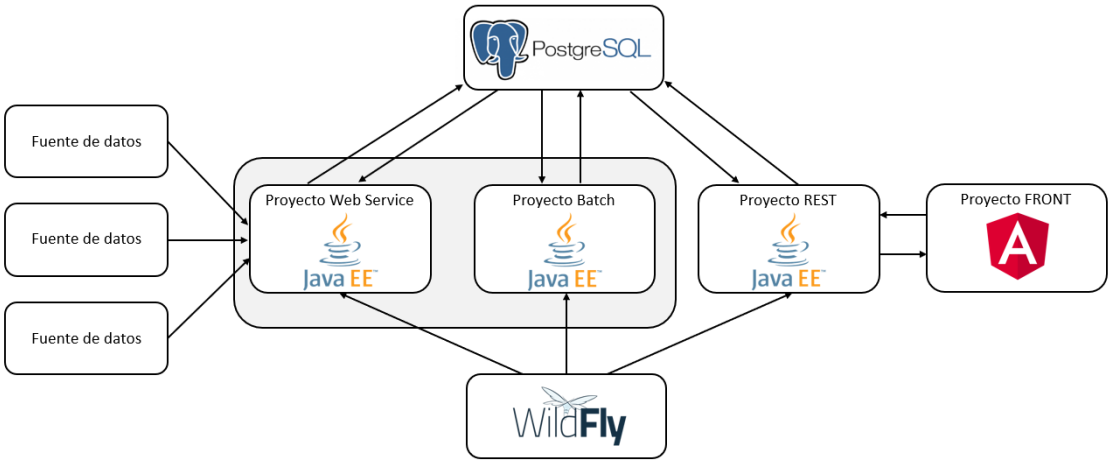


Figura 1. Arquitectura actual del proyecto de CIC.

Uno de los objetivos primordiales del proyecto de CIC es lograr el control de la calidad del aire en la ciudad, con el fin de notificar y actuar acorde a las medidas protocolarias establecidas por las autoridades competentes. Se estableció que, para lograr este objetivo, era necesario realizar en tiempo real la ingesta e incorporación, en la base de datos, de las mediciones captadas por los distintos analizadores o sensores, previa aplicación de los filtros precisos para garantizar la calidad técnica del proceso. Actualmente se obtienen mediciones cada 10 minutos, cuando se deberían de obtener como mínimo cada 10 segundos. Esto es debido a que la aplicación no soporta tal cantidad de datos en un intervalo tan pequeño, donde el cómputo del procesamiento de las mediciones entrantes es de 40 segundos.

A continuación, se explicarán en detalle el conjunto de datos, así como los proyectos Web Service y Batch, indicados en el recuadro gris de la Figura 1.

2.1. Conjunto de datos

En primer lugar, es esencial comprender el conjunto de variables que componen los datos a los que se van a hacer referencia durante todo el documento: las mediciones. Cada medición se capta desde una fuente de datos, la cual representa la zona geográfica donde se ubican las estaciones. Cada estación es un punto físico donde se encuentran los diferentes equipos asociados, que se encargarán de recoger los datos para el control del aire. Con el fin de recoger distintos tipos de medidas cada equipo tiene unas magnitudes asociadas como, por ejemplo, el ozono o el monóxido de carbono. Se explica gráficamente la estructura que tienen las fuentes de datos en la Figura 2.

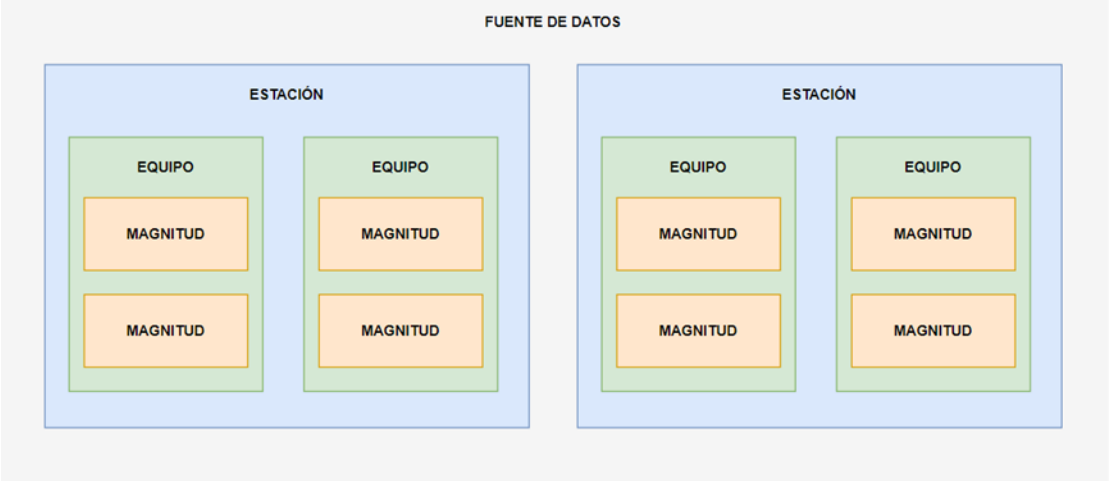


Figura 2. Estructura utilizada en las fuentes de datos.

Además de las variables mencionadas anteriormente, cada medición también incluye información sobre la fecha en la que se realizó, el estado que muestra si es válida o no, el intervalo que indica cada cuanto se realiza la medición, el índice que representa el número de muestras realizadas en el propio intervalo de tiempo, el valor promediado de todas las muestras en el intervalo y, opcionalmente, el nivel, que representa la altura a la que se ha realizado la medición.

Dependiendo de la fuente de datos emisora, las mediciones pueden tener que cumplir una serie de requisitos especiales. Entre estas fuentes se encuentra SODAR, en esta zona geográfica es de vital importancia que todas las mediciones tengan el parámetro nivel para poder medir a qué altura ha sido realizada la medición. Otro caso a tener en cuenta es la fuente de datos METEO, en esta zona geográfica el tipo de todos los equipos que generan mediciones deben ser "meteorológico", ya que se dedican exclusivamente a enviar datos meteorológicos.

2.2. Proyecto Web Service

Este proyecto se encarga de la recepción de las mediciones, donde se dispone de una conexión de tipo Web Service para recabarlas. Las fuentes de datos envían ficheros de tipo JSON cada 10 minutos de manera periódica con la información desarrollada anteriormente, un ejemplo de instancia es la que se muestra en el Cuadro 1.

```
{
  "recepcion": {
    "fuente_datos": "10.194.191.3",
    "estacion": 28122,
  },
  "mediciones": [
    {
      "analizador": "AV0080814915M",
      "estacion": 28122,
      "estado": "V",
      "fecha": "2000-06-17 00:00:00",
      "indice": 100,
      "intervalo": 600,
      "magnitud": 8,
      "valor": 1
    }
  ]
}
```

Cuadro 1. JSON de mediciones.

En cuanto los datos entran en el sistema, se procede a validarlos. Independientemente de la zona geográfica a la que correspondan las mediciones, todas estas deben pasar por una serie de controles antes de ser enviadas al área de trabajo. Estas incluyen la comprobación de la existencia en base de datos de la estación, equipo, magnitud, estado de la medición y el intervalo. Además de la comprobación del cumplimiento de los requisitos de las fuentes de datos SODAR Y METEO antes mencionadas.

En caso de que sean correctos y estén bien formados, se introducen a la base de datos como mediciones brutas, es decir, datos que todavía no se han tratado. Como se puede observar en el Cuadro 2, el sistema envía una respuesta indicando que las mediciones se han insertado correctamente.

```
{
   “resultado”: true,
   “mensaje”: {
         “mensaje”: “Las mediciones se han creado correctamente”,
         “errores”: null
   }
}
```


Cuadro 2. Respuesta correcta de la creación de mediciones.

Si se produce un error se le muestra al usuario a través de la respuesta, descartando todos los datos que se encuentren en el JSON enviado. Por ejemplo, en el Cuadro 3, se muestra el error cuando la estación no se encuentra definida en la base de datos.


```
{
   “resultado”: true,
   “mensaje”: {
         “mensaje”: “No se han podido crear las mediciones”,
         “errores”: [
               {
                     “detalle”: “No existe la estación con el código 28122”
               }
         ]
   }
}
```


Cuadro 3. Respuesta incorrecta de la creación de mediciones.

2.3. Proyecto Batch

El proyecto Batch realiza el procesado de las mediciones brutas, este se compone de tres etapas fundamentales: filtrado, recálculo y agregación (Figura 3). Estas desempeñan un papel fundamental en la obtención de información precisa y fiable. A continuación, se detallará cada una de estas etapas para comprender su función dentro del proceso.

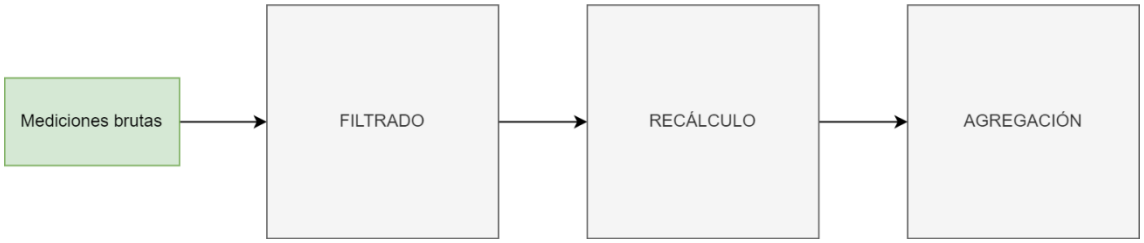


Figura 3. Flujo de los datos por cada una de las etapas.

2.3.1. Filtrado

En esta sección se realizan las verificaciones necesarias sobre los datos originales recibidos desde las diferentes fuentes de datos para garantizar su validez. Para ello, se llevan a cabo una serie de validaciones para asegurar la integridad de las relaciones entre las estaciones, equipos y magnitudes, donde el flujo realizado es el que se muestra en la Figura 4.

Se comprobará que el estado de la estación se encuentre activo y que no se haya dado de baja. En caso de que no se cumpla esta condición, se generará una alerta que indicará que “se han detectado datos de estaciones no activas”.

Posteriormente, se verificará si la relación entre el equipo y la estación se encuentra activa, ya que los equipos se pueden mover en el tiempo entre distintas estaciones. En caso de que no lo sea se generará una alerta que mostrará que “Se han detectado datos de equipo de medición inactivos”.

También se deberá comprobar que la relación entre la magnitud y el equipo esté activa. Si no se cumple esta condición, se enviará una alerta indicando que “Se han detectado datos de magnitud inactivos”.

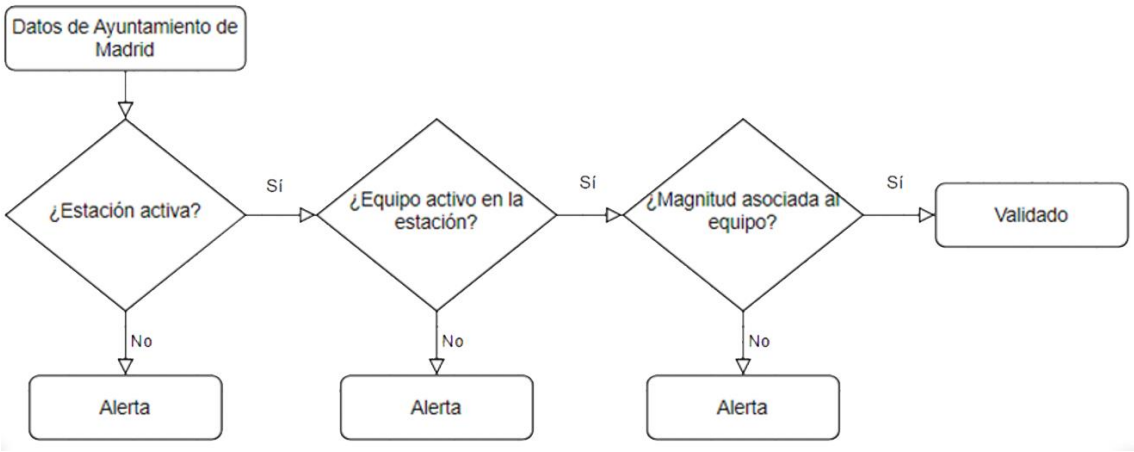


Figura 4. Flujo de filtrado de los datos del ayuntamiento de Madrid.

Una vez realizado el filtrado de las mediciones de Madrid, se debe realizar el flujo de la Figura 5, un filtrado de las mediciones que tienen ciertas magnitudes asociadas. Primero se comprueba si la magnitud asociada a la medición debe ser procesada, si no se encuentra en esta tabla se termina el filtrado. Si esta se debe tratar entonces se comprueba los valores límites mínimos y máximos asignados a la magnitud. Si estos se ven superados se modifica el valor a uno de los límites superados y el estado al indicado por el límite.

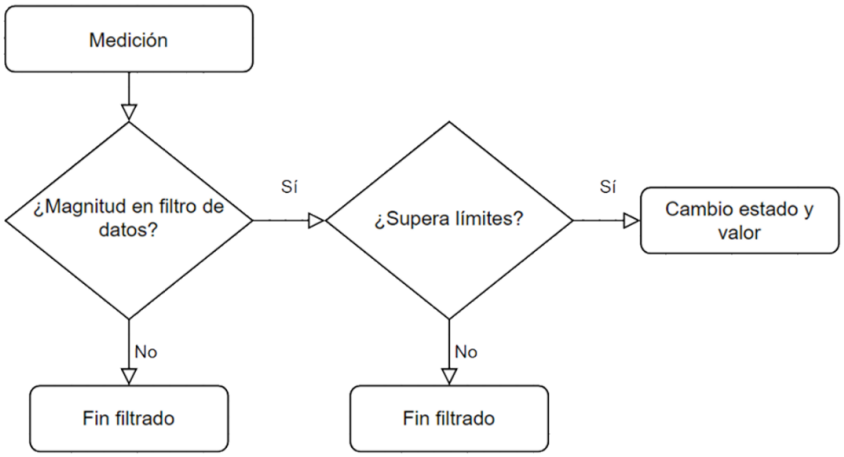


Figura 5. Flujo de filtrado de las magnitudes.

Si todas las comprobaciones se han superado de manera satisfactoria, se considerará que el dato de la medición es válido y pasará al área de trabajo donde se realizarán los recálculos correspondientes

2.3.2. Recálculo

Una vez que se ha completado el proceso de filtrado y transformación de los datos procedentes de los sistemas de origen, el siguiente paso implica el recálculo y ajuste de ciertos valores en función de magnitudes específicas, siguiendo reglas de negocio predefinidas. Este proceso se lleva a cabo para cada período de tiempo enviado y se centra en verificar la presencia de mediciones de determinadas magnitudes, agrupadas por estaciones y equipos.

En concreto, para cada período, se realiza una comprobación en relación con el grupo de óxidos de nitrógeno, asegurándose de que existan mediciones para las magnitudes de dióxido y óxido de nitrógeno. Si alguna de estas magnitudes falta en las mediciones del período, todas las mediciones agrupadas se etiquetan como "no válidas" para evitar que se realicen operaciones de agregación con ellas. Este enfoque garantiza la integridad y validez de los datos, asegurando la precisión y coherencia de los resultados obtenidos. En caso de que ninguna de estas magnitudes esté presente en el período, no se lleva a cabo ningún tipo de operación.

Este mismo proceso se aplica al grupo de magnitudes BTX, donde se verifica la disponibilidad de mediciones para tolueno, benceno y etilbenceno. Por último, en el grupo de HCT, se requiere la presencia de mediciones para hidrocarburos totales, metano.

2.3.3. Agregación

Por último, se lleva a cabo el proceso de agregación, donde se toman las mediciones, se realiza el promediado y se insertan en la base de datos. Es importante destacar que, para hacer el promedio, se considera la combinación de la estación, el equipo, la magnitud y la fecha.

En primer lugar, se encuentran los datos diezminutales, que corresponden a las mediciones con el periodo más corto disponible en la aplicación. En otras palabras, una medición con un intervalo de 10 minutos indica que el valor asociado es el promedio de esos 10 minutos.

Para los datos semihorarios, se deben recopilar 3 mediciones diezminutales para generar el promedio de sus valores y crear un dato semihorario. Por ejemplo, para obtener el dato semihorario de las 11:00, se requiere obtener las mediciones diezminutales de 10:30, 10:40 y 10:50.

En cuanto a los datos horarios, se necesitan 6 mediciones diezminutales para calcular el promedio. Por ejemplo, para el cálculo del valor horario de la 11:00 se utilizan las mediciones diezminutales de las 10:00, 10:10, 10:20; 10:30, 10:40 y 10:50.

Figura 6. Agrupación de mediciones semihorarias y horarias.

Los datos octohorarios requieren 8 mediciones horarias anteriores, incluyendo la hora de la agregación. Por tanto, si se desea obtener la medición octohoraria de las 10:00, se deben recopilar los datos de 03:00, 04:00, 05:00, 06:00, 07:00, 08:00, 09:00, 10:00.

Por último, se genera las mediciones diarias, este el promedio de todas las mediciones horarias en un día, en un total de 24 registros desde la 01:00 hasta las 00:00 del día siguiente.

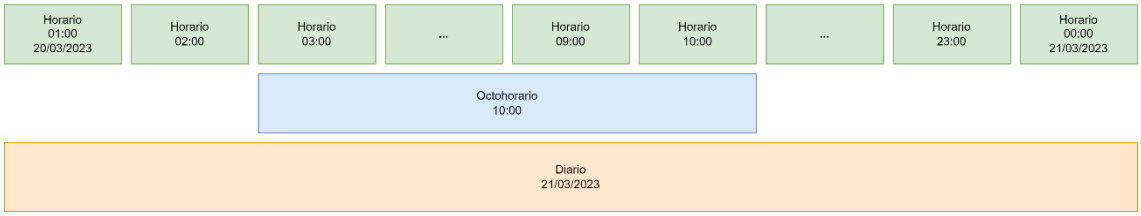


Figura 7. Agrupación de mediciones octohorarias y diarias.

2.4. Proyecto REST y FRONT

Para que el usuario pueda visualizar la información de las mediciones se ha desarrollado un entorno web a través de tecnologías FRONT y para la obtención de los datos se ha desarrollado un proyecto REST. Con el fin de poder mostrar las alertas generadas en el proceso de Filtrado se ha creado una ventana donde se indica por medio de una tabla estos errores como se muestra en la Figura 8.

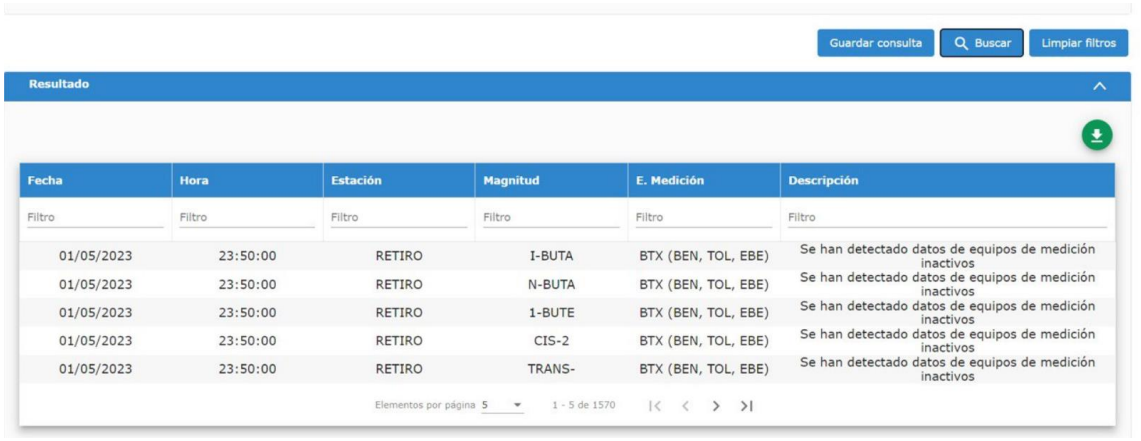


Figura 8. Tabla de alertas.

Para poder editar las mediciones que generan estas alertas se debe acceder a las mediciones que no han cumplido los requisitos establecidos y editar de forma manual cada una de estas, lo que se realiza a través del formulario que se muestra en la Figura 9.

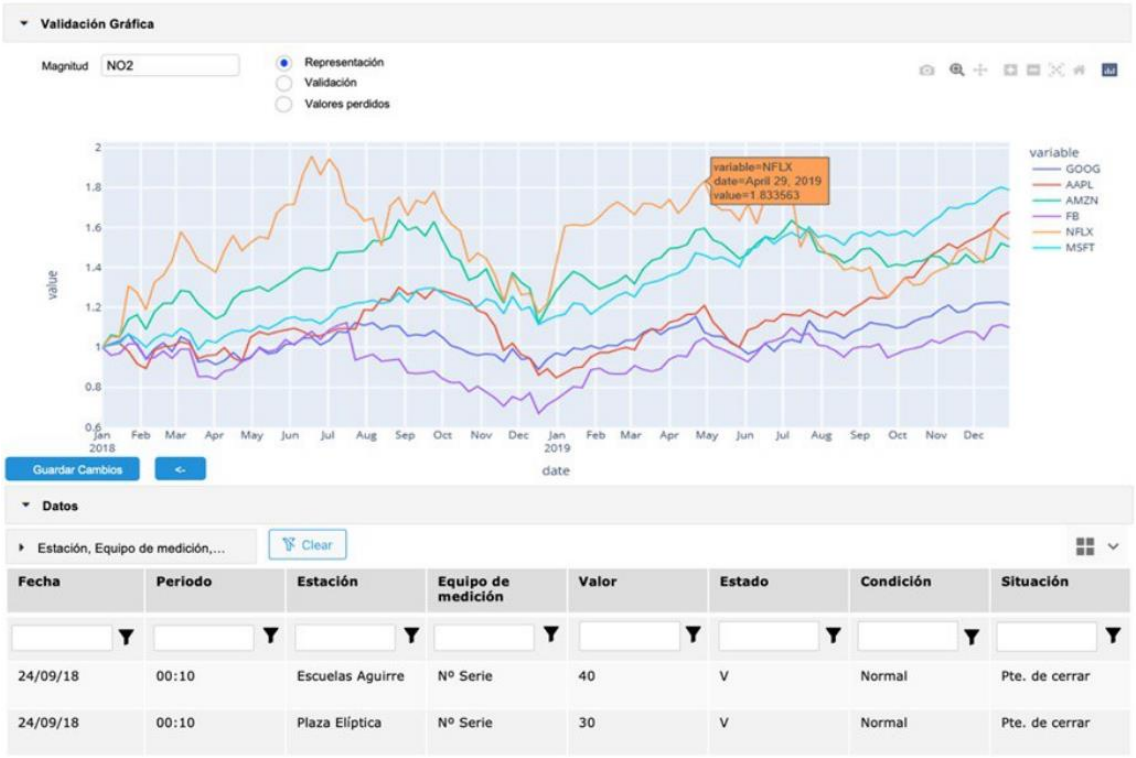


Figura 9. Gráfica y tabla de las medidas a modificar.

Una vez que se selecciona la medición en la gráfica que se quiere modificar, se accede al detalle mostrado en la Figura 10.

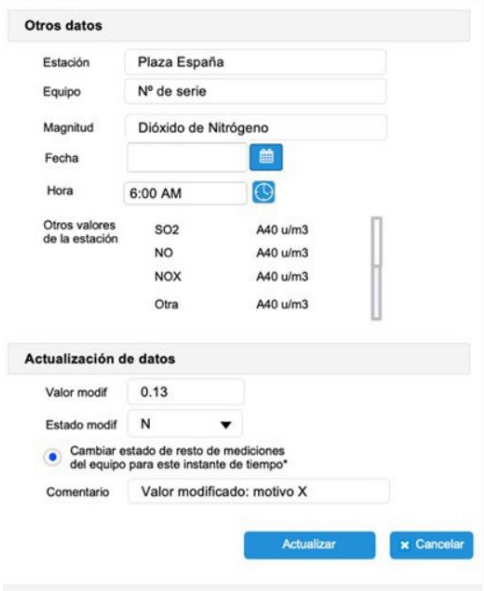


Figura 10. Detalle de la medición.

En caso de querer observar aquellas mediciones que se han realizado correctamente hay un apartado dentro de la aplicación donde el usuario final puede visualizarlas (ver la Figura 11).

Estación	E.Medición	Paralelo	Sustitución	Magnitud	Valor	Estado	Intervalo	Índice
Estación	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
ARTURO SORIA	OZONO	No	No	O3	78	Válido	Diez Minutos	96
BARAJAS PUEBLO	OXIDOS DE NITROGENO	No	No	NOX	12	Válido	Diez Minutos	90
BARAJAS PUEBLO	OXIDOS DE NITROGENO	No	No	NO	1	Válido	Diez Minutos	90
BARAJAS PUEBLO	OXIDOS DE NITROGENO	No	No	NO2	11	Válido	Diez Minutos	90
BARAJAS PUEBLO	OZONO	No	No	O3	89	Válido	Diez Minutos	90
BARAJAS PUEBLO	TEMPERATURA INTERNA	No	No	TMI	20	Válido	Diez Minutos	90

Figura 11. Tabla de mediciones válidas.

3. Tecnologías utilizadas

En este apartado se van a describir brevemente las tecnologías utilizadas dentro del proyecto, señalando las características más relevantes que las hacen las opciones más viables para el propósito que se persigue.

Entre las tecnologías más destacadas en este campo se encuentran las plataformas de mensajería distribuida y los motores de procesamiento de datos en memoria. Las plataformas de mensajería distribuida permiten la transferencia de grandes cantidades de datos en tiempo real entre diferentes sistemas y aplicaciones de manera escalable y confiable, mientras que los motores de procesamiento de datos distribuidos y en memoria permiten el análisis y procesamiento de grandes volúmenes de datos de manera rápida y eficiente.

3.1. Apache Kafka

En el ámbito de la transmisión de la información distribuida existen varias tecnologías, entre las que destacan son Apache Kafka y RabbitMQ. Aunque ambas son sólidas, elegí Kafka porque cuenta con un amplio apoyo y una comunidad activa, lo que ha llevado a su crecimiento y madurez en el campo del *big data*. Esto ha contribuido a enriquecer la tecnología, proporcionando una extensa documentación, solución de problemas y constante mejora de características y rendimiento [4].

Apache Kafka es una plataforma de mensajería distribuida, enfocada a la transmisión de datos en tiempo real entre aplicaciones y sistemas. Kafka está diseñado para manejar grandes volúmenes de datos y proporciona una escalabilidad horizontal, lo que significa que es fácil agregar más nodos para manejar un mayor volumen de datos. Cabe destacar que permite replicar los datos para mejorar la tolerancia a fallos, haciendo que sea muy resistente a caídas de servidores [2].

Una de las ventajas que otorga Apache Kafka es su arquitectura basada en particiones, la que garantiza un procesamiento de los flujos de alto rendimiento y en tiempo real. Esto permite dividir la carga de trabajo entre múltiples nodos, dando una alta escalabilidad y disponibilidad. Algo de vital importancia en escenarios donde la ingesta de datos es masiva y se requiere

procesamiento en tiempo real. Además, su enfoque en el procesamiento en tiempo real y su capacidad para manejar grandes volúmenes de datos lo convierten en una opción atractiva para muchas empresas, como Adidas para procesamiento de datos o Netflix para monitorización en tiempo real [3].

Sin bien RabbitMQ es una excelente opción para escenarios en los que se requiere enrutamiento de mensajes más complejos o integraciones con sistemas heterogéneos, Apache Kafka se destaca por su facilidad para escalar y su alto rendimiento al manejar grandes volúmenes de datos en tiempo real. Además, se integra con una variedad de sistemas y aplicaciones, siendo altamente configurable y satisfaciendo las necesidades específicas de cada proyecto [1].

Se debe tener en cuenta que Kafka es una arquitectura productor-consumidor, un patrón de uso de las tipologías de arquitecturas de colas de mensajes. En estas existe un emisor de los datos (productor) y un receptor (consumidor), pero con la diferencia de que estos mensajes no se envían directamente al destinatario, sino que el productor lo publica en un tópico y los diferentes consumidores se subscriben a uno o múltiples tópicos para obtener los datos como se puede observar en la Figura 12. Hay que destacar que estos tópicos se encuentran dentro de brokers, el equivalente a un servidor.

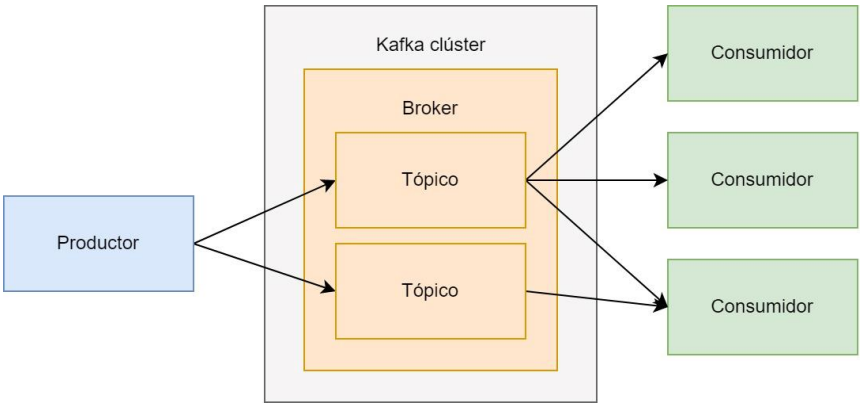


Figura 12. Representación de la arquitectura básica de Apache Kafka.

Dentro de estos tópicos se pueden llevar a cabo particionado y replicación. El particionado consiste en dividir un tópico en segmentos más pequeños, cada mensaje se dirige a una partición específica según un algoritmo de selección de partición, como se observa en la Figura 13. Por otro lado, la replicación implica crear copias del mismo tópico. Los mensajes se almacenan en varios brokers o servidores. Es importante destacar que un tópico puede estar replicado sin necesidad de estar particionado, y viceversa. Esta diferencia es importante remarcarla ya que en caso de que un tópico no esté replicado y un nodo experimente una caída, parte de la información asociada quedará inaccesible.

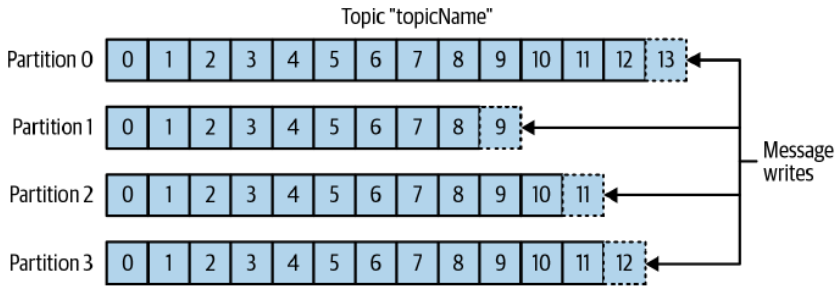


Figura 13. División de un tópico en particiones [2].

Las aplicaciones que leen los tópicos de las particiones son los consumidores. En Kafka se pueden crear grupos de consumidores, estableciendo que cada partición solamente pueda ser leída por uno de los consumidores y evitando que varios receptores hagan el procesamiento con los mismos datos. En la Figura 14 se muestra como cada consumidor del grupo apunta a una partición distinta.

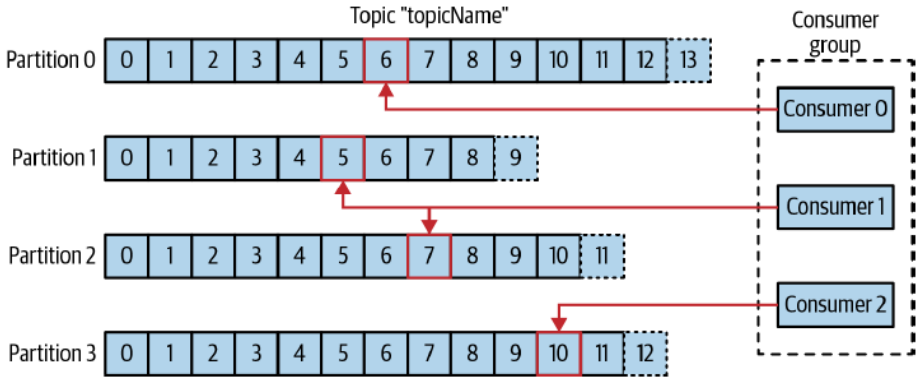


Figura 14. Consumidores del mismo grupo toman diferentes particiones [2].

3.2. Apache Zookeeper

Apache ZooKeeper es un servicio de coordinación utilizado para facilitar la gestión de aplicaciones distribuidas. En esencia, ZooKeeper proporciona servicios que ayudan a las aplicaciones a coordinar tareas y gestionar recursos en entornos distribuidos, asegurando la coherencia y confiabilidad en sistemas complejos.

Apache Kafka emplea ZooKeeper. Su adopción se debe a que este es capaz de brindar disponibilidad de los metadatos del clúster, sincronizar particiones, gestionar fallos y ofrecer capacidades de control de acceso y autorización, mejorando la confiabilidad y escalabilidad del sistema [2].

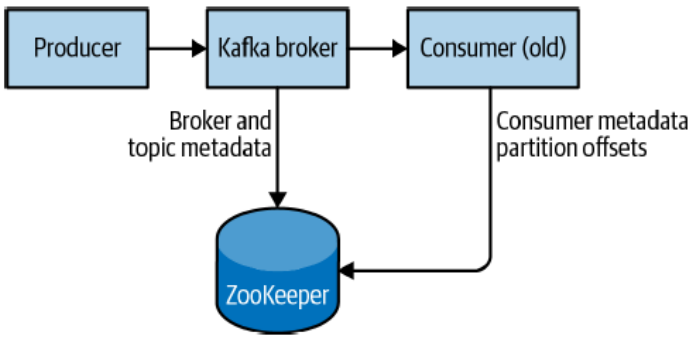


Figura 15. Estructura de la sincronización de Apache Zookeeper en Apache Kafka [2].

3.3. Apache Spark

Al elegir un motor de procesamiento de información, surgen diversas opciones, entre las cuales destacan Apache Flink, Apache Spark o Apache Storm. Aunque anteriormente Apache Storm era una opción popular, su adopción ha disminuido en los últimos años en comparación con las otras dos alternativas.

Apache Spark y Apache Flink son las dos tecnologías más utilizadas en la actualidad. Aunque comparten algunas similitudes, también presentan diferencias significativas en su enfoque de procesamiento. Spark utiliza un modelo de *micro-batch*, donde los datos se procesan en pequeños lotes de intervalos regulares.

Por otro lado, Flink se destaca por realizar el procesamiento de los datos en *streaming* de forma nativa, lo que brinda baja latencia y capacidad de procesar los flujos exactamente una vez. Mientras que Spark tiene una latencia más alta debido a la naturaleza de los lotes, logra procesar una mayor cantidad de información en comparación con Flink [4].

Es relevante señalar que Spark tiene una comunidad mucho más amplia, en la que se encuentra una mayor cantidad de documentación, tutoriales y recursos frente a Flink. Y, a pesar de ser una tecnología más reciente, la comunidad de Flink ha experimentado un crecimiento continuo en la actualidad.

Ambas tecnologías son buenas opciones para el procesamiento de datos. Sin embargo, en este proyecto se ha optado por Apache Spark debido a la facilidad que brinda para el desarrollo, su capacidad de manejar lotes y el respaldo que ofrece la comunidad, que brinda una gran cantidad de recursos y documentación.

Para lograr esto, Spark cumple con una serie de componentes claves, como su enfoque en el procesamiento de datos, es decir, cargar los datos que recibe desde los sistemas de almacenamiento y realizar el procesamiento necesario en memoria. Además, ofrece soporte para diferentes sistemas de almacenamiento, como Azure Storage, y sistemas de mensajería por colas, como Apache Kafka.

Otro aspecto clave es el soporte de una gran variedad de librerías, desde las estandarizadas hasta las de terceros de la comunidad con la que cuenta. Entre las que forman el núcleo de esta tecnología se encuentran Spark SQL y Datasets, Spark Streaming, MLib y GraphX [5].

- Spark SQL y Datasets: Permite utilizar datos estructurados, en la que se permite hacer uso de sentencias SQL o de funcionalidades de Datasets para hacer consultas, filtrado, transformaciones o agregaciones entre otras sobre los datos.

- Spark Streaming: Facilita el procesamiento de datos en *streaming*, en el que es escalable y tolerantes a fallos. Sin embargo, hay que remarcar que no es *streaming* al uso, ya que logra esto a través de *micro-batches* de unos 100 milisegundos.

- Spark MLib: Permite el aprendizaje automático en Spark, conteniendo herramientas para el uso de *machine learning*.

- Spark GraphX: Utilizado para la realización de operaciones y computación en grafos.

Principalmente se han utilizado Spark Streaming para la recepción y envío de la información obtenida a través de Kafka y Spark SQL y Datasets con el fin de procesar los datos recibidos.

Cabe destacar que actualmente la librería de Streaming ha dejado de recibir soporte dando paso a Structured Streaming, un nuevo motor de *streaming* más fácil de manejar. Como se ha indicado anteriormente, el procesamiento en *streaming* funciona a través de *micro-batches* de 100 milisegundos, en Structured Streaming, se ha llegado a reducir hasta 1 milisegundo [6].

Ahora los datos que se reciben se almacenan en una tabla de datos ilimitada y cuando se requieran los datos dependiendo de las necesidades se podrán obtener nuevos resultados, todos o los actualizados. Además, una de las grandes ventajas es el uso directo de los *Datasets* en lugar de los *Dstreams*, los cuales se utilizaban en la versión anterior, haciendo que se simplifique la manipulación de los conjuntos de datos.

A lo largo del proyecto se ha hecho uso de los *Datasets*, estos son conjuntos de datos con una cierta estructura y distribuidos por medio de un clúster de Spark. Cada transformación realizada sobre un *Dataset* en Spark retorna un *Dataset* nuevo con la transformación realizada, por lo tanto, cada uno es inmutable una vez es inicializado [7].

A pesar de que Structured Streaming ha experimentado un considerable desarrollo en cuanto al procesamiento en tiempo real, hay un componente esencial que no tiene el mismo funcionamiento que su versión anterior: las ventanas.

En Structured Streaming, se dispone de diversos tipos de ventanas de procesamiento, como *tumbling windows* (ventanas de tiempo que dividen el flujo de datos en intervalos de tiempo fijos y sin solapamientos), *sliding windows* (ventanas deslizantes que dividen los datos en ventanas superpuestas del mismo tamaño) o *session windows* (ventanas de sesión que agrupan datos en función de sesiones activas con períodos de inactividad entre eventos), con funcionalidades como son la recuperación de información y posterior tratamiento en caso de llegada tardía de datos [8]. Sin embargo, todas estas funcionalidades se aplican a funciones de conteo o agregaciones. Si se desean obtener todos los datos que llegan en una ventana para realizar transformaciones o modificaciones, no es posible.

Por este motivo, en este proyecto ha sido necesario recurrir a la versión anterior de procesamiento de datos, Spark Streaming, que sí ofrece esta funcionalidad para obtener y tratar todos los datos de una ventana. Se espera que en futuras versiones en Structured Streaming se resuelva esta problemática, añadiendo las características que se encuentran disponible en la versión anterior.

3.4. PostgreSQL

Para la persistencia de la información se ha hecho uso de PostreSQL, un potente sistema de base de datos relacional de objetos de código abierto que utiliza el lenguaje SQL combinado con muchas características que almacenan y escalan de forma segura las cargas de trabajo de datos más complicadas. Este ha sido proporcionado por el cliente como uso obligatorio, sin embargo, cuenta con características que lo convierten en una herramienta muy adecuada para el propósito utilizado [9].

PostgreSQL se ejecuta en todos los principales sistemas operativos, cuenta con funcionalidades para la integridad de datos o concurrencia, además de cumplir con el principio ACID, es decir,

garantizar la atomicidad (se ejecuta todo o nada de las transacciones), consistencia (solo se ejecutan aquellas operaciones que se pueden concluir), aislamiento (las transacciones no afectan a otras) y durabilidad (una vez que la transacción se ha realizado, aunque el sistema falle, se garantiza que se persiste el cambio), contando con una sólida comunidad.

3.5. Druid

Druid es una base de datos de alto rendimiento diseñada para el análisis de datos en tiempo real y por lotes. Se hace uso principalmente cuando se necesitan consultas rápidas con una gran cantidad de información en tiempo real de una aplicación [10].

La implementación de Druid se realiza en un clúster escalable con nodos que cumplen funciones específicas, por ejemplo, se pueden asignar más recursos para consultas de datos mientras que se tiene el mínimo de recursos para la ingesta. Esto permite que se utilicen tanto en ordenadores con pocos recursos como en despliegues a gran escala con una gran cantidad de nodos para manejar cargas de trabajo más exigentes [11].

Al igual que Kafka, Druid hace uso de Apache Zookeeper para gestionar todas las consultas y mantener la consistencia en el clúster. Además, su arquitectura se basa en servicios escalables e independientes entre sí para la consulta, ingesta y orquestación. Cada servicio puede configurarse con los recursos necesarios para abordar las cargas de trabajo requeridas.

Entre sus características se encuentran las siguientes:

- Formato de almacenamiento en columnas: Este enfoque mejora el rendimiento de las búsquedas, ya que solo se accede a las columnas necesarias para la consulta, lo que reduce la cantidad de datos a leer y, por tanto, un mejor rendimiento en las respuestas de las consultas.

- Sistema distribuido escalable: Como se mencionó anteriormente, Druid es una base de datos distribuida, lo que permite escalar horizontalmente añadiendo más nodos según se requiera para manejar un mayor volumen de datos o consultas más complejas. Esto garantiza un rendimiento óptimo en aplicaciones con alta demanda.

- Tolerante a fallos: Gracias al uso de su mecanismo de almacenamiento llamado “Deep storage”, el cual almacena los segmentos de datos que van llegando al sistema, Druid es tolerante a fallos en caso de que se utilice un almacenamiento compartido y distribuido (ver Figura 16). Esto consigue que, aunque todos los servidores de Druid fallen, los datos se mantengan y no se pierda. Además, la replicación de los datos mejora la tolerancia a fallos haciendo posible acceder a los datos incluso ante la caída de algún nodo.

- Mecanismo de ingesta con Kafka nativo: Al ser del ecosistema de Apache, Druid ofrece una integración nativa con Kafka. Esto facilita la configuración de una conexión con Kafka, ya que solo se le debe indicar la dirección del *broker* y el nombre del tópico, sin necesidad de requerir de mecanismos de terceros adicionales. La ingesta de datos desde Kafka se realiza mediante la configuración de un fichero llamado “spec”, lo que hace que la implementación en la aplicación sea sencilla y rápida.

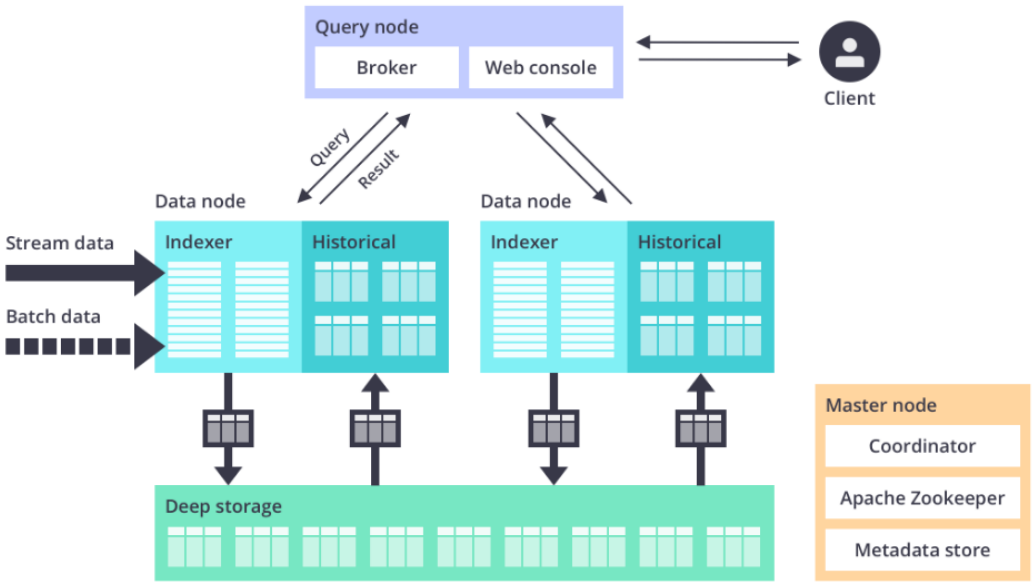


Figura 16. Arquitectura de Druid [9].

En comparación con otras tecnologías como Apache Cassandra (base de datos distribuida) o MongoDB, Druid destaca por su arquitectura orientada a consultas rápidas en tiempo real y su integración nativa con Kafka. Estas características hacen de Druid una opción sólida para el proyecto.

3.6. Grafana

Grafana es una plataforma de visualización y monitorización de los datos que ofrece a los usuarios la capacidad de consultar, visualizar y generar alertas a través de la creación de paneles personalizados. Esta herramienta permite analizar datos en tiempo real provenientes de diversas fuentes, incluyendo bases de datos y otros sistemas [12].

Dentro del mercado de soluciones de visualización de datos, existen otro tipo de plataformas como es Splunk, ampliamente utilizado para el análisis de logs y datos de registro; o Power BI, una herramienta de Microsoft con integración de diversas fuentes de datos y opciones avanzadas de análisis. Sin embargo, Grafana destaca en una serie de características que la hacen ideal para el propósito utilizado en este proyecto:

- Diversidad de fuentes de datos: Grafana ofrece una amplia gama de conectores, con lo que permite consolidar datos de múltiples fuentes en un solo proyecto, lo que facilita la integración y el análisis de la información de distintos sistemas.

- Soporte para datos en tiempo real y temporales: Grafana se caracteriza por su habilidad en representar datos de tiempo real y datos temporales. Los usuarios pueden personalizar los gráficos según las necesidades específicas, lo que facilita la compresión y toma de decisiones.

- Código abierto: Al ser una herramienta de código abierto, es gratuita y cuenta con una comunidad activa que contribuye al desarrollo y mejora constante de la plataforma. Esta comunidad proporciona recursos, soporte y nuevas funcionalidades.

Por todo lo descrito anteriormente, Grafana es una herramienta que cuenta con las características que se requieren para la visualización de los datos en este proyecto. Su capacidad de integrar diversas fuentes de datos, el análisis de la información en tiempo real y temporal, y su naturaleza de código con una comunidad activa hacen de Grafana una buena elección para permitir una visualización efectiva de los datos.

4. Diseño de la solución distribuida

La etapa de diseño desempeña un papel fundamental en este proyecto, ya que en ella se establecen los requisitos esenciales para la definición de la solución. Además, en este apartado se abordará la decisión de elegir la arquitectura más adecuada entre Lambda y Kappa.

4.1. Requisitos

Para cumplir con los objetivos del proyecto CIC y mejorar la arquitectura existente, se han identificado los siguientes requisitos a satisfacer:

- Optimización del procesamiento: La optimización del procesamiento es fundamental. Esto implica la implementación de una solución distribuida y escalable utilizando tecnologías de big data para manejar un volumen, como mínimo, de 2000 registros por segundo de ingesta.

- Tiempo de procesamiento eficiente: Actualmente el tiempo de procesamiento es de 40 segundos para 400 registros. Se establece como objetivo reducir este tiempo por debajo del segundo.

- Granularidad en los datos: Se busca alcanzar una mayor granularidad en los datos, logrando un promedio de ventana de 1 minuto para una visualización más detallada y precisa.

- Visualización en tiempo real y batch: Es necesario habilitar la visualización de los datos procesados en tiempo real, así como las agregaciones en streaming y en batch. Esto permitirá una comprensión completa de los datos en diferentes contextos.

- Monitorización de rendimiento: Se requiere la representación gráfica de métricas de rendimiento del sistema para evaluar su eficacia y tomar decisiones de configuración de despliegue de la solución.

- Cumplimiento de los requisitos de integridad y las reglas de validación existentes en el procesamiento de los datos en el sistema actual de CIC.

4.2. Arquitectura Lambda o Kappa

Como se ha explicado anteriormente, al surgir el paradigma de los sistemas distribuidos, también surgieron nuevas arquitecturas diseñadas para satisfacer las necesidades del análisis de datos y obtención de resultados lo más rápido posible. Entre estas arquitecturas destacan Lambda y Kappa, en la que su diferencia principal es la forma en la que manejan el flujo de datos.

Kappa se enfoca en el procesamiento de los datos en tiempo real, donde todas las operaciones se realizan en *streaming*. En este tipo de arquitectura priman la rapidez y disponibilidad de los resultados. Sin embargo, en el caso de que se requiera realizar agregaciones en *batch* con lotes de gran tamaño, no sería óptimo ya que consumiría una gran cantidad de recursos al tener que mantener la información en memoria y puede no ser del todo preciso si llegan datos fuera de orden.

Por otro lado, Lambda combina el procesamiento en *stream* con *batch*, permitiendo que la información sea dirigida a ambas capas antes de ser presentada en la capa de resultados. Sin embargo, en este contexto, resultaría inapropiada debido a la duplicación de los datos en ambas capas.

En este trabajo se ha planteado una modificación de esta última arquitectura para cumplir con los requisitos. Esto implica que el flujo de datos desde que se insertan las mediciones hasta el recálculo se tratará en el procesamiento *stream* teniendo la información en tiempo real y almacenando ésta en la base de datos, mientras que las agregaciones se realizarán en *batch* a partir de la información procesada.

Con ello se sigue manteniendo una arquitectura muy similar a Lambda, adecuándose a los requisitos del proyecto para realizar las agregaciones de forma correcta. Posteriormente habrá una capa de resultados para poder consultar la información que se filtra en el momento en tiempo real y aquella que se agrega en *batch*, sirviendo además para poder monitorizar una serie de métricas de rendimiento de Apache Spark y Kafka (ver Figura 17).

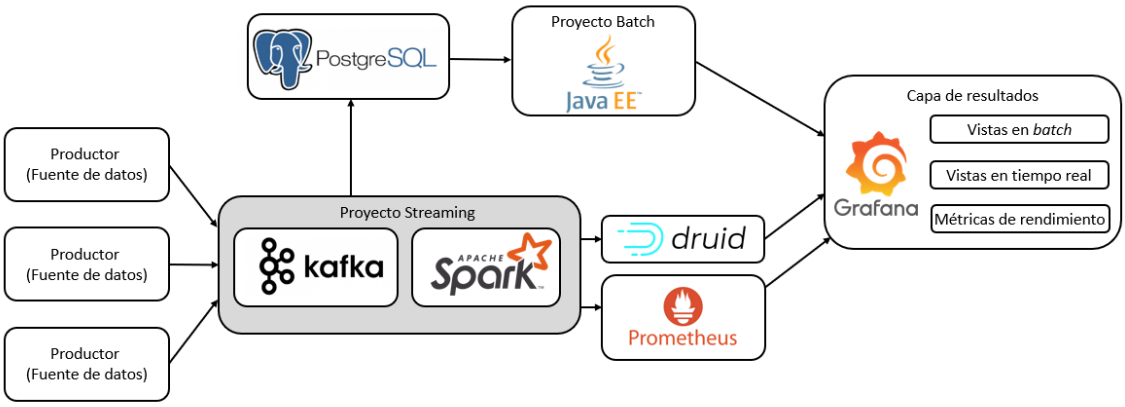


Figura 17. Arquitectura de la solución propuesta.

5. Implementación

En la Figura 18, se puede observar el flujo de trabajo completo en el que se realiza en el procesamiento *streaming*, el cual se irá explicando a continuación.

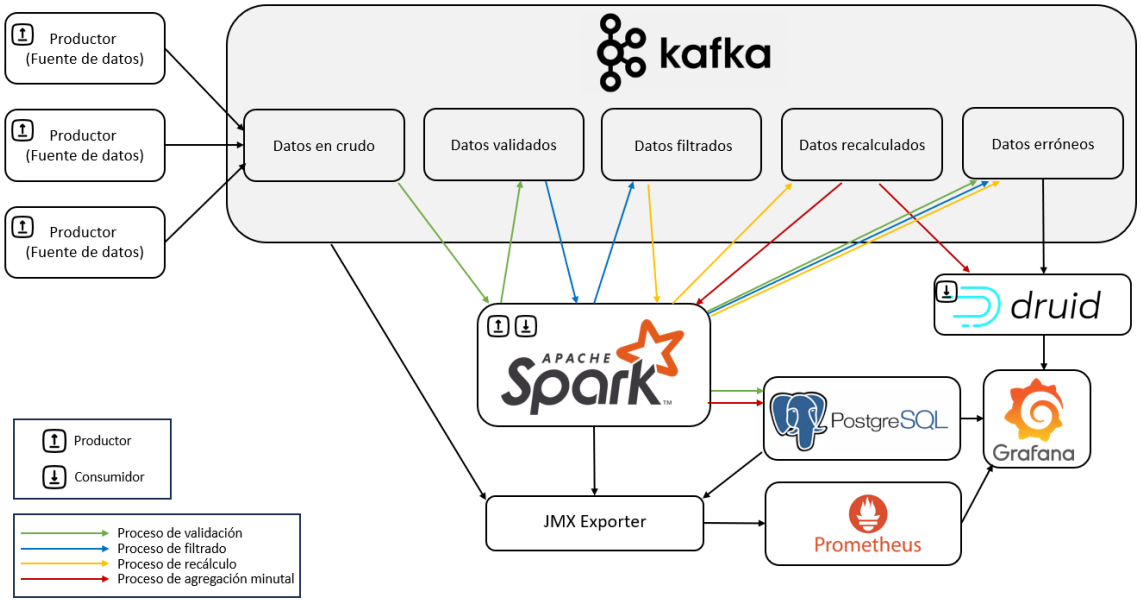


Figura 18. Esquema del procesamiento en streaming.

5.1. Entrada de datos

En la aplicación actual, todas las mediciones se recogen a través de un punto de entrada de tipo Web Service. Uno de los cambios realizados referente a esto es que las fuentes de datos que envían las mediciones actúen como productores y publiquen directamente los mensajes en el tópico designado para ello, de este modo se elimina cualquier cuello de botella que podría estar generando este único punto de entrada. Es decir, que cada fuente de datos se comportará como un productor independiente.

Para poder simular este escenario, se ha desarrollado un programa que deserializa los JSON que contienen las mediciones. Este lee cada medición que se encuentra en el fichero y las publica en el tópico, lo que permite simular el envío de datos desde las diferentes fuentes.

Para garantizar que la información sea coherente, se debe modificar cada medición, agregando un nuevo campo a cada una de ellas que identifica la fuente de datos de la que provienen. De esta manera, cada medición se puede tratar de forma independiente, permitiendo el procesamiento en tiempo real. Un ejemplo tras aplicar los cambios sería el que se muestra en el Cuadro 4, donde se observa que se han eliminado las etiquetas de recepción y mediciones.


```
{"fuente_datos":"28079095", "intervalo":600, "estacion": 8922,"magnitud":1,
"equipo":"AY0013818005U", "nivel":6, "fecha":"2022-10-13 01:00:00", "valor":1,
"estado":"Z","indice":11}
```


Cuadro 4. JSON de medición.

5.2. Diseño Kafka

Como consecuencia de la inmutabilidad de los tópicos, se requiere la creación de un nuevo tópico en Kafka por cada transformación que se realiza en Spark. Además, de este modo, se consigue una separación clara de las responsabilidades de cada uno de ellos, mejorando la modularidad y comprensión del flujo de datos. Cabe destacar que para el mantenimiento y depuración del sistema ha sido de gran utilidad crear tópicos con diferentes propósitos. En caso de que en el futuro se requieran de nuevos procesos o cambios, al estar modularizado no supondrá ningún problema para implementar nuevas funciones.

Es importante remarcar que al tener esta separación de tópicos se consigue que, en caso de caída de un proceso Spark, Kafka mantiene almacenadas un determinado tiempo las mediciones, lo que permite recuperar el estado y reanudar el procesamiento de los datos sin pérdidas.

También permite tener diferentes tecnologías en el procesamiento, en los dos primeros bloques de Spark se hace uso de Structured Streaming mientras que se utiliza Spark Streaming en los dos últimos.

Por ello, teniendo en cuenta las características de Kafka y con el propósito de implementar una aplicación modular y mantenible, se han definido los siguientes tópicos:

- Datos en crudo (entrada_topic): Mediciones brutas que llegan desde las fuentes de datos.
- Datos validados (filtrado_topic): Mediciones que han sido validadas y van a ser filtradas.
- Datos filtrados (recalculo_topic): Mediciones filtradas y van a ser recalculadas.
- Datos recalculados (agregacion_topic): Mediciones recalculadas y van a ser agregadas.
- Datos erróneos (errores_topic): Mediciones que han sufrido algún error en el proceso y van a ser almacenadas en Druid.

5.3. Diseño Spark

En el desarrollo de la implementación de este proyecto se ha hecho gran uso de las funciones Spark SQL, en la que ofrece una interfaz sencilla que permite trabajar tanto con datos estáticos como en *streaming*, además de contar con un motor que permite la optimización de las consultas de forma automática.

Por cada funcionalidad descrita durante el proyecto, es decir, Validación, Filtrado, Recálculo y Agregación, se han creado bloques por cada uno para procesar las mediciones en Spark, con ello se consigue que el proyecto sea modular. Esto permite que se minimicen las dependencias entre las diferentes partes, logrando una mayor flexibilidad y mantenibilidad en cada una de ellas, lo que facilita la corrección de errores o desarrollo de evolutivos [13].

Cabe destacar el tópico de datos erróneos, que está íntegramente enfocado a que todos los errores controlados que transcurran en las aplicaciones, como por ejemplo que no se encuentre en base de datos el código del equipo, se envíen al tópico con un mensaje descriptivo del error que se está produciendo.

5.3.1. Validación y Filtrado

A continuación, mediante programas Spark se validarán los datos, verificándose el cumplimiento de las reglas indicadas en el apartado 2.2 y se alimentará el tópico de "datos validados". Seguidamente el programa el programa Filtrado cumplirá con los requisitos establecidos en el apartado 2.3.1 enviando al tópico de Kafka "datos filtrados" las mediciones procesadas y en caso de que haya algún error al tópico "datos erróneos".

En los bloques de Validación y Filtrado se ha hecho uso de Spark Structured Streaming, que permite el procesamiento de los datos a medida que se van recibiendo por el tópico de Kafka. Para lograr los requisitos establecidos se han hecho uso de una gran cantidad de funciones que proporciona SparkSQL.

Con el fin de cargar todos los datos necesarios de base de datos y guardarlos en *Datasets* específicos para posteriormente hacer las consultas pertinentes contra las mediciones que llegan desde Kafka se ha utilizado la función que se muestra en el Cuadro 5.

```
// Cargar los datos de la tabla "estacion" y seleccionar la columna "id" y "codigo"
fuentesDatosDB=reader.option("dbtable", "equipo").load().select("id", "codigo").cache();
```

Cuadro 5. Lectura de base de datos.

Se ha utilizado en gran medida las funciones join que se proporcionan, para filtrar los *Datasets* con lo obtenido con la base de datos o realizar selecciones de datos necesarios para validar las mediciones (ver Cuadro 6).

```
// Validar las mediciones con los equipos usando una operación inner join
mediciones = mediciones.join(equiposDB, mediciones.col("equipo").equalTo(equiposDB.col("codigo")),
"inner")
   .select(mediciones.col("*"), equiposDB.col("id").as("equipo_id"));
```

Cuadro 6. Uso de función join.

Además, como se muestra en el Cuadro 7, para la unión de varios Datasets con el fin de enviar todo en uno mismo se debe hacer uso de la función "*union*" teniendo en cuenta que deben tener el mismo número de columnas en ambos. Con ello se solventa también el problema de que pueda haber duplicados al juntar ambos.

```
// Union de todos los errores
errores = errores.union(erroresEquipo);
```

Cuadro 7. Uso de función union.

Al enviar las mediciones al tópico de Kafka se debe tener en cuenta que para ello hay que asignarle varias opciones donde se indica la IP donde se encuentra el servidor de Kafka, el tópico al que se le quiere enviar, cada cuanto tiempo se quiere realizar esta funcionalidad y un *checkpoint* que tiene como fin garantizar tolerancia a fallos, como se muestra en el Cuadro 8 [8].

```
// Crear una consulta de streaming para escribir en el tópico Kafka
salidaFiltradoTopico = salidaDatasetTopico.writeStream().format("kafka")
.option("kafka.bootstrap.servers", "ip.kafka:9092")
.option("checkpointLocation", "/tmp/validacion/checkpoint")
.option("topic", "filtrado_topic")
.trigger(Trigger.ProcessingTime("1 seconds"))
.start();
```

Cuadro 8. Envío de Spark a Kafka.

Por último, para que la consulta de *streaming* siga funcionando se le debe de indicar que espere a una finalización (ver Cuadro 9).

```
// Espera de finalización
salidaFiltradoTopico.awaitTermination();
```

Cuadro 9. Espera de finalización de consulta streaming.

Sin embargo, se han necesitado hacer unos ajustes en el código para que sean eficiente en cuanto a tiempo de procesamiento. En primer lugar, aquellos datos que se recogen de la base de datos al utilizarse de manera constante cada vez que se realiza una iteración se introducen en caché, como se muestra en el Cuadro 10. Reduciendo notablemente el tiempo de espera entre estos datos [14].

```
// Carga de los códigos de los equipos desde PostgreSQL
Dataset<Row> equiposDB = reader.option("dbtable", equipo").load()
.select("id", "codigo", "eq_tipologia_id").cache();
```

Cuadro 10. Carga en caché de Dataset.

No obstante, se deben retirar de la caché una vez finalizada la iteración, en caso contrario se podría tener una saturación de ésta dando lugar a un bloqueo de la máquina (ver Cuadro 11).

```
// Eliminación de caché Datasets estáticos
equiposDB.unpersist();
```

Cuadro 11. Retirada de caché el Dataset.

También se debe tener en cuenta que, al haber una notoria diferencia entre tamaños de *Dataset* se producen *skew joins*, es decir, los datos que deben combinarse durante una operación de unión están distribuidos desproporcionadamente en los nodos que realizan esta unión. No obstante, Spark tiene una funcionalidad que está desactivada por defecto y permite la optimización de este tipo de *joins, mostrado en el Cuadro 12*.

```
// Configuración de Spark
SparkConf sparkConf = new SparkConf().setAppName("Validacion") .setMaster("local[*]")
.set("spark.sql.adaptive.enabled","true")
.set("spark.sql.adaptive.skewJoin.enabled", "true");

SparkSession spark = SparkSession.builder().config(sparkConf).getOrCreate();
```

Cuadro 12. Configuración skew joins.

Además, para el uso correcto en el clúster de Spark, cuando se tengan varios núcleos disponibles, se ha configurado la asignación de recursos de forma dinámica. De este modo para obtener los núcleos necesarios al realizar el procesamiento de los datos, donde se establece un mínimo y máximo para que no acapare una aplicación todos los núcleos disponibles. Esto se ha realizado en la configuración de Spark, al igual que la anterior mostrada en el Cuadro 13.

```
// Configuración de Spark
SparkConf sparkConf = new SparkConf().setAppName("Validacion")
.set("spark.dynamicAllocation.enabled", "true")
.set("spark.executor.cores", "1")
.set("spark.dynamicAllocation.minExecutors", "1")
.set("spark.dynamicAllocation.maxExecutors", "4");
```

Cuadro 13. Configuración de recursos.

Por otra parte, que para la obtención de las métricas de rendimiento en *streaming* es necesaria la configuración de cada aplicación. Esto, se introduce en la configuración de Spark de la siguiente manera en el Cuadro 14.

```
// Configuración de Spark
SparkConf sparkConf = new SparkConf().setAppName("Validacion")
.set("spark.sql.streaming.metricsEnabled","true");
```

Cuadro 14. Configuración de métricas de rendimiento.

En relación con el desarrollo con Spark, he encontrado varias limitaciones. Una de ellas ha sido la falta de una variable broadcast. Este tipo de variables se usan para transmitir información entre todos los ejecutores de Spark, haciendo que la carga de la información solo sea necesaria una única vez entre todos estos bloques. Sin embargo, en Spark Structured Streaming, se debe de leer de base de datos cada vez que se realiza una iteración del proceso. A pesar de ello, el rendimiento obtenido permite cumplir los requisitos del sistema para la cantidad de datos que se disponen actualmente.

Al comienzo del desarrollo de los jobs de Spark, se han hecho uso de varias funciones recogidas en Spark SQL que posteriormente no han sido válidas al implementar Structured Streaming. Entre estas se encuentra las funciones "*except*" o "*left outer join*", donde la documentación indica que no son funcionales debido un bajo rendimiento para el procesamiento en tiempo real [6].

Para solventar la falta de la funcionalidad "*except*" se ha hecho uso de la operación *join* "*left anti*" que permite la unión de dos Datasets donde se seleccionan las filas de la tabla izquierda que no tienen coincidencias en la tabla derecha (ver Cuadro 15).

```
// Obtención de las mediciones cuyo equipo no existe
erroresEquipo = df.join(equiposDB, df.col("equipo").equalTo(equiposDB.col("codigo")), "left_anti")
   .withColumn("error", functions.lit("El equipo no existe"));
```

Cuadro 15. Uso de función left anti join.

5.3.2. Recálculo y Agregación

Además, se ha desarrollado la aplicación de Recálculo que cumple con los requisitos establecidos en el apartado 2.3.2, enviando las mediciones procesadas al tópico "datos recalculados". Con el fin de cumplir uno de los requisitos establecidos para conseguir una granularidad más fina de las mediciones, se ha creado un proceso de Agregación en tiempo real para realizar el promedio de las mediciones por cada minuto, agrupadas por la magnitud, estación y equipo. En cuanto al procesamiento de las demás agregaciones mencionadas en el apartado 2.3.3, se ha optado por mantenerlos en *batch* para asegurar la eficiencia, sin que consuma recursos de manera innecesaria, para el cumplimiento de los objetivos del proyecto.

Para los bloques de Recálculo y Agregación se ha hecho uso de la versión anterior de Spark Structured Streaming, Spark Streaming. Esto es debido al problema que se encuentra en la primera versión mencionada donde no se pueden agrupar los datos por ventanas, mientras que en su versión anterior sí es posible.

El procesamiento de la entrada de los datos desde Kafka difiere a como se realiza en Spark Structured Streaming. En este se crea un flujo de entrada (*stream*) que recibe datos en crudo

desde Kafka. Se utiliza *createDirectStream* para configurar la conexión con Kafka y especificar el tópico al que se suscribirá para recibir datos, como se ve en el Cuadro 16.

```
// Stream que recibe los datos en crudo
JavaInputDStream<ConsumerRecord<String, String>> stream =
KafkaUtils.createDirectStream(ssc,LocationStrategies.PreferConsistent(),
ConsumerStrategies.<String, String>Subscribe(topics, kafkaParams));
```

Cuadro 16. Captura de datos en crudo en Spark Streaming.

Observando el Cuadro 17, después de recibir los datos de Kafka, se realiza una transformación para convertir los registros en formato JSON en objetos de la clase Medicion. Esto implica el proceso de deserialización de los datos en bruto en objetos estructurados que pueden ser más fáciles de manejar y analizar.

```
// Se transforman los datos a la clase medición
JavaDStream<Medicion> userEvents = stream.map(record -> {
        ObjectMapper mapper = new ObjectMapper();
        return mapper.readValue(record.value(), Medicion.class);
});
```

Cuadro 17. Transformación de datos en crudo a Medicion.

En Spark Streaming es intuitivo hacer uso de las ventanas donde se obtienen todas las mediciones enviadas en un periodo de tiempo determinado. El principal motivo del uso de estas ventanas ha sido que en el Recálculo es necesario tener agrupadas las mediciones a cada segundo, mientras que para la agregación se necesita agrupar los datos en función de la estación, equipo, magnitud y fecha cada 60 segundos, haciendo una medición minutal.

A continuación, se especifica la ventana que se desea obtener agrupados los datos para su posterior procesamiento. Además, con el fin de que la ventana de datos de la agregación minutal no tenga pérdidas en caso de que algún dato no llegue a tiempo se ha implementado un deslizamiento de ventana, asegurando la integridad de los datos, como se aprecia en el Cuadro 18.

```
// Lectura de mediciones
JavaDStream<Medicion> userEventsWindowed =
UserEvents.window(Durations.seconds(80), window.slide(20));
```

Cuadro 18. Ventana de tiempo con deslizamiento.

En este último paso (ver Cuadro 19), los datos en forma de JavaDStream se convierten en un Dataset<Row>. Esto es útil porque los Datasets en Spark ofrecen capacidades más avanzadas para el procesamiento y análisis de datos.

```
// Lectura de mediciones
userEventsWindowed.foreachRDD(rdd -> {
   if (!rdd.isEmpty()) {
      Dataset<Row> medicionesEntrantes = spark.createDataset(rdd.rdd(),
Encoders.bean(Medicion.class)).toDF()
   }
});
```

Cuadro 19. Conversión de RDD a Dataset.

Para la inserción de las mediciones desde Spark Streaming a Kafka su procesamiento es diferente a como se realiza en Structured Streaming, donde la forma más eficiente es hacer un bucle e ir enviando una a una estas mediciones, mostrado en el Cuadro 20.

```
// Configuración de Spark
JavaRDD<Row> medicionesRDD = mediciones.toJavaRDD();
medicionesRDD.foreachPartition(records -> {
// Generación del productor de Kafka
Producer<String, String> kafkaProducer = new KafkaProducer<>(kafkaParamsOut);

// Recorrer cada una de las mediciones
while (records.hasNext()) {
  Row medicion = records.next();
  // Convertir el objeto Medicion a JSON
  String json = medicion.json();
  // Generación del registro a enviar a Kafka
  ProducerRecord<String, String> record = new ProducerRecord<>("agregacion_topic", json);
  // Envío a Kafka
  kafkaProducer.send(record);
}
kafkaProducer.close();
});
```

Cuadro 20. Envío de datos desde Spark a Kafka.

Al igual que en Validación y Filtrado, en estos dos bloques se han aplicado las mismas funcionalidades para guardar temporalmente los datos de la base de datos en caché, así como la configuración para la obtención de métricas de *streaming* y la asignación dinámica de recursos.

Además, para la visualización del código realizado se puede acceder a este a través del Git [15]. Cabe destacar que con el fin de que se simplifique el despliegue de las aplicaciones, se han generado por cada proyecto (Validación, Filtrado, Recálculo y Agregación) un empaquetado Uber-Jar, es decir, un único archivo Jar que contiene el código de la aplicación y todas sus dependencias.

5.4. Monitorización del entorno

Para monitorizar las mediciones procesadas, los errores generados y las métricas de uso de recursos se ha utilizado Grafana. En esta se han creado varios *dashboards* donde se irán explicando cada uno de ellos.

El primer *dashboard* se ha realizado para mostrar las mediciones recibidas. Para ello, se han diseñado ventanas para monitorizar las mediciones procesadas en tiempo real obtenidas a través de Druid. Además de contar con gráficas para mostrar los datos generados en las agregaciones, obtenidas a través de PostgreSQL.

En la Figura 19 se puede observar en la ventana superior las medidas procesadas en tiempo real. Debido a las características de Druid, solamente ha sido posible realizar una tabla mostrando estas mediciones. En el resto de las ventanas se puede observar gráficamente las agregaciones realizadas por la aplicación, por estación, magnitud y equipo. Las mediciones minutales son procedentes de la aplicación Agregación de Spark, mientras que las restantes proceden de las vistas creadas en PostgreSQL. Además, se le proporciona al usuario final varios menús desplegables para que pueda seleccionar qué estación, magnitud o equipo desea ver en la pantalla.

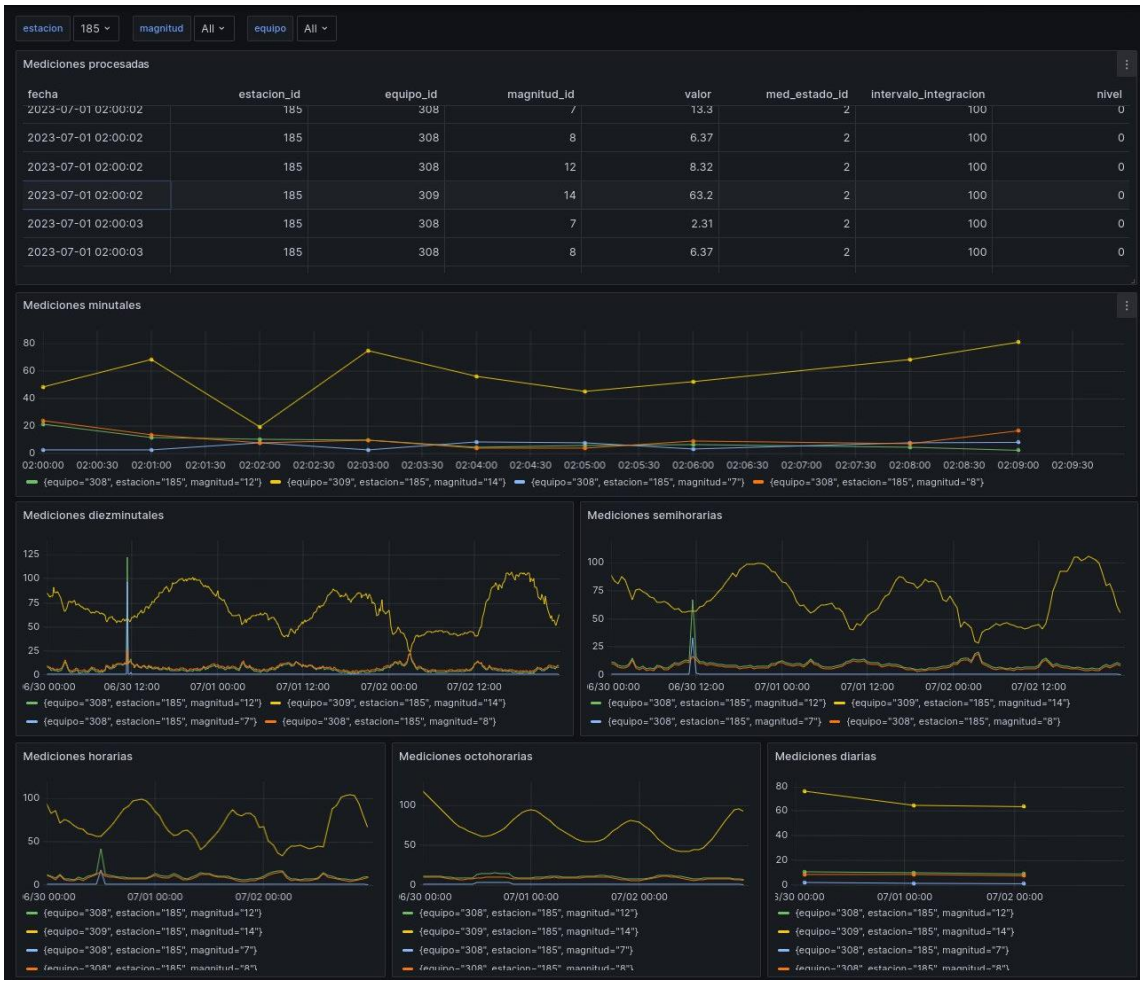
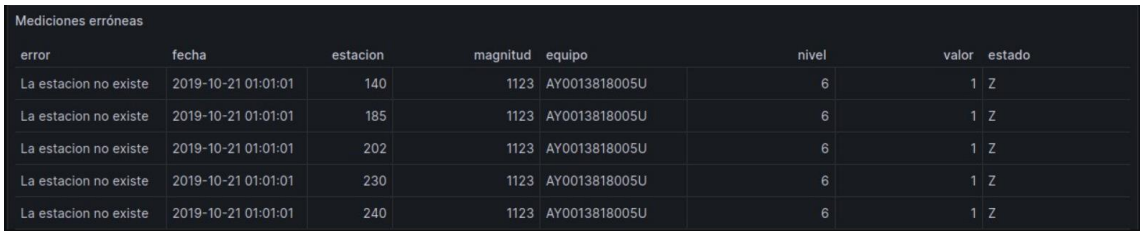


Figura 19. Gráfica de las mediciones procesadas en tiempo real.

En caso de que se generen errores también se dispone de Druid, que hace de intermediario entre Kafka y Grafana, permitiendo obtener los datos en tiempo real debido a su naturaleza de base de datos de alto rendimiento.

Dentro del *dashboard* se muestra una tabla donde se disponen de todos los errores generados, indicando en la primera columna el error que se produce, en el caso de la Figura 20, de validación de la estación.



Mediciones erróneas

error	fecha	estacion	magnitud	equipo	nivel	valor	estado
La estacion no existe	2019-10-21 01:01:01	140	1123	AY0013818005U	6	1	Z
La estacion no existe	2019-10-21 01:01:01	185	1123	AY0013818005U	6	1	Z
La estacion no existe	2019-10-21 01:01:01	202	1123	AY0013818005U	6	1	Z
La estacion no existe	2019-10-21 01:01:01	230	1123	AY0013818005U	6	1	Z
La estacion no existe	2019-10-21 01:01:01	240	1123	AY0013818005U	6	1	Z

Figura 20. Tabla datos erróneos.

Para la correcta configuración de Druid se han definido un fichero "*spec_procesado*" para recoger del tópico de "datos recalculados" para mostrar los datos procesados en tiempo real en Grafana y "*spec_errores*" para recoger del tópico las "datos erróneos", que se puede ver en el Git asociado a este proyecto [15], este realiza la conexión entre el tópico de Kafka de datos erróneos y la base de datos de Druid.

Grafana no ofrece de forma oficial un conector a Druid. Sin embargo, la comunidad ha realizado un conector para solucionar este problema, pudiendo mostrar los datos publicados en Kafka. [16]

También se ha configurado un *dashboard* para ver las métricas de rendimiento de Spark con el fin de comprender el uso de recursos dentro del proyecto. Para obtener estas métricas se ha hecho uso de la combinación de Prometheus y JMX Exporter, que es una de las pocas opciones que se encuentran para este propósito. Para ello se han configurado en Prometheus los diferentes puertos que abre JMX Exporter para exponer las métricas ofrecidas, como se puede ver en el fichero *"jmx_spark"* de Git [15].

Además, también se debe ejecutar a su vez el fichero de JMX Exporter que indica qué métricas se deben de recoger, indicando el puerto para que se muestre la información relevante de Spark Streaming, realizado como se indica en el Cuadro 21.

```
spark-submit --class "es.validacion.Main" --master local[1] --conf "spark.driver.extraJavaOptions=-
javaagent:/directorio_jmx_exporter/jmx_prometheus_javaagent-
0.19.0.jar=7080:/directorio_jmx_exporter/jmx_spark.yml"   /directorio_aplicaciones/Validacion.jar
```

Cuadro 21. Comando de ejecución de aplicación de Spark.

Esto solo tomará datos de cada aplicación, por lo que también es necesario establecer un *sink* de Prometheus, es decir, un componente que se utiliza para enviar y recopilar métricas, que está implementado de forma nativa en Spark, aunque se encuentra en estado experimental [17]. Esto se realiza en el fichero de configuración de *metrics.conf* de Spark como se muestra en el Cuadro 22.

```
*.sink.jmx.class=org.apache.spark.metrics.sink.JmxSink
master.source.jvm.class=org.apache.spark.metrics.source.JvmSource
worker.source.jvm.class=org.apache.spark.metrics.source.JvmSource
driver.source.jvm.class=org.apache.spark.metrics.source.JvmSource
executor.source.jvm.class=org.apache.spark.metrics.source.JvmSource

*.sink.prometheusServlet.class=org.apache.spark.metrics.sink.PrometheusServlet
*.sink.prometheusServlet.path=/metrics/prometheus

master.sink.prometheusServlet.path=/metrics/master/prometheus
applications.sink.prometheusServlet.path=/metrics/applications/prometheus
```

Cuadro 22. Configuración sink de Prometheus para Spark.

Una vez realizado esto, se pueden obtener las ventanas que se muestran en la Figura 21**,** que permiten hacerse una idea del rendimiento que tiene cada una de las aplicaciones en Spark. Se encuentran 4 ventanas que muestran información relevante a la hora de analizar las aplicaciones, donde son obtenidas a cada segundo.

El driver en Spark es un componente esencial encargado de iniciar y coordinar las distintas aplicaciones de Spark, así como la asignación de recursos a cada una de estas. Empezando por la parte superior izquierda, se tiene el uso de la memoria del driver en tanto por ciento. Para ello se ha utilizado la métrica spark_driver_jvm_heap_usage, siendo 0 el valor más bajo y 100 la carga máxima de memoria soportada.

A la derecha de este se encuentra la memoria utilizada en bits de cada una de las aplicaciones, indicando cual es el máximo de cada nodo trabajador. En el caso de que se ejecute en un solo nodo el trabajador será el mismo, mientras que en el clúster puede llegar a haber varios nodos, donde la memoria utilizada será una media del conjunto de nodos utilizados por cada

aplicación. Para lograrlo se ha llamado a las métricas metrics_jvm_heap_max_Value y spark_executor_JVMHeapMemory.

En la línea inferior empezando por la izquierda, se muestra el tiempo de procesamiento de cada ejecución, expresado en segundos por cada batch. Se han utilizado las métricas spark_driver_structured_streaming_latency para Structured Streaming y spark_driver_streaming_lastCompletedBatch_processingDelay para Streaming.

En la última ventana se encuentran las mediciones que llegan a cada aplicación por segundo, haciendo uso de las métricas spark_driver_structured_streaming_inputRate_total para Structured Streaming y spark_driver_streaming_lastReceivedBatch_records para Streaming.

Se debe tener en cuenta que, en las últimas dos ventanas, para las aplicaciones que utilizan Structured Streaming, aparecen varias líneas por cada una de ellas. Estas representan cada *query* que se ejecuta dentro de la aplicación, es decir, en el caso de la Validación se tienen dos *queries*, que registran las mediciones en diferentes tópicos. Asimismo, ocurre con Filtrado que tiene dos *queries* para guardar los datos en distintos tópicos. Con el fin de que las métricas sean más aclaratorias se ha optado por dejar una línea por aplicación para que no den lugar a confusión.

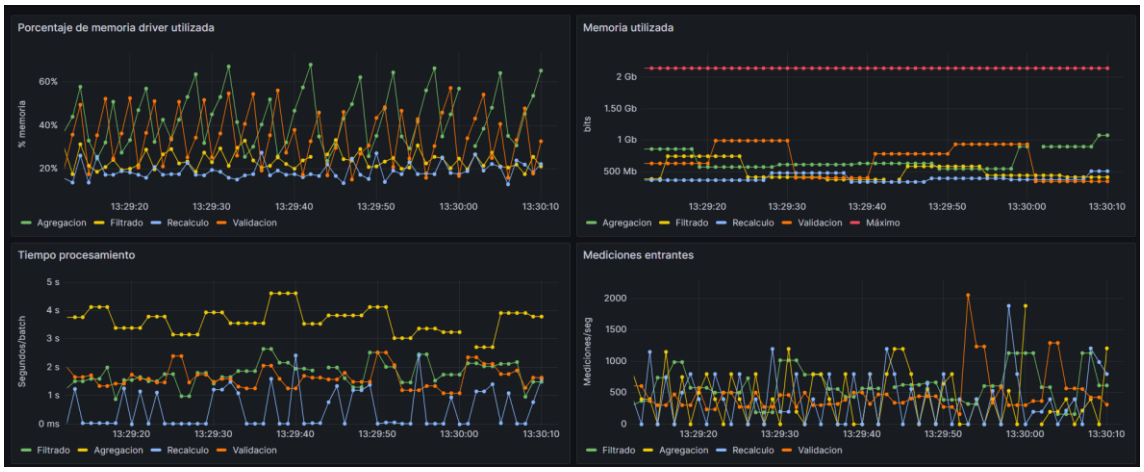


Figura 21. Métricas de rendimiento de Spark.

Sin embargo, aunque se hayan establecido una serie de parámetros que pueden ser valiosos, no se ha conseguido extraer el porcentaje de uso de la CPU en Spark. Además, en la documentación de JMX Exporter indica que este no puede obtener estas métricas [18]. A pesar de haber navegado por los foros intentando encontrar una solución a esto, no se ha conseguido obtener ningún resultado, dando por hecho que es un error de diseño no mostrar cuanta CPU se consume por cada aplicación.

Para la monitorización del rendimiento de Kafka se ha utilizado JMX Exporter al igual que en Spark, donde se ha configurado un fichero (jmx_kafka) para obtener estas métricas de rendimiento. Cuando se lanza el servidor de Kafka se debe insertar una variable que indique que se deben recoger mediciones, asignando el agente de JMX que irá recogiendo las mediciones, el puerto donde se van a recoger y el fichero de las métricas de rendimiento que se desean, como se indica en el Cuadro 23.

```
KAFKA_OPTS="-javaagent:/directorio_jmx_exporter/jmx_prometheus_javaagent-0.19.0.jar=6072:
/directorio_jmx_exporter/jmx_kafka.yml" bin/kafka-server-start.sh config/server.properties
```

Cuadro 23. Comando para el servidor de Kafka.

Por tanto, para mostrar estas métricas en Grafana se ha hecho de nuevo uso Prometheus, donde se han expuesto a través de un *dashboard* configurado por la comunidad. En este se indican las mediciones entrantes, así como los bytes de entrada y salida por cada tópico de la aplicación, el uso de la CPU y la memoria utilizada por la máquina virtual en bytes (ver Figura 22).

Figura 22. Métricas de rendimiento de Kafka.

Además, se han obtenido varias métricas de rendimiento de PostgreSQL, con el fin de conocer posibles bloqueos que puedan ocurrir al realizar consultas e inserciones contra la base de datos. Para ello se ha hecho uso una vez más de JMX Exporter, donde se ha configurado junto con la base de datos el agente como se indica en el GitHub de Postgres-Exporter [19]. En la Figura 23 se puede ver el tiempo de uso de la CPU, la memoria utilizada en bytes y el número de *deadlocks*, es decir, dos procesos que se bloquean porque esperan a que el otro termine para continuar su trabajo.

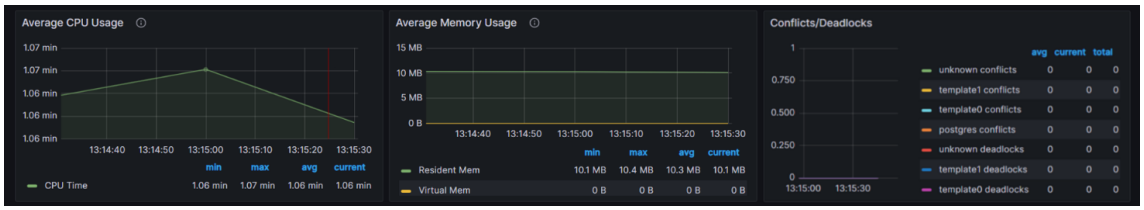


Figura 23. Métricas de rendimiento de PostgreSQL.

6. Pruebas de rendimiento

Una vez hecho el desarrollo y probado el funcionamiento de cada aplicativo independientemente en local, se realizó el despliegue en Google Cloud Engine y se procedió a realizar las pruebas de rendimiento. Para este análisis se utilizaron las métricas de recursos de Spark que muestran información relevante de *streaming,* de Kafka para analizar el rendimiento de la tecnología y de PostgreSQL con el fin de comprender las posibles latencias que surgen al consultar la base de datos.

Para generar los flujos de datos necesarios para las pruebas, se creó un conjunto de datos sintético, dado que el conjunto de datos original contiene únicamente mediciones en

intervalos de diez minutos. Para llevar a cabo esta tarea, se optó por generar nuevas mediciones a partir de las mediciones de diez minutos que se encuentran en la base de datos.

Inicialmente, se crearon mediciones sintéticas para cada minuto, coincidiendo con los valores originales al realizar la agregación de diez minutos. Posteriormente, se procedió a generar mediciones por segundo a partir de cada minuto. Este proceso se llevó a cabo con el propósito de verificar que los resultados finales concuerdan con los datos actualmente disponibles en la aplicación, y así validar que los procedimientos se están ejecutando de manera correcta.

Para las pruebas se han implementado varias configuraciones para comprobar el rendimiento del proyecto que se irán viendo a continuación cada uno de ellos:

- Despliegue en un nodo.
- Despliegue en varios nodos.
- Simulación de escalado del clúster.
- Simulación de caída de un nodo trabajador del clúster.

6.1. Despliegue en un nodo

Las pruebas en un solo nodo no las pude hacer en mi máquina local por falta de recursos. Para solventarlo hice un despliegue en una máquina en la nube de Google con 4 núcleos, pero llegaba a un punto de saturación que lo hacía inviable. Por este motivo fue necesario el aumento de 2 núcleos más a la máquina donde se han realizado las pruebas, con la configuración de la Tabla 1.

	Tipo de CPU	Memoria RAM	Almacenamiento	Sistema Operativo
N2-standard (custom)	6 CPU virtuales (3 núcleos)	16 GB	30 GB	Ubuntu 20 LTS

Tabla 1. Máquina con 6 núcleos.

Se comenzó analizando el rendimiento con la entrada de 400 mediciones por segundo, para conocer cuál sería el rendimiento con la carga actual de la aplicación de CIC. Atendiendo a la Figura 24, se puede observar en la gráfica superior izquierda que el uso del planificador del driver no tiene una carga de la memoria alta, haciendo que no llegue a la saturación. Respecto a la ventana superior derecha se ha asignado 2 Gb a cada aplicación lanzada, no obstante, se puede ver que no supera el uso de 1 Gb de memoria, dando por hecho que no se tiene para esta carga un problema de memoria. Viendo la gráfica inferior izquierda, el tiempo de procesamiento de los procesos de Filtrado, Validación y Recálculo rondan entre el segundo y los 2 segundos, aunque hay ocasiones que se tienen picos que superan el límite marcado. Mientras que Agregación tiene unos tiempos más altos frente al resto de las aplicaciones ya que debe hacer una inserción de las mediciones en la base de datos.

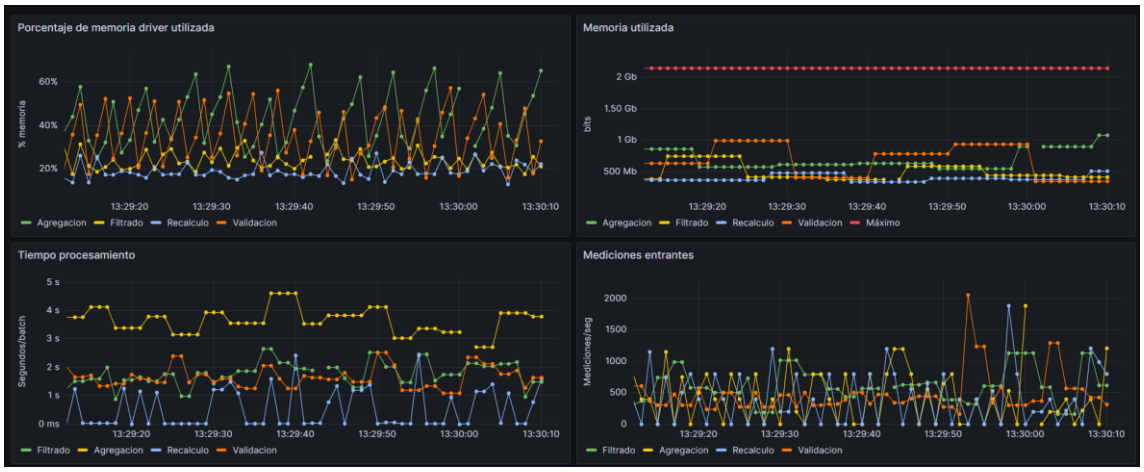


Figura 24. Rendimiento en un nodo con 400 mediciones/segundo.

Los resultados obtenidos en cuanto a la precisión de los datos han sido acordes a lo esperado, donde se han procesado correctamente las mediciones haciendo que las agregaciones minutales generadas coincidan con los datos originales.

Analizando Kafka que se puede observar en la Figura 25 que con el número de entrada de las mediciones por segundo no se tiene una saturación de este, ya que el porcentaje de uso de la CPU es sumamente bajo, sin llegar a utilizar el núcleo completo. Obteniendo el número de las mediciones correctas por cada tópico.

Figura 25. Rendimiento de Kafka con 400 mediciones/segundo.

A pesar de que las métricas de base de datos de PostgreSQL de la Figura 26 indiquen que los tiempos son favorables respecto al uso de la CPU y sin ninguna generación de *deadlock*. Se debe destacar que cuando se han realizado pruebas sin consultar a base de datos el tiempo ha disminuido por debajo del segundo en todas las aplicaciones. Por lo que se puede sacar como conclusión de que hay una latencia intrínseca en la obtención e inserción de mediciones en la base de datos.

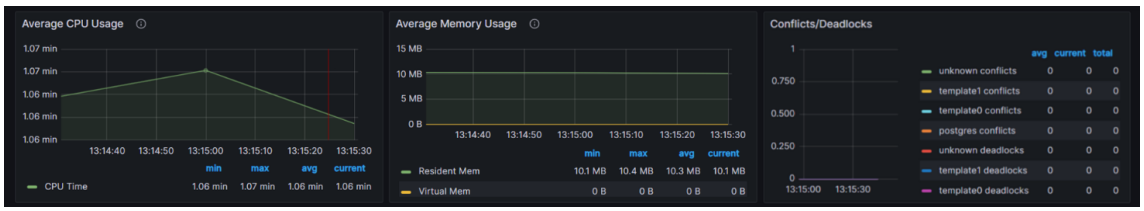


Figura 26. Rendimiento de PostgreSQL con 400 mediciones/segundo.

Posteriormente se incrementó el número de mediciones entrantes por segundo para comprobar los límites que se pueden realizar con la solución planteada. La siguiente prueba fue con 1000 mediciones por segundo, con unos resultados muy similares. A pesar de ello se aprecia en la Figura 27 que el uso de la memoria ha aumentado superando la barrera de 1 Gb, pero sin llegar a saturar el límite establecido.

Figura 27. Rendimiento en un nodo con 1000 mediciones/segundo.

Aumentando el número de entradas hasta 2000 mediciones por segundo el uso de la memoria es muy similar al anterior caso, pero se observa en la Figura 28 que la gráfica de los tiempos de procesamiento Filtrado supera el límite de los 2 segundos de forma constante. Esto puede indicar una falta de recursos en cuanto a núcleos para procesar tal cantidad de tuplas por segundo.

Figura 28. Rendimiento en un nodo con 2000 mediciones/segundo.

Con el fin de comprobar el punto de saturación de esta solución se ha seguido aumentando la entrada hasta 4000 por segundo. En los resultados se puede apreciar en la Figura 29 que el rendimiento empeora notablemente, donde Filtrado comienza a aumentar en gran medida su tiempo. Cabe destacar que no es por problema de memoria viendo las gráficas superiores.

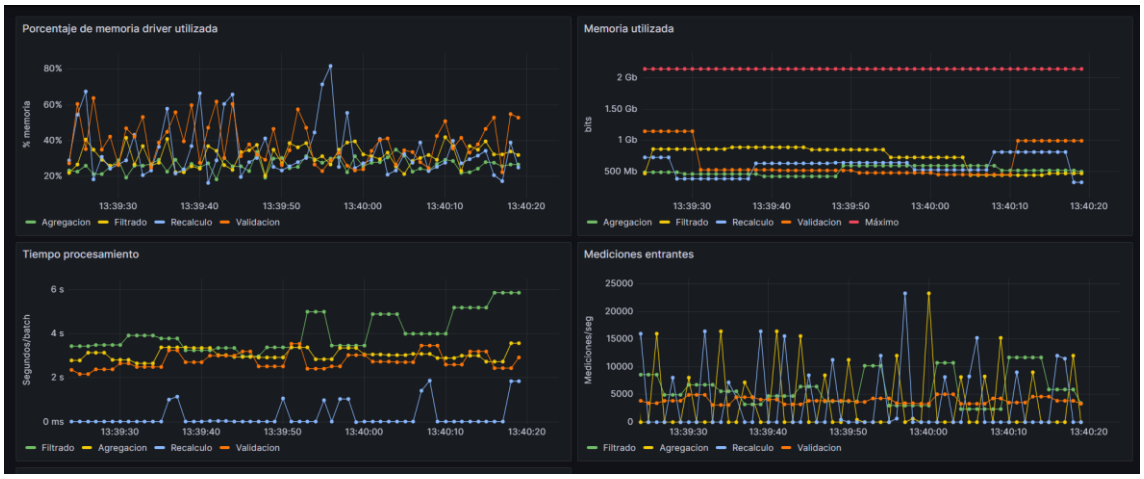


Figura 29. Rendimiento en un nodo con 4000 mediciones/segundo.

Atendiendo a las métricas de rendimiento de Kafka y PostgreSQL tampoco se han encontrado indicios de una pérdida de rendimiento, donde se puede apreciar en la Figura 30 que Kafka sigue sin utilizar un núcleo al completo y no tiene picos de memoria mientras procesa 4000 entradas por cada tópico.

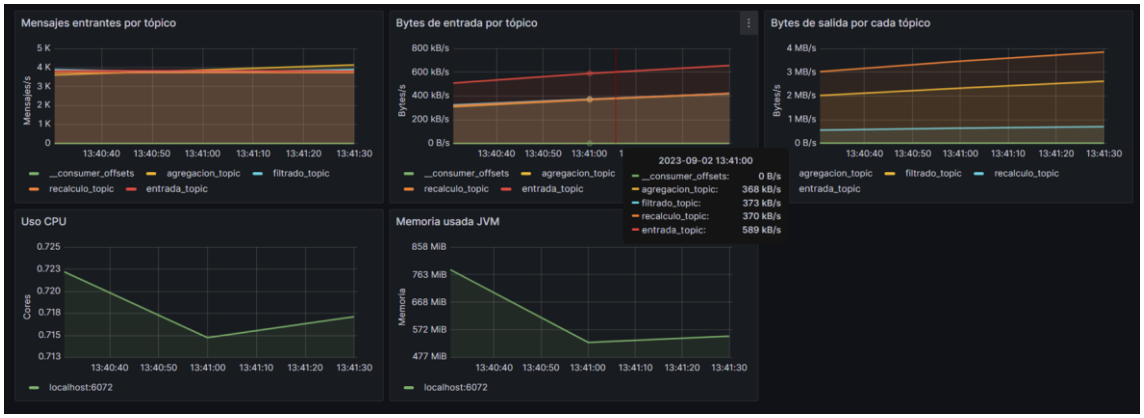


Figura 30. Rendimiento de Kafka con 4000 mediciones/segundo.

En la figura 31 la base de datos de PostgreSQL mantiene unos valores estables y sin generar ningún deadlock dentro de este. Por lo que indica que Spark necesita una mayor cantidad de núcleos para poder actuar frente a tal cantidad de mediciones por segundo.

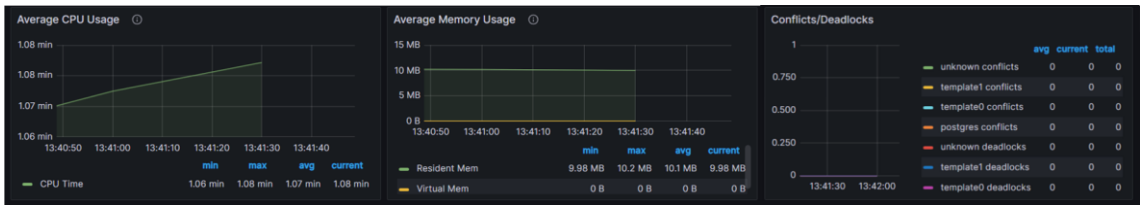


Figura 31. Rendimiento de PostgreSQL con 4000 mediciones/segundo.

Por ello, se probó una configuración donde se le asignó a la misma máquina más núcleos y memoria, teniendo los recursos de la Tabla 2. Cabe destacar que se les asignó a los procesos de Validación y Filtrado un núcleo más a cada una por la falta de rendimiento vista en la última prueba.

	Tipo de CPU	Memoria RAM	Almacenamiento	Sistema Operativo
N2-standard-8	8 CPU virtuales (4 núcleos)	32 GB	30 GB	Ubuntu 20 LTS

Tabla 2. Máquina con 8 núcleos.

Se realizó una prueba con 4000 mediciones por segundo y en los resultados obtenidos en la Figura 32 se puede observar que se han conseguido unos mejores tiempos frente a lo demostrado con 6 núcleos. Las métricas de la base de datos y de Kafka fueron las mismas, mientras que el tiempo de procesamiento de Spark disminuyó a pesar de que Filtrado en varias ocasiones supere el límite de los 2 segundos por aplicación. Esto indica que, pese a que con 6 núcleos puede ser suficiente para el procesamiento de 2000 mediciones, si se le asignan más núcleos a cada aplicación de Spark se pueden conseguir unos mejores tiempos de procesamiento, como es de esperar.

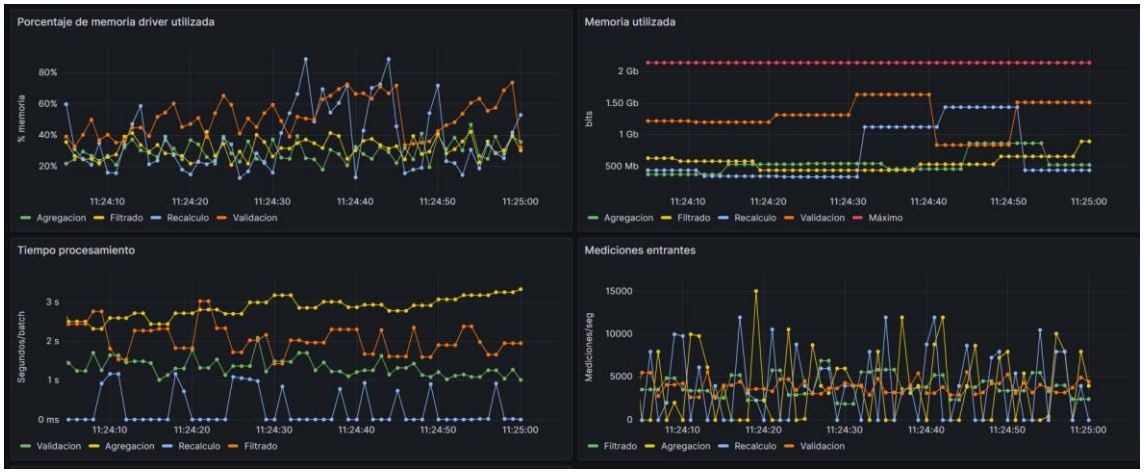


Figura 32. Rendimiento en nodo mejorado con 4000 mediciones/segundo.

6.2. Despliegue en varios nodos

Con el fin de sacar el mayor partido a Spark se configuró una serie de máquinas para hacer un clúster escalable. Sin embargo, se tuvieron varias dificultades, debido a que en el periodo de prueba que otorga Google, se establecen unas cuotas que limitan el uso de recursos. En caso de que se realicen máquinas virtuales en la misma región se limitan hasta 8 núcleos como máximo por tipo de máquina y 12 núcleos en todas las regiones disponibles como se puede ver en la Figura 33.

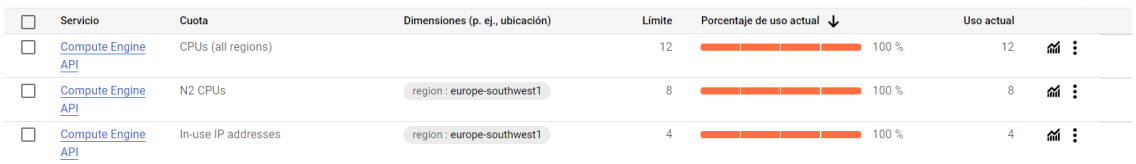


Figura 33. Cuotas de Google Cloud Project.

Con el fin de no pasar las cuotas indicadas anteriormente se mantuvo una máquina con 6 núcleos con la configuración de la Tabla 1, que incluye todas las tecnologías utilizadas en un solo nodo y actuaría como nodo maestro en Spark. Para los nodos trabajadores del clúster de Spark se han creado máquinas con recursos de la Tabla 3 con el fin de llegar al límite de 12 núcleos totales.

	Tipo de CPU	Memoria RAM	Almacenamiento	Sistema Operativo
T2D-standard-2	2 CPU virtuales (1 núcleo)	8 GB	10 GB	Ubuntu 20 LTS

Tabla 3. Máquina trabajadora Spark con 2 núcleos.

Se debe tener en cuenta que las aplicaciones de Spark en modo clúster tienen un comportamiento diferente que cuando se lanzan en modo cliente. Estas al lanzarse en el clúster se les asignan unos recursos que luego no son compartidos entre las diferentes aplicaciones en caso de que se estén utilizando, frente al modo cliente que se pueden lanzar varias aplicaciones sobre un mismo núcleo [20].

Para el despliegue correcto del clúster debe haber un nodo maestro, encargado de la coordinación y administración de éste. Para ello se debe ejecutar el comando mostrado en el Cuadro 24. En esta se puede ver la opción "-h", que indica la dirección IP donde el nodo maestro estará disponible y la "0.0.0.0" para que este escuche a todas las direcciones IP disponibles en la máquina.

```
./start-master.sh -h 0.0.0.0
```

Cuadro 24. Comando para iniciar máster de Spark.

Por otra parte, al lanzar los nodos trabajadores en el clúster se debe hacer como se muestra a continuación, donde se debe de indicar la dirección IP interna del nodo maestro (ver Cuadro 25).

```
./start-worker.sh spark://ip.interna.master:7077
```

Cuadro 25. Comando para inciar trabajador de Spark.

Para el envío de las mediciones hacia Kafka se ha abierto el puerto 9092 en la máquina virtual para que se puedan realizar conexiones externas y enviar datos a los tópicos. Para ello se debe modificar el fichero server.properties de Kafka donde se le indica cuál es su IP externa como se muestra en el Cuadro 26.

```
advertised.listeners=PLAINTEXT://ip.externa:9092
```

Cuadro 26. Configuración para recibir conexiones externas a Kafka.

A la vista de los resultados anteriores, se espera que el rendimiento en el clúster de Spark sea mejor. Sin embargo, se debe tener en cuenta que por limitaciones del número de máquinas la configuración realizada es un nodo maestro con todas las tecnologías de la Tabla 1 y 2 nodos trabajadores con la configuración de la Tabla 3, obteniendo que cada aplicación tendrá asignado 1 núcleo y 2 Gb de memoria.

En la primera prueba se analizó la ingesta de 400 mediciones por segundo (Figura 34) y los resultados son muy similares a los obtenidos anteriormente en un nodo. Además, las métricas de Kafka y PostgreSQL son muy similares debido a que se encuentran en la misma máquina que cuando se realizaron las pruebas en un nodo.

Figura 34. Rendimiento en varios nodos con 400 mediciones/segundo.

Se siguió incrementando el número de mediciones entrantes por segundo hasta llegar al punto de ruptura del caso en un solo nodo, 4000 medidas por segundo. Se observa en la Figura 35 que el rendimiento se sigue manteniendo estable, aunque ha incrementado para el caso de la aplicación de Filtrado. Además, se aprecia un incremento del consumo de la memoria, esto puede indicar que sea necesario aumentar el tamaño de esta para mejorar su capacidad de procesamiento de tuplas.

Figura 35. Rendimiento en varios nodos con 4000 mediciones/segundo.

Sin embargo, se siguieron aumentando las mediciones hasta llegar a las 8000 por segundo (Figura 36). En este caso se observó que el tiempo de procesamiento de la aplicación de Filtrado aumenta superando en varias ocasiones los dos segundos, pero no se tiene un problema en cuanto a consumo de memoria en las aplicaciones.

Figura 36. Rendimiento en varios nodos con 8000 mediciones/segundo.

6.3. Escalado en caliente

Para la realización de este apartado se creó un nuevo nodo con las características de la Tabla 3. Es importante considerar que el tipo de clúster Spark en modo "standalone" carece de una capacidad intrínseca para el escalado en tiempo real, gestionando los recursos y estableciendo límites de manera automática para la expansión dinámica del clúster. En este caso se hizo uso de una configuración que permite asignar un mínimo y máximo de ejecutores a cada aplicación para que no tomen todos los recursos del clúster. Además, se le puede asignar un número de núcleos iniciales para comenzar la aplicación, y cada vez que se requiera más potencia para el procesamiento lo tomará del clúster.

Por tanto, se añadió un nuevo nodo al clúster otorgando a la aplicación de Validación y Filtrado 1 núcleo más a cada una. Se puede ver en la Figura 37 que los resultados son positivos en cuanto al escalado, ya que se ha conseguido reducir el tiempo del procesado de los datos. Por lo que, si es necesario hacer el procesamiento de más mediciones, añadiendo más máquinas al clúster podría solventar el problema de rendimiento.

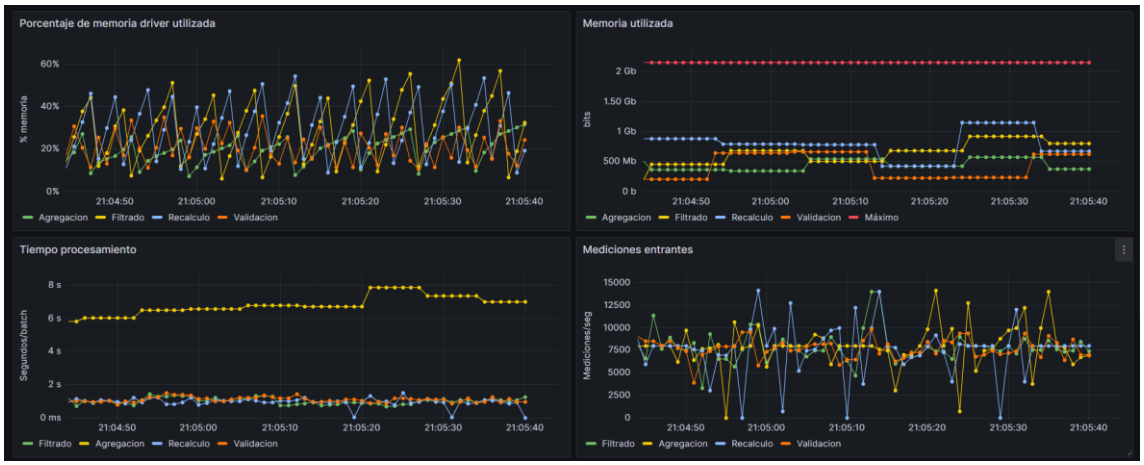


Figura 37. Rendimiento en varios nodos escalada con 8000 mediciones/segundo.

Para comprobar el rendimiento de la aplicación escalada se siguió aumentando el número de mediciones entrantes hasta 20000 por segundo, donde los resultados se mantuvieron estables para las aplicaciones de Validación, Filtrado y Recálculo. En cambio, la aplicación de Agregación al tener que realizar inserciones en base de datos ha seguido incrementando su tiempo. Se puede apreciar en la Figura 38, que los tiempos de las tres primeras aplicaciones mencionadas se mantiene por debajo de los 2 segundos, mientras que la Agregación cada vez aumenta más su tiempo de procesamiento. Además, se observa un aumento del uso de la memoria, sin ser determinante para el aumento del tamaño de este.

Figura 38. Rendimiento en varios nodos escalada con 20000 mediciones/segundo.

En la última prueba realizada, mostrada en la Figura 39, se vio que para la inserción de 24000 mediciones por segundo comienzan a empeorar el rendimiento de las aplicaciones en su conjunto. Sin embargo, si se observa a la utilización de la memoria sigue sin llegar a saturarse, por lo que si se introdujera una mayor cantidad de trabajadores con núcleos para las aplicaciones se podría seguir procesando estas consiguiendo los tiempos establecidos en los requisitos.

Figura 39. Rendimiento en varios nodos escalada con 24000 mediciones/segundo.

Cabe destacar que para que se haga escalado en caliente de forma automática se podría utilizar Kubernetes o Google Dataproc, ambas opciones son viables para una autogestión del clúster. Mientras que en el modo standalone, Yarn y Mesos no se encuentra una posibilidad de gestión de máquinas para su creación y posterior inserción dentro del clúster.

6.4. Simulación caída de nodo

Para esta prueba se ha hecho uso de Spark UI, esta aplicación muestra para cada proceso cuántos núcleos tiene asignados, así como la memoria por cada ejecutor y el estado en el que se encuentra. En caso de que un nodo falle y la aplicación donde se alojaba tiene más nodos, aquella información que se estuviera procesando en el momento de la caída se moverá a otro trabajador y se seguirá realizando el procesamiento [21].

En caso de caída completa del nodo se puede ver que este proceso se para, pero no se detiene al completo, es decir, se queda pendiente de obtener recursos del clúster. Al igual que ocurre cuando se insertan más aplicaciones al clúster sin tener recursos suficientes, se quedarán en un estado de espera (*waiting*) hasta que se inserten más recursos u otro nodo libere sus recursos y queden disponibles, como se puede ver en la Figura 40.

▾ Running Applications (4)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20230823183531-0017 (kill)	Agregacion	0	1024.0 MiB		2023/08/23 18:35:31	tfmkafkaspark	WAITING	0.3 s
app-20230823183519-0016 (kill)	Recalculo	0	1024.0 MiB		2023/08/23 18:35:19	tfmkafkaspark	WAITING	12 s
app-20230823183415-0015 (kill)	Filtrado	0	1024.0 MiB		2023/08/23 18:34:15	tfmkafkaspark	WAITING	1.3 min
app-20230823182533-0014 (kill)	Validacion	1	1024.0 MiB		2023/08/23 18:25:33	tfmkafkaspark	RUNNING	10.0 min

Figura 40. Aplicaciones con los núcleos asignados en el clúster de Spark.

7. Conclusiones y líneas futuras

En el presente trabajo se ha realizado una implementación de las tecnologías *big data* con el objetivo principal de mejorar el rendimiento, la escalabilidad y tolerancia a fallos del proyecto original, consiguiendo los requisitos planteados. Entre estos logros alcanzados están el aumento del número de las fuentes de datos, la tasa de ingesta de datos, la monitorización de los datos en tiempo real, además de la visualización de métricas de rendimiento de la solución, permitiendo a las operadoras configurar la solución en función de las necesidades de cada momento.

En el plano personal, puedo decir que, han sido varias las dificultades encontradas en la puesta en marcha de todas las herramientas y en la resolución de los problemas que iban surgiendo, esto me ha permitido forjarme en la resolución de problemas y en la búsqueda de soluciones alternativas. Por otra parte, las tecnologías usadas son hoy en día de gran utilidad y demandadas en el mundo profesional por lo que este TFM ha contribuido a que tenga una formación más sólida y especializada en la construcción de aplicaciones de uso intenso de datos.

En cuanto a las líneas de trabajo futuro se encuentran la configuración del clúster a través de gestores de recursos como Kubernetes o Google Dataproc, tecnología hoy muy implantada. Por otra parte, como trabajo futuro, quedaría pendiente resolver la ingesta directa de las fuentes de datos, esto es, sustituir el actual serializador por sus correspondientes publicadores para cada sensor de las distintas fuentes de datos.

8. Bibliografía

[1] «RabbitMQ frente a Kafka: diferencia entre los sistemas de colas de mensajes - AWS». https://aws.amazon.com/es/compare/the-difference-between-rabbitmq-and-kafka/ (accedido 3 de agosto de 2023).

[2] G. Shapira, *Kafka : the definitive guide : real-time data and stream processing at scale*. O'Relly, 2021.

[3] «Apache Kafka: Powered by». https://kafka.apache.org/powered-by (accedido 25 de julio de 2023).

[4] «A side-by-side comparison of Apache Spark and Apache Flink for common streaming use cases | AWS Big Data Blog». https://aws.amazon.com/es/blogs/big-data/a-side-by-side-comparison-of-apache-spark-and-apache-flink-for-common-streaming-use-cases/ (accedido 3 de agosto de 2023).

[5] «¿Qué es Apache Spark? | Google Cloud». https://cloud.google.com/learn/what-is-apache-spark?hl=es (accedido 2 de mayo de 2023).

[6] «Structured Streaming Programming Guide - Spark 3.4.0 Documentation». https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html (accedido 2 de mayo de 2023).

[7] «Spark SQL and DataFrames - Spark 3.4.0 Documentation». https://spark.apache.org/docs/latest/sql-programming-guide.html (accedido 2 de mayo de 2023).

[8] «Structured Streaming Programming Guide - Spark 3.4.1 Documentation». https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#recovering-from-failures-with-checkpointing (accedido 5 de septiembre de 2023).

[9] «PostgreSQL: About». https://www.postgresql.org/about/ (accedido 2 de mayo de 2023).

[10] «Introduction to Apache Druid · Apache Druid». https://druid.apache.org/docs/latest/design/ (accedido 3 de agosto de 2023).

[11] «Technology | Apache® Druid». https://druid.apache.org/technology/ (accedido 6 de septiembre de 2023).

[12] «Grafana | Query, visualize, alerting observability platform». https://grafana.com/grafana/ (accedido 3 de agosto de 2023).

[13] «How Can We Measure Our Software's Modularity and Dependencies? | by Héla Ben Khalfallah | Better Programming». https://betterprogramming.pub/inside-software-modularity-and-related-metrics-2e5af2b447dc (accedido 25 de julio de 2023).

[14] «Performance Tuning - Spark 3.4.1 Documentation». https://spark.apache.org/docs/latest/sql-performance-tuning.html#caching-data-in-memory (accedido 6 de agosto de 2023).

[15] «David Gragera Iglesias / TFM_SIVCA · GitLab». https://gitlab.com/dgi646/tfm_svca (accedido 13 de septiembre de 2023).

[16] «Druid plugin for Grafana | Grafana Labs». https://grafana.com/grafana/plugins/grafadruid-druid-datasource/ (accedido 23 de agosto de 2023).

[17] «Monitoring and Instrumentation - Spark 3.3.1 Documentation». https://spark.apache.org/docs/3.3.1/monitoring.html (accedido 12 de agosto de 2023).

[18] «GitHub - prometheus/jmx_exporter: A process for exposing JMX Beans via HTTP for Prometheus consumption». https://github.com/prometheus/jmx_exporter (accedido 20 de agosto de 2023).

[19] «GitHub - prometheus-community/postgres_exporter: A PostgreSQL metric exporter for Prometheus». https://github.com/prometheus-community/postgres_exporter (accedido 20 de agosto de 2023).

[20] «Job Scheduling - Spark 3.4.1 Documentation». https://spark.apache.org/docs/latest/job-scheduling.html (accedido 3 de agosto de 2023).

[21] «Spark Standalone Mode - Spark 3.4.1 Documentation». https://spark.apache.org/docs/latest/spark-standalone.html#high-availability (accedido 23 de agosto de 2023).