Arquitecturas de Streaming para Analítica de Salud: Un Estudio Comparativo de los Enfoques Kappa y Delta

Entregado como requisito para la obtención del titulo de Master en Big Data

Emiliano Conti - 289917

Tutor: Alejandro Bianchi

Universidad ORT

13 de abril de 2025

Disclaimer

Yo, Emiliano Conti declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto Final del Master en Big Data;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Firma: _____

Fecha: _____

Abstract

En el contexto del monitoreo remoto de pacientes, el procesamiento eficiente y confiable de datos de sensores en tiempo real se vuelve un requisito fundamental para la prevención y mejora de la atención médica.

Este trabajo analiza y compara dos arquitecturas modernas de procesamiento de datos en streaming: *Kappa* y *Delta*, con el objetivo de determinar cuál resulta más adecuada para este tipo de sistemas.

Se desarrolló un sistema de monitoreo basado en sensores que simula un entorno hospitalario y utiliza puntuaciones derivadas del sistema *NEWS2* para evaluar el estado de salud de los pacientes. Sobre este entorno se implementaron ambas arquitecturas utilizando un stack tecnológico compuesto por herramientas como Apache Kafka, Apache Flink y Apache Doris, entre otras.

La comparación se realizó en función de métricas técnicas como latencia, throughput, uso de recursos y tolerancia a fallos, así como también aspectos operativos y de costo.

Los resultados muestran que la arquitectura *Kappa* ofrece menores tiempos de latencia, mientras que la arquitectura *Delta* destaca por su alto throughtput, escalabilidad, gestión operativa y flexibilidad para reprocesamiento histórico y mejor integración con almacenamiento analítico.

Finalmente, se discuten los desafíos asociados a la implementación de estas arquitecturas en entornos de salud, y se presentan recomendaciones para su adopción según las necesidades específicas del sistema.

Índice general

Capítulo 1

Introducción

1.1. Descripción del Proyecto

Este proyecto compara las arquitecturas Kappa y Delta en el contexto del monitoreo remoto de pacientes mediante sensores IoT. En la era de la salud digital, estos sistemas generan grandes volúmenes de datos en tiempo real que requieren procesamiento eficiente. Se analizarán ambas arquitecturas, se definirán métricas de comparación y se implementarán en un caso de uso de monitoreo de pacientes. El objetivo es determinar la arquitectura más adecuada, considerando factores como latencia, escalabilidad, complejidad de implementación y manejo de datos históricos y en tiempo real.

1.2. Objetivos

1. Realizar un análisis teórico exhaustivo de las arquitecturas Kappa y Delta, detallando sus componentes y flujos de datos.

2. Definir un conjunto de métricas y criterios para la comparación objetiva de ambas arquitecturas en el contexto del monitoreo remoto de pacientes.

3. Implementar ambas arquitecturas utilizando un conjunto de datos simulado de sensores de monitoreo de pacientes.

4. Ejecutar pruebas de rendimiento y funcionalidad en ambas implementaciones.

5. Analizar los resultados obtenidos y determinar la arquitectura más adecuada para el caso de uso específico de monitoreo remoto de pacientes.

1.3. Caso de Estudio: Big Data en Sistema de Salud

1.3.1. Contexto del Sistema

Sistema de salud integral que incluye perfiles de pacientes, telemedicina e integración con dispositivos IoT. El objetivo es mejorar la atención médica, con foco en prevencion, utilizando tecnologia.

1.3.2. Descripción del Caso de Uso

Monitoreo continuo y en tiempo real de la salud del paciente mediante el uso de Big Data. Se espera ademas, tener la capacidad de identificar patrones y tendencias en los datos médicos. Así como también proporcionar recomendaciones personalizadas.

1.3.3. Proceso

1. **Recopilación de Datos**:
    - Dispositivos IoT (datos en tiempo real)
2. **Almacenamiento y Gestión**:
    - Almacenamiento de datos centralizada, segura y escalable
3. **Análisis de Datos**:
    - Procesamiento en tiempo real
    - Análisis históricos
4. **Generación de Insights**:
    - Tableros
    - Alertas en tiempo real
5. **Intervención y Seguimiento**:
    - Monitoreo continuo
    - Feedback y mejora continua del sistema

Capítulo 2

Marco Teórico

2.1. Introducción a Big Data y Streaming de Datos

2.1.1. Sistemas Distribuidos

Un sistema distribuido es una colección de elementos computacionales autonomos que para su usuario parecen un sistema único y coherente. [17]

Los sistemas distribuidos tienen dos caracteristicas que pueden regularse para escalar: Procesamiento y Almacenamiento.

En el último tiempo, ha habido una tendencia a preferir que la escala de ambas propiedades sea individual. Es decir, que se pueda escalar por un lado la potencia de procesamiento y por otro la capacidad de almacenamiento.

Consistencia

La Consistencia es la propiedad que tiene un sistema distribuido en la que todos los nodos ven los mismos datos al mismo tiempo. Esto significa que cualquier lectura en cualquier momento deberá devolver el valor más reciente escrito para ese dato. Si un sistema es consistente, una vez que se realiza una escritura, todas las lecturas subsiguientes deben reflejar esa escritura; sin importar desde que nodo se hagan. Esta propiedad garantiza que los clientes de los sistemas nunca vean datos desactualizados o inconsistentes.

Disponibilidad

La Disponibilidad es la propiedad que tiene un sistema distribuido para responder a todas las peticiones, ya sean de lectura o escritura, sin fallos. Un sistema disponible garantiza que cada solicitud reciba una respuesta sin importar el estado individual de cada nodo que lo compone. Esto significa que incluso si algunos nodos falla, el sistema en su conjunto debe poder seguir dando servicio a las peticiones que recibe.

Tolerancia a Particiones

La Tolerancia a Particiones es la propiedad que tiene un sistema distribuido en la que continua funcionando a pesar de la perdida de conectividad entre nodos. Una partición ocurre cuando hay una ruptura en la comunicación dentro de la red, lo que resulta en que dos o más segmentos de la red no puedan comunicarse entre sí. Un sistema tolerante a particiones puede seguir operando incluso cuando estas particiones ocurren, lo que significa que puede manejar retrasos o pérdidas de mensajes entre nodos sin fallar por completo.

2.1.2. Definición y características del Big Data

Big Data es un término paraguas que se usa para denominar a un conjunto de tecnologías que manejan grandes volúmenes de datos. La pregunta que se presenta entonces es: ¿qué tan grandes deberían ser estos volúmenes para ser considerados Big Data? O incluso, ¿existen otras características que definan lo que es Big Data? Una definición generalmente aceptada es la siguiente:

Las tecnologías de Big Data están orientadas a procesar datos de alto volumen, alta velocidad y alta variedad para extraer el valor de datos previsto y asegurar una alta veracidad de los datos originales y la información obtenida, lo que demanda formas de procesamiento de datos e información rentables e innovadoras para mejorar el conocimiento, la toma de decisiones y el control de procesos.

Todo esto exige nuevos modelos de datos que soporten todos los estados y etapas de los datos durante todo su ciclo de vida, y nuevos servicios y herramientas de infraestructura que permitan obtener y procesar datos de una variedad de fuentes y entregar datos en una variedad de formas a diferentes consumidores y dispositivos de datos e información. [7]

Por lo que podríamos considerar que es Big Data todo aquello que esté orientado a datos cuyo volumen, velocidad y variedad no puedan ser tratados por un modelo de procesamiento de datos tradicional (como podrían ser las bases de datos relacionales). Con el objetivo de generar valor, asegurando la veracidad de los datos originales y la información obtenida.

2.1.3. Streaming de datos

El streaming de datos, también conocido como procesamiento de flujo, es un paradigma de procesamiento de datos en el que los datos se tratan como un flujo continuo e ilimitado de eventos discretos. En el contexto de Big Data, el streaming permite procesar y analizar grandes volúmenes de datos en tiempo real o casi real, a medida que se generan o llegan al sistema. [8]

2.1.4. Teorema CAP

El Teorema CAP es un concepto fundamental en el diseño de sistemas distribuidos. Este establece que es imposible garantizar al mismo tiempo, tanto la Consistencia (Consistency), Disponibilidad (Availability) y la Tolerancia a las Particiones (Partition Tolerance).

Según esto, un sistema distribuido sólo es capaz de garantizar dos de estas propiedades al mismo tiempo. En general, para los sistemas de Big Data de Streaming, la disponibilidad es una propiedad obligatoria, ya que cualquier inactividad puede resultar en la pérdida de datos valiosos o en la imposibilidad de realizar acciones.

Por otro lado, la Tolerancia a Particiones es también indispensable para estos sistemas, que por su naturaleza requieren que su capacidad de procesamiento este distribuida a través de múltiples nodos dispersos en una red no confiable; por lo que son suceptibles a que se genere una partición. Por lo tanto, si no tuviera esta propiedad el servicio podría dejar de ser disponible.

Entonces, como corolario, un sistema de Big Data de Streaming debe también ser tolerante a las particiones para poder ser disponible. Esto nos deja con una única opción: relajar el "grado de consistencia" hasta un punto razonable que permita que el sistema siga siendo eficáz.[14]

Consistencia Eventual

La consistencia eventual es un modelo de consistencia en sistemas distribuidos que garantiza que, si no se realizan nuevas actualizaciones a un objeto, en algun momento (eventualmente) todos los accesos a ese objeto devolverán el último valor actualizado.

La consistencia eventual se alinea con las compensaciones descritas por el teorema CAP, permitiendo que estos sistemas prioricen la disponibilidad y la tolerancia a particiones.

Además, facilita la escalabilidad horizontal, crucial para manejar el crecimiento continuo de datos y clientes de los sistemas. Por último, es importante diseñar cuidadosamente el sistema para manejar las posibles inconsistencias temporales y asegurar que la aplicación pueda tolerar y resolver estas situaciones de manera apropiada [14]

2.1.5. Desafíos en el manejo de datos de streaming

1. **Procesamiento en tiempo real y baja latencia**

   El procesamiento de datos debe ocurrir con un retraso mínimo para proporcionar resultados en tiempo real.

   Un desafío clave en el streaming es lograr equilibrar la latencia, el costo y la correctitud simultáneamente [1].

2. **Manejo de datos fuera de orden**

   Los datos pueden llegar en un orden diferente al que fueron generados, lo que complica el procesamiento.

   El procesamiento de eventos fuera de orden es un desafío fundamental en los sistemas de streaming [8, p. 87].

3. **Escalabilidad**

   Los sistemas deben poder manejar volúmenes crecientes de datos sin degradación del rendimiento.

   La escalabilidad en sistemas de streaming implica la capacidad de aumentar el rendimiento añadiendo recursos computacionales [15].

4. **Tolerancia a fallos y consistencia**

   El sistema debe poder recuperarse de fallos sin pérdida de datos y mantener la consistencia eventual de los resultados.

   Garantizar la semántica de "exactly-once" en presencia de fallos es un desafío significativo en streaming [4].

5. **Procesamiento de ventanas temporales**

   Definir y procesar eficientemente ventanas de tiempo sobre streams de datos continuos.

   El procesamiento de ventanas temporales es fundamental en aplicaciones de streaming y requiere consideraciones cuidadosas en cuanto a la semántica del tiempo y la completitud de los datos [1].

6. **Integración con sistemas batch**

   Combinar eficazmente el streaming con sistemas batch existentes.

   La integración de paradigmas batch y streaming, a menudo referida como 'procesamiento híbrido', presenta desafíos únicos en términos de consistencia de datos y modelos de programación [4].

2.1.6. Conceptos clave en Streaming

El streaming se refiere al análisis y manipulación de datos en tiempo real a medida que se generan o reciben. Según Carbone et al. [4], los conceptos fundamentales incluyen:

- **Flujo de datos**: Una secuencia potencialmente infinita de registros que llegan continuamente [1].
- **Latencia**: El tiempo entre la llegada de un dato y su procesamiento, crucial para aplicaciones en tiempo real [1].
- **Ventanas**: Mecanismos para agrupar datos en intervalos finitos para su procesamiento [1].
- **Estado**: Información que se mantiene entre eventos para cálculos incrementales [4].
- **Watermarks**: Indicadores de progreso del tiempo en el flujo de datos [1].

2.1.7. Batch vs Streaming

La elección entre batch y streaming depende de los requisitos específicos de la aplicación:

Característica	**Batch**	**Streaming**
Latencia	Alta (horas a días)	Baja (milisegundos a minutos)
Procesamiento	Alto	Moderado a alto
Complejidad	Menor	Mayor
Consistencia	Fuerte	Eventual
Uso típico	Análisis histórico, reportes	Monitoreo, alertas, decisiones inmediatas

Cuadro 2.1: Comparación de batch y streaming

Se puede decir que el streaming es esencial para aplicaciones que requieren decisiones inmediatas, mientras que el análisis batch es más adecuado para análisis profundos de grandes volúmenes de datos históricos.[16]

2.1.8. Evolución de las arquitecturas de procesamiento de datos

La evolución de las arquitecturas de procesamiento de datos ha sido impulsada por la necesidad de manejar volúmenes cada vez mayores de datos en tiempo real:

1. **Arquitecturas batch**: Sistemas tradicionales como Hadoop MapReduce, diseñados para procesar grandes volúmenes de datos estáticos [6].

2. **Arquitecturas de streaming puro**: Como Apache Storm, enfocadas en el procesamiento en tiempo real pero con limitaciones en la consistencia y exactitud [18].

3. **Arquitectura Lambda**: Propuesta por Marz [12], combina procesamiento batch y en tiempo real para balancear latencia, throughput y tolerancia a fallos.

4. **Arquitectura Kappa**: Introducida por Kreps [10], simplifica Lambda tratando todos los datos como streams.

5. **Arquitectura Delta**: Desarrollada por Databricks, también como respuesta a Lambda optimiza el procesamiento de datos tanto batch como streaming [2] [11].

2.2. Tecnologías para Streaming en Big Data

A continuación se discuten las tecnologías que podrían estar involucradas en el desarrollo de un sistema de Streaming en Big Data con el objetivo de poder definir un stack tecnológico que cumpla con los requisitos de los sistemas a construir.

2.2.1. Mensajería Distribuida

Las tecnologías de mensajería distribuidas en tiempo real cumplen el crucial rol de actuar como intermediarios entre las fuentes de datos y los sistemas que efectivamente procesan estos datos. [9]

Deben funcionar como un conducto de alta capacidad de almacenamiento y baja latencia, capturando y canalizando los flujos de información desde sus múltiples orígenes y hacia sus diversos destinos en tiempo real. [13]

Su papel es fundamentalmente el de un sistema nervioso central, coordinando y distribuyendo datos a través de complejas arquitecturas distribuidas. Actúan como amortiguadores, absorbiendo picos en el flujo de datos y garantizando un procesamiento constante y eficiente. Además, estas tecnologías sirven como una capa de abstracción, desacoplando los productores de datos de los consumidores, lo que permite una mayor flexibilidad y escalabilidad en el diseño del sistema.

Apache Kafka

Apache Kafka es el estandar de facto de este tipo de sistemas. Utiliza un modelo de publicación-subscripcion (pub/sub) basado en logs, donde los datos se envían a "topics" y se almacenan en particiones distribuidas. Las particiones tienen una garantía de orden de los mensajes y permiten la retención de datos a largo plazo.

Además, proporciona conectores para integración con diversos sistemas y un amplio ecosistema. A nivel de seguridad ofrece encriptación en tránsito mediante TLS y permite ser configurado para soportar encriptación en reposo (aunque esto debe hacerse a nivel de sistema de archivos).

Por último, su arquitectura distribuida y replicada permite una alta disponibilidad y tolerancia a fallos.

Apache Pulsar

Apache Pulsar es también una plataforma de mensajería y streaming distribuida, al igual que Apache Kafka, pero que se distingue por tener una arquitectura basada en capas, separando la capa de almacenamiento de la capa de procesamiento; lo que permite escalar cada uno independientemente.

Apache Pulsar soporta modelos de entrega como colas, publicación y suscripción y puede ofrecer garantías de entregar un mensaje exactamente una vez ("exactly-once"). Ofrece encriptación a nivel de mensaje, lo que permite una granularidad fina en cuanto a que encriptar. También soporta TLS para la encriptación en tránsito y permite la configuración de encriptación en reposo.

Por último, también soporta almacenamiento de mensajes a largo plazo y soporte nativo para esquemas: Esto es, permite definir la estructura y el tipo de datos de los mensajes, lo que a su vez permite una validación automática de los datos y una serialización/deserialización más eficiente. Esto trae consigo además, la capacidad de evolucionar estos esquemas de mensajes, de forma que productores y consumidores evolucionen independientemente.

Amazon Kinesis

Amazon Kinesis es un servicio de streaming de datos administrado en la nube de AWS. Está diseñado para recopilar, procesar y analizar datos de streaming en tiempo real a gran escala. Kinesis, en realidad, se compone de varios servicios:

1. Kinesis Data Streams para ingestión de datos en tiempo real
2. Kinesis Data Firehose para cargar datos en los servicios de almacenamiento disponibles de AWS
3. Kinesis Data Analytics para procesar datos con SQL o Java
4. Kinesis Video Streams para streaming de video

Adicionalmente ofrece capacidades de auto-escalado, replicación entre zonas de disponibilidad para alta durabilidad, encriptación en reposo (y puede habilitarsele la encriptación en tránsito) y permite la retención de datos hasta 365 días.

Como se puede ver, al ser tan completo permitiría implementar, al menos en principio, una gran parte de un sistema de Big Data en tiempo real.

Azure Event Hubs

Azure Event Hubs es un servicio de ingestión de datos en tiempo real administrado en la plataforma Microsoft Azure. Se supone que está diseñado para soportar millones de eventos por segundo con baja latencia. Al igual que Kinesis, ofrece una muy buena con otros servicios de Microsoft Azure, lo que permite por ejemplo capturar directamente los eventos en los servicios de almacenamiento disponibles en este proveedor de nube.

Event Hubs es compatible con el protocolo Kafka, lo que permite a las aplicaciones existentes de Kafka conectarse sin cambios de código. Ofrece una retención de mensajes por defecto de 1 dia que puede aumentado hasta 7. En caso de necesitar una retención más a largo plazo se recomienda guardar los eventos en Azure Blob Storage o Azure Data Lake para su posterior procesamiento. Cuenta también con encriptación en transito con TLS y en reposo.

Proporciona además, características como el procesamiento batch para optimizar el rendimiento, control de acceso basado en roles, y encriptación en reposo y en tránsito.

2.2.2. Comparación

Escalabilidad

- **Apache Kafka:** Muy Alta escalabilidad horizontal, millones de mensajes/segundo.
- **Apache Pulsar:** Muy alta escalabilidad horizontal, millones de mensajes/segundo con separación de almacenamiento y cómputo.
- **Amazon Kinesis:** Buena escalabilidad, con 1.000 mensajes por segundo con la configuración por defecto aunque con configuración adicional puede llegar al millon por segundo hipotético.
- **Azure Event Hubs:** Buena escalabilidad, con 1.000 mensajes por segundo. También puede escalar con configuración adicional a los 20.000 mensajes. Existe la posibilidad de tener una instancia dedicada que permite escalar a millones de eventos por segundo de forma hipotética.

Retención de datos

- **Apache Kafka:** Configurable, potencialmente ilimitada.
- **Apache Pulsar:** Ilimitada por diseño.
- **Amazon Kinesis:** Hasta 365 días, configurable.
- **Azure Event Hubs:** Hasta 7 días, opción de Capture para largo plazo.

Garantías de entrega

- **Apache Kafka:** At-least-once por defecto, exactly-once configurable.
- **Apache Pulsar:** Exactly-once nativo.
- **Amazon Kinesis:** At-least-once.
- **Azure Event Hubs:** At-least-once.

Encriptación

- **En reposo:**
  - **Apache Kafka:** Configurable.
  - **Apache Pulsar:** Nativo.
  - **Amazon Kinesis:** Por defecto (AWS KMS).
  - **Azure Event Hubs:** Por defecto.
- **En tránsito:** Todos soportan TLS/SSL.

Observaciones clave

- Pulsar destaca en escalabilidad, retención, seguridad y consistencia nativos.
- Kafka ofrece el ecosistema más maduro y desarrollado que le otorga características muy completas.
- Servicios gestionados (Kinesis, Event Hubs) tienen encriptación en reposo por defecto, pero retención limitada.

Característica	**Kafka**	**Pulsar**	**Kinesis**	**Event Hubs**
Escalabilidad	Muy Alta	Muy alta	Alta	Alta
Retención	Configurable	Ilimitada	365 días máx.	7 días máx.
Garantías	Exactly-once	Exactly-once	At-least-once	At-least-once
Enc. en reposo	Configurable	Sí	Sí (AWS KMS)	Sí
Enc. en tránsito	Sí	Sí	Sí	Sí
Throughput	Muy alto	Muy alto	Alto	Alto

Cuadro 2.2: Comparación de Sistemas de Mensajería

2.2.3. Motores de Procesamiento

Si las tecnologías de mensajería distribuida son el sistema nervioso de un sistema de Big Data de Streaming, los motores de procesamiento podrían considerarse su cerebro.

Actúan como la capa que transforma, enriquece y analiza los datos cumpliendo varios roles:

- Aplicar la lógica de negocio sobre los datos mientras estos fluyen
- Detectar patrones y anomalías sobre el flujo de datos
- Mantener el contexto y estado necesario para las operaciones con históricos o agregaciones
- Garantizar la consistencia de las operaciones incluso ante fallos del sistema
- Distribuir la carga de trabajo entre diferentes nodos, paralelizando tareas

Estos componentes pueden enlazarse y programarse de diversas maneras. Permitiendo ensamblarlos de forma de cumplir con los requisitos de negocio.

Apache Spark

Apache Spark se destaca como uno de los motores de procesamiento más populares y versátiles en el ecosistema de Big Data. Ofrece dos APIs principales para el procesamiento en tiempo real: Spark Streaming y Structured Streaming, siendo esta última la más moderna y recomendada.

Implementa el procesamiento en tiempo real tratando los datos streaming como micro-batches y su enfoque de procesamiento es en memoria, siendo capaz de mantener su estado distribuido a través de checkpoints, que son archivos que se guardan en un sistema de almacenamiento al que todos los nodos pueden acceder.

Su modelo de procesamiento le permite ofrecer un cierto equilibrio entre latencia y throughput. La abstracción fundamental de Spark son los RDDs (Resilient Distributed Datasets), que son colecciones inmutables de datos distribuidos que pueden ser procesadas en paralelo, y los DataFrames, que proporcionan una abstracción de más alto nivel similar a una tabla de base de datos.

Spark destaca por su amplio soporte de lenguajes de programación:

- Scala
- Python
- Java
- R
- SQL

Su API unificada y extenso catálogo de bibliotecas incluye MLlib para machine learning, GraphX para procesamiento de grafos, y Spark SQL para procesamiento estructurado.

Apache Flink

Apache Flink adopta un enfoque nativo de streaming, tratando al procesamiento batch como un caso especial de streaming con límites finitos. Su arquitectura está diseñada para mantener estado distribuido con garantías de consistencia muy fuertes y latencias extremadamente bajas.

El motor gestiona automáticamente la distribución del estado y los checkpoints, asegurando semánticas de exactly-once y permitiendo recuperación exacta ante fallos sin duplicados. Su modelo de procesamiento se basa en dos conceptos fundamentales:

- Marcas de agua (Watermarks): Son metadatos que fluyen en el stream de datos indicando el progreso del tiempo del evento, permitiendo manejar datos desordenados.
- Ventanas de tiempo (Windows): Permiten agrupar y procesar datos en intervalos temporales definidos, soportando diversos tipos como tumbling, sliding y session windows.

Flink proporciona APIs de diferentes niveles:

- ProcessFunction: API de bajo nivel que ofrece máximo control sobre tiempo, estado y ventanas
- DataStream API: API de alto nivel para operaciones de streaming comunes
- Table API y SQL: APIs declarativas para operaciones relacionales

Los lenguajes soportados son:

- Java
- Scala
- Python
- SQL

Apache Beam

Apache Beam, por su lado, se distingue por proporcionar un modelo de programación unificado que abstrae el motor de ejecución subyacente. Su potencia radica en la capacidad de escribir la lógica de procesamiento una vez y ejecutarla en diferentes motores de procesamiento (runners) como Spark o Flink.

Esta capacidad de abstracción es particularmente valiosa en escenarios donde la portabilidad y la flexibilidad de despliegue son requisitos clave, permitiendo cambiar de motor de procesamiento según evolucionen las necesidades y sin reescribir el código de procesamiento.

Apache Samza

Apache Samza se distingue por su estrecha integración con Apache Kafka y su arquitectura diseñada para mantener el estado de procesamiento de forma distribuida con un modelo de particionamiento que permite escalar horizontalmente.

Samza proporciona un modelo de procesamiento simple pero potente, con fuerte énfasis en la gestión de estado local y la tolerancia a fallos. Se utiliza en Linkedin y la arquitectura Kappa fue propuesta inicialmente pensando en la utilización de este motor de procesamiento.

Apache NiFi

Apache NiFi aborda el procesamiento de datos desde una perspectiva de orquestación y gobierno de datos; centrándose en la automatización del flujo de datos entre sistemas.

Su arquitectura está orientada a la trazabilidad y auditabilidad de cada dato que fluye por el sistema, manteniendo un registro detallado de todas las transformaciones y movimientos. Los datos fluyen a través de un grafo de procesadores que pueden transformar, enrutar y mediar entre diferentes protocolos y formatos. NiFi se destaca por su capacidad para garantizar la entrega confiable de datos, proveer linaje de datos completo y permitir modificaciones de flujos en tiempo real sin necesidad de detener el sistema.

Por último, NiFi proporciona una interfaz visual para diseñar, controlar y monitorizar flujos de datos.

Apache Kafka Streams

Apache Kafka Streams es una biblioteca de procesamiento de streaming que forma parte del ecosistema de Apache Kafka, diseñada para construir aplicaciones y microservicios de procesamiento en tiempo real.

Opera con un modelo de procesamiento que permite operaciones con manejo de estado, incluyendo agregaciones por ventanas, manejo de múltiples streams de datos y transformaciones complejas mediante APIs de alto y bajo nivel. Tiene un manejo de estado distribuido que se gestionan con los mismos logs de Kafka.

En cuanto a rendimiento, Kafka Streams alcanza una latencia típica de decenas de milisegundos a segundos, dependiendo de la complejidad del procesamiento y la configuración, mientras que su throughput puede escalar linealmente añadiendo más instancias.

También ofrece grantías de procesamiento at-least-once con posibilidad de exactly-once mediante configuración.

Apache Pulsar Functions

Apache Pulsar Functions ofrece capacidad de cómputo integrandose directamente en la infraestructura de Apache Pulsar.

Este framework permite implementar funciones livianas que procesan mensaje a mensaje. El manejo del estado es distribuido y se realiza mediante un almacenamiento basado en RocksDB, que permite mantener información entre invocaciones de funciones de manera consistente y tolerante a fallos. En términos de rendimiento, está optimizado para baja latencia, típicamente en el rango de milisegundos, gracias a su modelo de procesamiento directo.

El throughput puede escalar horizontalmente añadiendo más instancias de funciones, y el sistema proporciona garantías de procesamiento exactly-once La arquitectura de este sistema está diseñada para ser simple y eficiente, permitiendo casos de uso como enriquecimiento de datos, filtrado, y transformaciones en tiempo real.

2.2.4. Comparación

Modelo de Procesamiento

- **Apache Spark:** Micro-batches.
- **Apache Flink:** Streaming nativo con procesamiento registro a registro.
- **Apache Beam:** Mismo que el modelo de su motor asociado
- **Apache Samza:** Streaming nativo con procesamiento basado en tiempo
- **Apache NiFi:** Procesamiento basado en flujos de datos dirigidos
- **Kafka Streams:** Procesamiento de baja complejidad mensaje a mensaje integrado con Kafka
- **Pulsar Functons:** Procesamiento de baja complejidad mensaje a mensaje integrado con Pulsar

Manejo de Estado

- **Apache Spark:** Estado en memoria distribuido entre los nodos procesadores.
- **Apache Flink:** Estado distribuido utilizando snapshots.
- **Apache Beam:** Mismo que el manejo de su motor asociado
- **Apache Samza:** Estado local con respaldo en Kafka
- **Apache NiFi:** Se maneja localmente en memoria en el nodo que procesa el flujo
- **Kafka Streams:** Estado local con respaldo en Kafka
- **Pulsar Functons:** Almacenado y gestionado con la instancia de manejo de almacenamiento (BookKeeper) de Pulsar

Latencia

- **Apache Spark:** Dependiendo de la configuración del micro-batch puede ir desde milisegundos a segundos.
- **Apache Flink:** Microsegundos a milisegundos.
- **Apache Beam:** Mismo que el de su motor asociado
- **Apache Samza:** Milisegundos a segundos
- **Apache NiFi:** Segundos a Minutos
- **Kafka Streams:** Milisegundos a segundos
- **Pulsar Functons:** Milisegundos a segundos

Capacidad de Procesamiento

- **Apache Spark:** Cientos de miles de eventos por segundo por nodo
- **Apache Flink:** Millones de eventos por segundo por nodo
- **Apache Beam:** Mismo que el de su motor asociado
- **Apache Samza:** Cientos de miles de eventos por segundo por partición
- **Apache NiFi:** Miles de eventos por segundo por nodo
- **Kafka Streams:** Cientos de miles de eventos por segundo por partición
- **Pulsar Functons:** Decenas de miles de eventos por segundo por nodo

Garantías de Entrega

- **Apache Spark:** at-least-once por defecto aunque puede ser configurado para permitir exactly-once
- **Apache Flink:** exactly-once nativo
- **Apache Beam:** exactly-once, at-least-once y at-most-once dependiendo del motor utilizado
- **Apache Samza:** at-least-once por defecto aunque puede ser configurado para permitir exactly-once
- **Apache NiFi:** at-least–once con garantía de entrega a pesar de fallas del sistema
- **Kafka Streams:** at-least-once y exactly-once
- **Pulsar Functons:** mejor rendimiento con at-least-once pero soporta exactly-once y at-most-once

Observaciones clave

- Spark es el que tiene un ecosistema más extendido y es más accesible
- Flink tiene las mejores prestaciones a nivel de latencia y rendimiento
- NiFi no es una buena opción para streaming por su latencia
- Kafka Streams y Pulsar Functions pueden ser usados de forma ligera para ruteo de mensajes

	Modelo	**Estado**	**Latencia**	**Proc.**	**Garantías**
Apache Spark	Micro-batches	En memoria	Baja	Alto	Exactly-once
Apache Flink	Streaming	Distribuido	Muy baja	Muy Alto	Exactly-once
Apache Beam	Variable	Variable	Variable	Variable	Variable
Apache Samza	Streaming	Por partición	Baja	Alto	Exactly-once
Apache NiFi	Flujo dirigido	Procesador	Media-Alta	Medio	At-least-once
Kafka Streams	Streaming	Distribuido	Baja	Alto	Exactly-once
Pulsar Functions	Mensaje	Distribuido	Baja	Alto	Exactly-once

Cuadro 2.3: Comparativa de Motores de Procesamiento Big Data

2.2.5. Almacenamiento de Datos

Formatos de Almacenamiento

Los formatos de datos son un componente fundamental en cualquier arquitectura de sistemas de información moderna, ya que determinan no solo cómo se almacena la información, sino también cómo se procesa, transmite y analiza.

La elección adecuada del formato de datos puede tener un impacto significativo en el rendimiento, la escalabilidad y la eficiencia del sistema en su conjunto. Para un sistema de Big Data, donde se manejan grandes volúmenes de información, la importancia de estos formatos se magnifica, ya que pueden significar la diferencia entre un sistema eficiente y uno que consume recursos excesivos.

Además, los formatos de datos actúan como un lenguaje común entre diferentes componentes, facilitando la interoperabilidad y la integración de tecnologías.

Formatos Orientados a Filas

Los formatos orientados a filas representan la forma tradicional de almacenamiento de datos, donde cada registro se almacena de manera secuencial. Este enfoque ha sido la base de los sistemas de gestión de bases de datos durante décadas y sigue siendo crucial en muchos escenarios.

- Los registros completos se almacenan de manera contigua en disco
- Cada fila contiene todos los campos de un registro
- Optimizado para acceder a registros completos
- Los nuevos registros se añaden secuencialmente de forma eficiente
- Óptimo cuando las consultan necesitan todos los campos
- Fácil modificación de registros individuales
- Debe leer datos innecesarios cuando solo se necesitan algunas columnas
- Los datos heterogéneos juntos reducen la efectividad de los métodos de compresión
- Menos eficiente para análisis de columnas específicas

Formatos Orientados a Columnas

Los formatos de almacenamiento columnar representan un paradigma fundamental en el manejo de datos masivos, especialmente en entornos analíticos. A diferencia del almacenamiento tradicional orientado a filas, donde los registros se almacenan secuencialmente, el almacenamiento columnar organiza los datos por columnas, lo que ofrece ventajas significativas en ciertos escenarios.

- En lugar de almacenar registros completos de manera contigua, los datos se organizan por columnas
- Cada columna se almacena en bloques separados de memoria o disco
- Los valores similares se almacenan juntos, mejorando la compresión
- Los datos similares almacenados juntos permiten mayores tasas de compresión
- Solo se leen las columnas necesarias para una consulta
- Facilita operaciones como SUM, AVG, COUNT sobre columnas específicas
- Permite procesamiento eficiente de datos en hardware

Comparativa con Almacenamiento por Filas:

Consideremos una tabla simple de usuarios:
Formato por Filas:

```
[ID1, "Juan", 25] -> [ID2, "Ana", 30] -> [ID3, "Pedro", 28]
```

Formato Columnar:

```
IDs:     [ID1 -> ID2 -> ID3]
Nombres: ["Juan" -> "Ana" -> "Pedro"]
Edades:  [25 -> 30 -> 28]
```

Aspecto	Formato Columnar	Formato por Filas
Lectura parcial	Muy eficiente	Menos eficiente
Inserción de registros	Más lenta	Más rápida
Compresión	Alta	Moderada
Consultas analíticas	Excelente	Regular
Consultas transaccionales	Regular	Excelente

Formatos Específicos

JSON (JavaScript Object Notation) se ha convertido en el estándar de facto para el intercambio de datos en aplicaciones modernas, especialmente en entornos Web y APIs. Su popularidad se debe a su simplicidad, legibilidad humana y amplia compatibilidad con prácticamente todos los lenguajes de programación.

A pesar de no ser el más eficiente en términos de espacio y rendimiento (ya que es un formato basado en filas), su flexibilidad para representar datos estructurados y semiestructurados lo hace invaluable en sistemas donde la interoperabilidad y la facilidad de desarrollo son prioritarias.

Es particularmente útil en aplicaciones donde las transacciones individuales y la flexibilidad del esquema son más importantes que la eficiencia en el procesamiento de grandes volúmenes de datos.

Apache AVRO destaca como un formato de serialización de datos binario que combina la eficiencia del almacenamiento binario con la flexibilidad de esquemas evolutivos.

Su característica más distintiva es su capacidad para manejar cambios en el esquema de datos a lo largo del tiempo sin requerir cambios en el código o reescritura de datos existentes. Para esto, AVRO almacena el esquema junto con los datos, lo que permite una deserialización precisa y eficiente.

Es especialmente valioso en sistemas de mensajería y streaming de datos, donde la evolución del esquema y la eficiencia en la transmisión son cruciales. Su formato binario compacto y su capacidad de compresión lo hacen ideal para sistemas distribuidos donde el ancho de banda y el almacenamiento son consideraciones importantes.

Apache Parquet se ha establecido como el formato columnar dominante en el ecosistema de Big Data, especialmente para cargas de trabajo analíticas. Su diseño columnar permite una compresión altamente eficiente y un muy buen rendimiento en consultas que involucran solo un subconjunto de columnas.

Parquet destaca particularmente en escenarios de análisis de datos, donde su capacidad para manejar esquemas complejos anidados y su integración con casi todas las herramientas lo hacen indispensable. La adopción generalizada de Parquet en la industria, lo ha convertido en el estándar de facto para almacenamiento de datos analíticos.

Optimized Row Columnar (ORC) inicialmente fue desarrollado para optimizar Hive, y aunque ofrece excelentes capacidades de compresión y rendimiento en consultas, su relevancia ha disminuido significativamente en los últimos años frente a Parquet.

Aunque ORC sigue siendo relevante en sistemas legacy y específicos de Hive, la tendencia de la industria se ha movido claramente hacia Parquet como el formato columnar preferido para análisis de datos a gran escala.

Almacenamiento de Objetos en la Nube

El almacenamiento de objetos en la nube constituye un paradigma de almacenamiento donde los datos se organizan y gestionan como objetos independientes dentro de una estructura plana. Cada objeto almacenado comprende tres elementos fundamentales: los datos en sí mismos, un conjunto extenso de metadatos que describen y categorizan la información, y un identificador único global que permite su localización y recuperación.

Dicho paradigma, se caracteriza por su naturaleza distribuida y su capacidad para manejar tanto datos estructurados como no estructurados, permitiendo almacenar desde documentos, archivos multimedia y hasta flujos de datos en tiempo real.

Los sistemas de almacenamiento de objetos se destaca por su capacidad para satisfacer las demandas contemporáneas de procesamiento de datos a gran escala. Como por ejemplo, ofrece una escalabilidad prácticamente ilimitada, pudiendo crecer según las necesidades sin preocuparse por restricciones de capacidad.

Por otro lado, la durabilidad y disponibilidad de los datos se garantiza a través de la replicación automática en múltiples ubicaciones de forma automática y transparente. Adicionalmente, proveen APIs normalmente basadas en el protocolo HTTP que permite acceder de forma interoperable y estandarizada a los recursos almacenados.

El uso de estos sistemas también conlleva sus propios desafíos. El más importante puede ser la latencia; pero también deben los grados de consistencia que ofrecen.

Formato de Tabla Analítica

Los formatos de tabla analítica son tecnologías diseñadas para resolver los desafíos del manejo de datos a gran escala. Surgen como respuesta a las limitaciones de los formatos de archivo tradicionales como Parquet y ORC cuando se trabaja con ellos en la nube.

Las características más importantes que aporta un formato de tabla analítica son:

- Proveen abstracciones sobre la metadata de archivos
- Permiten el uso de tablas con semántica SQL y evolución de esquema en las mismas
- Transacciones ACID
- Actualizaciones y Borrados
- Optimización de datos para mejoras de rendimiento
- Compatibilidad con múltiples motores de procesamiento
- Control de versiones en los datos

Los formatos de tablas analíticas dan a sus sistemas subyacentes estas características a través de diferentes mecanismos y estrategias. La principal es la gestión de metadatos, ya que implementan estructuras de datos altamente optimizadas que permiten rastrear eficientemente los archivos y sus cambios; mientras mantienen un historial detallado de transacciones que garantiza la consistencia de los datos, complementando con estrategias efectivas de particionamiento y organización.

Para la optimización del rendimiento, estos formatos emplean diversas técnicas como la compactación automática de archivos, estrategias de caching de datos para acceso rápido, utilización de formatos de archivo columnares como Parquet u ORC, y la incorporación de capacidades avanzadas de indexación y filtrado optimizado.

La consistencia de los datos se garantiza mediante la implementación de transacciones ACID completas, que proporcionan un sólido aislamiento entre operaciones de lectura y escritura, manejan conflictos de manera automática y aseguran la consistencia en escenarios de operaciones concurrentes. En cuanto a la evolución y mantenimiento, estos formatos facilitan cambios de esquema sin interrupciones en el servicio.

Todas estas mecanismos trabajan en conjunto para proporcionar una solución completa para el manejo de datos a escala masiva, que como subproducto permite tratar la escritura de los archivos como un canal de mensajes sobre el que se puede hacer streaming.

Los exponentes más importantes de estos formatos son: Delta Table, Apache Iceberg y Apache Hudi. Existen además otros formatos menos conocidos como Apache Paimon, que aunque no tienen la misma adopción, ofrecen características similares.

Delta Lake es un sistema de almacenamiento de datos diseñado por Databricks que utiliza archivos Parquet como base, organizándolos en una estructura de directorios con dos componentes principales:

- Los archivos de datos en formato Parquet
- El directorio _delta_log para metadatos y registro de transacciones

Esta arquitectura mantiene las ventajas de Parquet mientras añade capacidades transaccionales y de control de versiones, implementando un sistema de checkpoints para optimizar el rendimiento y un manejo de concurrencia que combina control optimista con serialización de escrituras.

Las características fundamentales de Delta Lake incluyen transacciones ACID completas, capacidad de acceso a versiones anteriores, evolución de esquema controlada, operaciones de Merge sofisticadas y optimización automática de datos mediante compactación de archivos y mantenimiento de estadísticas.

El sistema también proporciona soporte para procesamiento de datos en tiempo real con semántica exactly-once y una integración robusta principalmente con Spark.

Apache Hudi desarrollado inicialmente por Uber, es una plataforma enfocada en crear una plataforma analítica transaccional, disponibilizando dos tipos principales de formatos de tablas: Copy On Write (optimizado para lecturas) y Merge On Read (optimizado para escritura).

Utiliza una linea de tiempo para gestionar metadatos y registrar cronológicamente todas las acciones, implementando un sistema de indexación que permite optimizar operaciones y facilitar búsquedas rápidas, además maneja control de concurrencia optimista que mantiene lecturas sin bloqueos mientras serializa escrituras.

Las características principales de Hudi incluyen procesamiento incremental para manejar streams de datos, gestión sofisticada de registros individuales con versionado a nivel de registro, borrado lógico y evolución de esquemas. También proporciona garantías de consistencia con transacciones ACID y semántica exactly-once, ofreciendo buena integración con motores de procesamiento como Spark y Flink.

Hudi, al ser una plataforma, no ofrece solo su formato de tabla analítica, sino también "Table Services" que son componentes computacionales que permiten la optimización como compactación automática y limpieza de almacenamiento.

Apache Iceberg diseñado inicialmente por Netflix, implementa una arquitectura de almacenamiento que se caracteriza por un modelo de metadatos enfocado en la evolución del esquema.

Utiliza una estructura jerárquica que separa completamente los metadatos de los datos, implementando control de versiones basado en snapshots atómicos e inmutables, donde cada snapshot representa un punto en el tiempo de la tabla y contiene referencias a todos los archivos de datos válidos para esa versión, permitiendo operaciones concurrentes sin necesidad de bloqueos pesados.

Entre sus características principales destacan una gestión de esquemas flexible que permite evolucionar tanto el esquema como la estrategia de particionamiento sin reescribir datos, una optimización de consultas avanzada basada en estadísticas detalladas a nivel de columna y archivos, y un sistema de control de concurrencia optimista.

Además, disponibiliza herramientas de mantenimiento como expiración de snapshots, compactación de archivos y reescritura de datos para optimización física, junto con una robusta integración con diferentes motores de procesamiento como Spark y Flink.

Apache Paimon es un formato de tabla analítica que busca permitir la construcción de arquitecturas de streaming.

Nació como un proyecto dentro del mismo Apache Flink por lo que tiene una integración muy fuerte con este motor de procesamiento. Paimon se destaca por su capacidad de manejar datos en tiempo real y batch, permitiendo la construcción de pipelines híbridos. Lo hace mediante el uso de su formato de datos propio en conjunto con mecanismos para la gestión de metadatos que son similares a Apache Hudi.

Permite actualizaciones en tiempo real a través del motor de procesamiento y deduplicar los datos de forma personalizada. También se encarga automáticamente de la limpieza y compactación de archivos, y ofrece soporte para la evolución de esquemas, control de versiones y transacciones.

Catálogos de Metadatos

Los catálogos de metadatos actúan como un registro centralizado y organizado de toda la información sobre las tablas y conjuntos de datos en un sistema. Son una capa de abstracción que mantiene información crítica sobre la estructura, ubicación, esquema, particiones, historial de versiones y estadísticas de los datos, permitiendo una gestión eficiente y un acceso optimizado a los mismos.

Son necesarios para que sistemas externos conozcan la ubicación y la estructura de los datos almacenados.

Hive Metastore es la implementación más adoptada y el estándar de estos catálogos. Opera como un servicio centralizado que almacena la información sobre las estructuras de datos, típicamente usando una base de datos relacional como base (por ejemplo PostgreSQL o MySQL). Sin embargo, puede presentar problemas con operaciones concurrentes complejas. Sin embargo, se pueden implementar capas de abstracción que permitan manejar estos escenarios, y generalmente se prefiere utilizar este sistema ya que está ampliamente probado.

Project Nessie es otra implementación de catálogo de datos, aunque con un enfoque moderno, aplicando conceptos de control de versiones a los datos; similar a como lo hace git. No intenta mantener compatibilidad con Hive Metastore, sino que ofrece un nuevo paradigma de gestión de datos, por lo que puede implementar características más complejas. Sin embargo, esto también significa que requiere más esfuerzo en integración con herramientas existentes ya que su ecosistema aún no está tan desarrollado.

Apache Polaris es un proyecto relativamente nuevo que busca estandarizar y modernizar la gestión de metadatos. Es un esfuerzo para estandarizar y unificar los catálogos de metadatos. De esta manera, Apache Polaris busca ser una solución neutral y agnóstica a los distintos proveedores de catálogos.

Bases de Datos Analíticas

Las bases de datos analíticas, también conocidas como bases de datos OLAP (Online Analytical Processing) orientadas a tiempo real, son sistemas especializados diseñados para procesar y analizar grandes volúmenes de datos con énfasis particular en consultas complejas y agregaciones. Estos sistemas están arquitecturalmente optimizados para procesar rápidamente consultas que involucran múltiples dimensiones y métricas sobre conjuntos masivos de datos, permitiendo análisis en tiempo real o casi real.

Estas bases de datos se distinguen por su capacidad de manejar cargas de trabajo analíticas complejas mientras mantienen latencias bajas y consistentes; especializandose en resultados en segundos o milisegundos.Esta característica se debe a su arquitectura orientada específicamente al análisis, que contrasta con los sistemas diseñados primariamente para transacciones (OLTP) o almacenamiento general de datos.

Por diseño permiten un análisis multidimensional eficiente, soportan alta concurrencia de usuarios, y pueden integrarse efectivamente con fuentes de datos en streaming. Además, su diseño orientado a columnas permite una compresión más eficiente y mejor rendimiento en consultas analíticas que típicamente involucran solo un subconjunto de ellas. Proporcionando capacidades avanzadas de agregación y pueden manejar eficientemente tanto datos históricos como en tiempo real.

Ninguno de estos beneficios vienen sin sus propios desafios, ya que se requiere una cuidadosa planificacion y operativa para mantener su rendimiento. Además, más que nunca, es necesario organizar correctamente los indices y los esquemas de particionamiento son claves.

Ejemplos de estos sistemas son:

Apache Druid es una base de datos analítica distribuida diseñada principalmente para análisis en tiempo real de grandes volúmenes de datos de series temporales. Su arquitectura se distingue por su capacidad de ingesta en tiempo real combinada con consultas de baja latencia, utilizando un modelo de almacenamiento columnar híbrido que combina datos en memoria con almacenamiento en disco.

Apache Pinot fue desarrollado inicialmente por LinkedIn y se enfoca en proporcionar análisis en tiempo real con latencias extremadamente bajas, incluso en escenarios de alta concurrencia de usuarios. Su arquitectura está optimizada para consultas de lectura masivas y paralelas. Además, destaca por su modelo de consistencia eventual y su capacidad para manejar esquemas dinámicos.

Apache Doris inicialmente fue desarrollado por Baidu, integra capacidades de almacenamiento columnar MPP (Procesamiento Paralelo Masivo) con funcionalidades OLAP, ofreciendo una solución más cercana a una base de datos tradicional pero con capacidades analíticas avanzadas. Su arquitectura es más simple en comparación con Druid y Pinot, lo que facilita la operación y mantenimiento, manteniendo un buen rendimiento para consultas analíticas. Permite realizar consultas federadas sin necesidad de ingestar información duplicada.

2.2.6. Comparación

Observaciones clave

- Apache Parquet es el formato más utilizado para análisis de datos
- El almacenamiento de objetos especialmente en la nube es extremadamente barato y su uso permite extender los casos de uso de almacenamiento
- De entre los formatos de tabla analítica, Apache Iceberg es el que se encuentra en mejor posición de adopción debido a su naturaleza abierta
- Hive Metastore sigue siendo el estandar de facto de la industria debido a su amplia adopción y madurez
- Apache Doris permite reducir la duplicación de datos al tener capacidades tanto de almacenamiento en tiempo real como de consultas federadas

2.2.7. Tecnologías Complementarias

Consideramos tecnologías complementarias, aquellas que no son directamente parte de la arquitectura, pero que son necesarias para su correcto funcionamiento.
Estas tecnologías son sobre todo referentes a la gestión de contenedores, monitorización y visualización de datos.

Docker

Docker es una plataforma de software que permite crear, desplegar y ejecutar aplicaciones en contenedores. Los contenedores son entornos ligeros y portátiles que encapsulan una aplicación y todas sus dependencias, lo que garantiza que funcionen de manera consistente en diferentes entornos.

A nivel de desarrollo, también permite crear imágenes de contenedores que pueden ser compartidas y reutilizadas, facilitando la colaboración entre equipos y la integración continua. Es especialmente sencillo de usar y provee Docker Compose, una herramienta que permite definir y ejecutar aplicaciones multi-contenedor.

Prometheus

Prometheus es un sistema de monitorización y alerta open-source, Es especialmente eficaz para monitorizar entornos dinámicos como los que se encuentran en arquitecturas de microservicios y contenedores. Permite recolectar y analizar métricas en tiempo real de manera altamente eficiente y tiene un potente lenguaje de consultas.

Destaca por su capacidad para manejar millones de métricas simultáneamente y su arquitectura pull-based, que le permite escalar horizontalmente sin problemas. Además, cuenta con un sistema de alertas flexible y puede integrarse fácilmente con Docker y otras herramientas de orquestación, lo que lo convierte en una herramienta fundamental para monitorizar aplicaciones modernas a gran escala.

Grafana

Grafana es una plataforma de visualización y análisis de datos de open-source que destaca especialmente cuando se combina con Prometheus. Su principal valor radica en su capacidad para transformar datos complejos en visualizaciones claras e interactivas a través de tableros personalizables.

Cuando se utiliza junto con Prometheus, Grafana actúa como la capa de visualización, permitiendo crear paneles intuitivos que muestran en tiempo real el estado y rendimiento de los sistemas. Esto facilita la detección temprana de problemas, el análisis de tendencias y la toma de decisiones basada en datos.

Una de sus características más poderosas es la capacidad de combinar datos de múltiples fuentes en un único dashboard, proporcionando una vista unificada del sistema.

2.3. Desafíos en Arquitecturas de Streaming

Diseñar e implementar sistemas de procesamiento de datos en tiempo real conlleva varios desafíos técnicos y operativos. A continuación se presentan los más relevantes para arquitecturas de streaming como Kappa y Delta.

2.3.1. Procesamiento Fuera de Orden

En sistemas distribuidos, los eventos pueden llegar desordenados debido a retrasos de red o procesamiento. Para mitigar este problema, se utilizan:

- **Ventanas con tolerancia al desorden** (*allowed lateness*)
- **Algoritmos tolerantes al orden de llegada**

Ambas soluciones incrementan la latencia del sistema, por lo que se debe alcanzar un equilibrio entre precisión y velocidad.

2.3.2. Manejo de Archivos Pequeños

El procesamiento continuo genera muchos archivos pequeños, lo que reduce la eficiencia de lectura/escritura y afecta negativamente el rendimiento. Para resolverlo se aplican técnicas como:

- Compactación periódica
- Uso de formatos columnar como Parquet
- Estrategias de particionamiento y organización eficientes

2.3.3. Gestión del Estado

El manejo de estado distribuido es clave para mantener agregaciones, ventanas y cálculos históricos. Requiere:

- **Checkpointing confiable** para tolerancia a fallos
- **Escalabilidad del backend de estado** (e.g. RocksDB)
- **Estrategias de limpieza** para evitar crecimiento no controlado

2.3.4. Backpressure

Cuando el sistema no puede procesar datos tan rápido como los recibe, se produce backpressure. Un buen diseño debe:

- Detectar desequilibrios entre fuentes y sinks
- Ajustar dinámicamente el ritmo de procesamiento
- Monitorear métricas como tiempo en cola y uso de buffers

2.3.5. Reprocesamiento

Reprocesar datos históricos por errores o nuevas reglas de negocio requiere:

- **Persistencia de eventos crudos**
- **Mecanismos de replay controlado**
- **Separación de recursos** para evitar degradar el procesamiento en línea

2.3.6. Conformidad Normativa

Cumplir con normativas como GDPR o HIPAA impone requerimientos estrictos:

- **Cifrado en tránsito y en reposo**
- **Gestión de consentimiento y derecho al olvido**
- **Trazabilidad y auditoría de accesos**

Estos desafíos deben abordarse desde la arquitectura, el diseño de datos y las políticas operativas para garantizar un sistema robusto, escalable y confiable.

2.4. Arquitecturas de Referencia

Una arquitectura de referencia representa una plantilla abstracta y probada que encapsula las decisiones arquitectónicas fundamentales de un sistema, mejores prácticas y experiencias acumuladas en un dominio específico.

Esta proporciona un vocabulario común, además de componentes estandarizados y patrones de interacción que sirven como base para el desarrollo de los sistemas concretos.

La arquitectura de referencia no solo define la estructura y comportamiento base del sistema, sino que también establece los principios de diseño, restricciones técnicas y mecanismos de extensibilidad que guiarán estas implementaciones.

2.4.1. Instancias de Arquitectura

Una plataforma que soporte el Monitoreo Remoto de Pacientes, necesita soportar grandes volúmenes de datos analizados como streaming. Además, para poder dar soporte al uso de la plataforma se requiere poder analizar el histórico de dichos datos.

Por último, el tiempo en que el análisis de los datos de streaming es disponibilizado debe ser lo más cercano a tiempo real para poder tomar decisiones a tiempo para la salud del paciente.

Existen tres grandes familias de arquitecturas de referencia que cubren estos tres casos:

- Lambda
- Kappa
- Delta

De entre ellas, se instanciarán y se compararán Kappa y Delta; ya que son evoluciones propuestas sobre Lambda que siguen distintos caminos para alcanzar el mismo objetivo.

Realizar esta instanciación implica seleccionar las tecnologías que se usarán para cumplir las condiciones de la arquitectura de referencia; así como también componentes que si bien no son descritos por la arquitectura de referencia, son necesarios para cumplir con las características de calidad necesarias para el caso de uso propuesto.

Por último, las dos instancias de arquitectura implementadas utilizarán las mismas tecnologías, o tanto como sea posible, de modo que los resultados sean comparables.

2.5. Arquitectura Lambda

2.5.1. Descripción General

La Arquitectura Lambda es un paradigma de procesamiento de datos diseñado para manejar grandes cantidades de información en sistemas de Big Data. Propuesta por Nathan Marz en 2011, esta arquitectura busca abordar las limitaciones de los sistemas de procesamiento batch (batch) y en tiempo real, combinando ambos enfoques para proporcionar una vista completa y actualizada de los datos.

2.5.2. Componentes Principales

La Arquitectura Lambda se compone de tres capas fundamentales:

Batch Layer

- Almacena el conjunto completo de datos históricos.
- Procesa periódicamente volúmenes arbitrarios de datos.
- Genera vistas pre-computadas para consultas eficientes.

Serving Layer

- Almacena las vistas pre-computadas de la capa batch.
- Proporciona acceso de baja latencia a los resultados.

Speed Layer

- Procesa datos en tiempo real.
- Genera vistas de estos datos.
- Mantiene los datos guardados unicamente hasta que la Batch Layer haya hecho el reprocesamiento de los datos historicos.

Vista Lógica

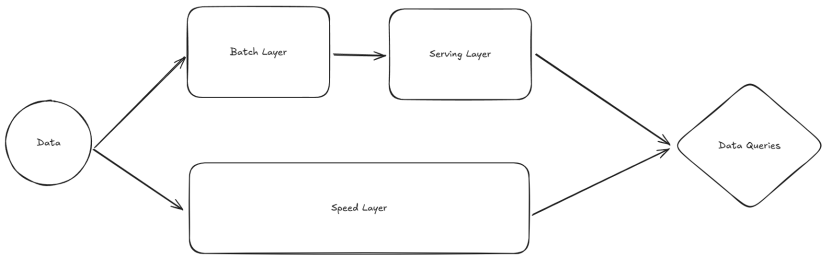

Figura 2.1: Diagrama de la Arquitectura Lambda

2.5.3. Capacidades

- **Procesamiento de datos a gran escala**: Maneja eficientemente volúmenes masivos de datos.
- **Baja latencia**: Proporciona resultados en tiempo real para consultas.
- **Tolerancia a fallos**: Mantiene la integridad de los datos incluso en caso de fallos del sistema.
- **Escalabilidad**: Se adapta fácilmente al crecimiento del volumen de datos.
- **Flexibilidad**: Permite el procesamiento tanto batch como en tiempo real.
- **Consistencia eventual**: Garantiza que los datos eventualmente reflejarán todos los cambios.
- **Reprocesamiento**: En caso de necesitar reprocesar los datos, este proceso es trivial, pues se tiene almacenado el histórico completo.

2.5.4. Desafíos

- **Complejidad**: La implementación y mantenimiento pueden ser complejos debido a la duplicación de lógica en batch layer y speed layer.
- **Latencia**: El procesamiento batch genera latencia debido al tiempo de la actualización de vistas.
- **Costo**: Al utilizar recursos computacionales diferentes entre el procesamiento batch y en stream, esto puede requerir varios nodos computacionales, lo que incrementa los costos.

2.6. Arquitectura Kappa

2.6.1. Descripción General

La Arquitectura Kappa surge en 2014 como respuesta de parte de Jay Kreps a la Arquitectura Lambda. Si bien Lambda puede describirse de forma "sencilla" como una serie de transformaciones y además pone mucho enfasis en la posibilidad y facilidad de reprocesar los datos; tiene la desventaja de obligar a mantener código que debe producir el mismo resultado en dos sistemas distribuidos.

Esto implica que cualquier cambio o mejora que reciba uno debe recibir un tratamiento de reingenieria para que el otro también lo tenga. Y, según argumenta Jay Kreps, la tendencia es a optimizar el código para uno de los motores (incluso si un mismo motor soporta dos modos de trabajo, la semántica que maneja será distinta por lo que terminará siendo una base de código distinta). [10]

La Arquitectura Kappa busca responder la pregunta:

¿Por qué un sistema de procesamiento de Streams no podría incrementar su paralelismo y reprocesar su historia muy rápido?

La intuición detrás de esta arquitectura es la siguiente:

- Usar Kafka o algún otro sistema que permita tener la traza completa de los datos que se quieren reproducir para múltiples suscriptores
- Cuando se quiera hacer reprocesamiento, iniciar una segunda instancia de procesamiento que comience del principio de la historia
- Redirigir el procesamiento de la nueva historia a una tabla auxiliar
- Cuando el segundo procesamiento haya alcanzado al anterior, hacer que empiece a usarse la tabla auxiliar en lugar de la anterior
- Parar el procesamiento anterior y eliminar la tabla anterior

De todas maneras, el resultado del procesamiento o los estados intermedios pueden llegar a ser guardados a su vez por alguna otra herramienta para realizar procesamiento en batch.

2.6.2. Componentes Principales

Stream Store Layer

- Actúa como un registro inmutable de todos los eventos de datos entrantes.
- Permite la reproducción de datos históricos para reprocesamiento cuando se actualiza la lógica de procesamiento.

Stream Processing Layer

- Ingiere datos en tiempo real desde diversas fuentes.
- Procesa estos datos utilizando un sistema de procesamiento de streams.
- Aplica la lógica de negocio y las transformaciones necesarias a los datos entrantes.

Serving Layer

- Almacena los resultados procesados del stream.
- Proporciona acceso de baja latencia a los resultados.

2.6.3. Vista Lógica

Figura 2.2: Diagrama de la Arquitectura Kappa

2.6.4. Capacidades

La Arquitectura Kappa ofrece varias capacidades clave:

- Reduce la complejidad del sistema al unificar el procesamiento batch y streaming
- Mejora la mantenibilidad al no tener que duplicar lógica para los distintos esquemas de procesamiento
- Garantiza la coherencia entre los resultados del streaming y el reprocesamiento.

2.6.5. Desafíos

A pesar de sus ventajas, la Arquitectura Kappa presenta algunos desafíos:

- Requiere un incremento en el uso de recursos muy grande cuando es necesario reprocesar los datos históricos.
- El costo de retención de eventos a largo plazo son enormes y vuelven prohibitiva esta arquitectura sin cambios.
- Es necesario reprocesar toda la historia en caso de que exista la necesidad de borrado de información

2.7. Arquitectura Delta

La Arquitectura Delta surge desde Databricks, al igual que la Arquitectura Kappa, como una respuesta a los desafíos que presenta la arquitectura Lambda.

A diferencia de la suposición que hace Kappa de que todo puede ser tratado como un Stream, Delta por su parte, utiliza el almacenamiento de objetos en la nube como base y agrega por encima tecnología de metadata que otorga la posibilidad de utilizar este sistema de almacenamiento como un canal de mensajes de modo que se puede hacer Streaming sobre él; además de agregar otras capacidades.

Esto permite utilizar un único flujo de datos tanto para análisis en tiempo real como análisis histórico.

2.7.1. Descripción General

Delta surge de la necesidad de procesar datos masivos a bajo costo; intentando aprovechar la estructura existente en las organizaciones que utilizan almacenamiento en la nube.

Los formatos de tabla analítica permiten abstraer el almacenamiento y tratarlo como si fuera una tabla, por lo que se ingestan y luego, mediante el uso de motores de procesamiento se realiza el análisis de dichos datos, que se vuelcan en otras tablas dentro de la misma infrastructura.

Esto permite no tener que distinguir entre batch y streaming, ya que los formatos de tabla analítica proveen mecanismos para detectar los cambios en las tablas y transmitirlos, generando un stream interno que puede ser aprovechado para realizar un análisis incremental.

Por lo general se utiliza un patrón de diseño de datos llamado Medallion, que define tres niveles de calidad de datos:

- Bronze: Donde se almacenan los datos que llegan en crudo
- Silver: Donde se filtran, limpian y enriquecen los datos de Bronze
- Gold: Donde se analizan los datos para generar información valiosa para el negocio

2.7.2. Componentes Principales

Ingestion Layer

- Recibe eventos y los envía al Data Lakehouse Layer
- Su función es más limitada que en la Arquitectura Kappa

Data Lakehouse Layer

- Es una capa montada sobre almacenamiento barato como los servicios de almacenamiento de objetos en la nube
- Los formatos de tabla analítica se montan sobre este almacenamiento
- Recursos de cómputo llamados Table Services pertenecen a esta capa y dan mantenimiento al almacenamiento
- Los motores de procesamiento analizan los datos en varias etapas y las vuelvan nuevamente sobre el almacenamiento
- Generalmente se opta por una sub-arquitectura en niveles, cuyo último nivel son los datos procesados disponibles para el negocio

Catalog Layer

- Provee una capa de governanza permitiendo acceso granular a los datos, auditoría y políticas de retención
- Permite interoperar con terceros, permitiendoles descubrir tablas en base a metadatos
- Ofrece estadísticas de las tablas y herramientas para optimizar las consultas sobre ellas
- Es necesario para acceder a los datos históricos

Serving Layer

- Almacena los resultados procesados del stream por un periodo de tiempo.
- Proporciona acceso de baja latencia a los resultados del procesamiento y al histórico de datos disponibles.

2.7.3. Vista Lógica

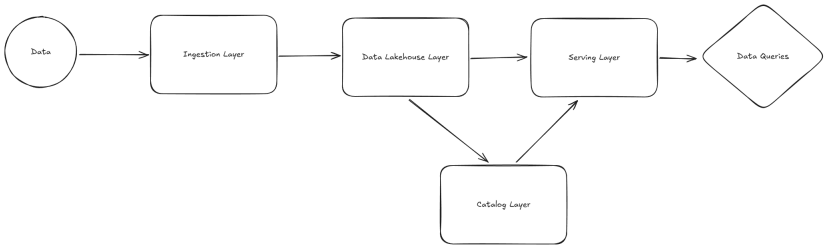

Figura 2.3: Diagrama de la Arquitectura Delta

2.7.4. Capacidades

- Garantiza transacciones para un sistema distribuido
- Reduce los costos de almacenamiento y procesamiento de la información
- Define una fuente de verdad única que puede ser usada por todos los procesos de análisis
- Reduce la cantidad de código que se debe mantener
- Permite agregar nuevas fuentes de datos sin necesidad de cambios en los procesos de análisis

2.7.5. Desafíos

- La latencia es un problema si se necesitan capacidades de análisis en tiempo real
- Se requieren compromisos de latencia y rendimiento por el problema de "Manejo de archivos pequeños"
- Si el Catalog Layer no se construye correctamente no es posible que la arquitectura escale

2.8. Monitoreo Remoto de Pacientes

Los Sistemas de Monitorización Remota de Pacientes (RPM, por sus siglas en inglés) constituyen un paradigma tecnológico de salud en el área de la Telemedicina que permite la adquisición, transmisión y análisis, idealmente en tiempo real, de datos fisiológicos del paciente fuera de los entornos clínicos tradicionales, mediante una red de dispositivos médicos y sensores de dispositivos inteligentes.

Este enfoque contribuye a mejores resultados para los pacientes, disminuye costos para las instituciones de salud y permite dar un acceso más generalizado a los servicios médicos. Es especialmente para el seguimiento de condiciones pre-existente, población anciana y monitoreo luego de intervenciones quirúrgicas.[5]

2.8.1. Monitoreo de Signos Vitales

Frecuencia Respiratoria

- Número de ciclos respiratorios (inspiración/espiración) por minuto
- Valores normales adulto: 12-20 respiraciones/min

Saturación de Oxígeno

- Porcentaje de hemoglobina unida a oxígeno en sangre arterial
- Valores normales: 95-100 %

Presión Sistólica

- Presión máxima ejercida por la sangre sobre las paredes arteriales durante la sístole
- Valores normales: 90-120 mmHg
-

Frecuencia Cardíaca

- Número de contracciones cardíacas por minuto
- Valores normales adulto: 60-100 latidos/min

Temperatura

- Medida del calor corporal
- Valores normales: 36.5-37.5°C

Escala Glasgow

- Escala neurológica que evalúa nivel de consciencia
- Evalúa la apertura ocular, la respuesta verbal y la respuesta motora en distintos rangos
- Valores normales: 15
- Es difícil de automatizar

Nivel de Conciencia

- Sistema simplificado de evaluación del estado de consciencia utilizado en valoración inicial y monitoreo
- Utiliza el sistema APVU: Alerta, Respuesta a estímulos verbales, Respuesta a estímulos dolorosos, Sin respuesta
- Valores normales: 0
- Al ifual que la escala Glasgow es dificil de automatizar

2.8.2. Identificación de Riesgo en Pacientes

En un entorno hospitalario, la monitorización de los signos vitales constituye un pilar fundamental en la evaluación del estado clínico de un paciente. Estos parámetros fisiológicos esenciales incluyen la presión arterial (PA), la saturación de oxígeno en sangre (SpO2), la temperatura corporal (T), la frecuencia cardíaca (FC) y la frecuencia respiratoria (FR), los cuales son registrados sistemáticamente en intervalos de 4 a 6 horas como parte del protocolo estándar de vigilancia para detectar posibles deterioros en la condición del paciente.

En diversos establecimientos sanitarios a nivel global, el personal médico y de enfermería implementa metodologías estandarizadas de evaluación, conocidas como sistemas de alerta temprana (SAT). Estos sistemas utilizan algoritmos validados que asignan puntuaciones específicas a las desviaciones de los rangos normales de los signos vitales, permitiendo la activación de alertas cuando se detectan patrones que indican un deterioro clínico.

Esta práctica sistemática facilita la identificación a tiempo de pacientes en riesgo y permite la intervención terapéuticas a tiempo, contribuyendo significativamente a la reducción de eventos adversos y a la optimización de los resultados clínicos.

Existen diferentes estándares para la detección, muchos dependientes del contexto de la unidad donde se atienda al paciente. [3]

MEWS

MEWS (Modified Early Warning Score) es un sistema de puntuación fisiológica validado para la detección temprana del deterioro clínico en pacientes hospitalizados, que evalúa cinco parámetros vitales fundamentales: frecuencia respiratoria, frecuencia cardíaca, presión arterial sistólica, temperatura y nivel de consciencia.

Cada parámetro recibe una puntuación de 0 a 3 según la gravedad de su alteración, siendo 0 el valor normal y 3 el más patológico; La suma total de estos valores genera una puntuación que oscila entre 0 y 14, categorizando el riesgo del paciente en bajo (0–1), medio (2–3), alto (4–5) o crítico ($\geq$6), lo que determina la frecuencia de monitorización necesaria y las intervenciones requeridas, desde una vigilancia rutinaria cada 8-12 horas en puntuaciones bajas hasta la activación inmediata del equipo de respuesta rápida y posible traslado a UCI en puntuaciones críticas.

NEWS2

El NEWS2 (National Early Warning Score 2) es una versión mejorada y actualizada del sistema de alerta temprana, adoptado como estándar por el Servicio Nacional de Salud del Reino Unido, que evalúa siete parámetros fisiológicos: frecuencia respiratoria (3-0 puntos), saturación de oxígeno (con dos escalas distintas según el riesgo de insuficiencia respiratoria hipercápnica, 3-0 puntos), uso de oxígeno suplementario (2 puntos si requiere), temperatura (3-0 puntos), presión arterial sistólica (3-0 puntos), frecuencia cardíaca (3-0 puntos) y nivel de consciencia utilizando la escala ACVPU.

La puntuación total varía de 0 a 20, estratificando el riesgo en bajo (0-4), medio (5-6), alto (7 o más, o cualquier parámetro individual con puntuación de 3) y determinando la respuesta clínica necesaria, desde monitorización estándar hasta evaluación urgente por equipo de cuidados críticos, representando una mejora significativa respecto al MEWS al incluir la saturación de oxígeno y la confusión como nuevo nivel de consciencia.

SOFA

El SOFA (Sequential Organ Failure Assessment Score) es un sistema de puntuación diseñado para evaluación diaria (cada 24 horas) de la disfunción/fallo multiorgánico en unidades de cuidados intensivos, evaluando seis sistemas orgánicos: respiratorio (mediante la relación PaO2/FiO2 evaluada con cada gasometría, 0-4 puntos), cardiovascular (mediante presión arterial media y requerimiento de vasopresores monitorizados continuamente, 0-4 puntos), hepático (mediante bilirrubina sérica medida diariamente, 0-4 puntos), coagulación (mediante recuento plaquetario diario, 0-4 puntos), renal (mediante creatinina sérica diaria o gasto urinario horario, 0-4 puntos) y neurológico (mediante la escala de Glasgow evaluada cada 4 horas o con cambios clínicos, 0-4 puntos).

Cada sistema recibe una puntuación de 0 (normal) a 4 (máxima disfunción), con una puntuación total que varía de 0 a 24 puntos, calculándose cada 24 horas o antes si hay deterioro clínico significativo, siendo especialmente relevante el cambio en la puntuación a lo largo del tiempo.

qSOFA

El qSOFA (quick Sequential Organ Failure Assessment) es una versión simplificada del SOFA, diseñada para la identificación rápida de pacientes con sospecha de sepsis y alto riesgo de mortalidad fuera de la UCI, evaluando únicamente tres parámetros clínicos que se pueden medir de manera inmediata a pie de cama, sin necesidad de pruebas de laboratorio: alteración del estado mental (escala de Glasgow $\leq$13 puntos, 1 punto), frecuencia respiratoria elevada ($\geq$22 respiraciones/minuto, 1 punto) y presión arterial sistólica baja ($\leq$100 mmHg, 1 punto).

La puntuación total varía de 0 a 3 puntos, donde una puntuación $\geq$2 indica alto riesgo de mortalidad y la necesidad de evaluación más exhaustiva, monitorización estrecha y consideración de traslado a un nivel superior de cuidados; el qSOFA debe reevaluarse con cada valoración del paciente o ante cualquier cambio en su estado clínico, típicamente cada 1-2 horas en pacientes inestables o con sospecha de sepsis, siendo una herramienta especialmente útil en servicios de urgencias, plantas de hospitalización y entornos extrahospitalarios.

2.8.3. Desafíos

Los sensores empleados en la monitorización remota permiten un monitoreo continuo de los parámetros fisiológicos correspondientes, proporcionando una frecuencia de recolección de datos significativamente superior a los intervalos tradicionales establecidos en entornos convencional.

Sin embargo, la implementación de estos sistemas de monitorización remota presenta diversos desafíos técnicos y operativos que requieren consideración.

Entre las principales se encuentran:

- Movilidad del usuario que puede afectar la calidad de la señal
- Uso correcto del dispositivo
- Variabilidad en las condiciones ambientales
- Fallos intermitentes de los sensores
- Agotamiento de la batería de los dispositivos
- Pérdida de conectividad en la transmisión de datos
- Falta de datos por dificultad de medición como en el caso de la escala Glasgow o niveles de conciencia

Esto genera la necesidad de procesar los datos adecuadamente y desarrollar técnicas para la limpieza y validación de datos. A su vez, se deben implementar estrategias para el manejo de datos faltantes e incompletos. [3]

Capítulo 3

Metodología

3.1. Criterios de Evaluación

Para evaluar y comparar las arquitecturas Kappa y Delta en el contexto del monitoreo remoto de pacientes, se considerarán creiterios unificados y medibles. Estos servirán como base para una evaluación objetiva de las arquitecturas Kappa y Delta en el contexto del monitoreo remoto de pacientes, permitiendo tomar criterios fundamentados para la elección de una u otra según el escenario.

3.1.1. Latencia y Rendimiento

Se implementarán mediciones a través de puntos de instrumentación estratégicos a lo largo de los componentes de la arquitectura. Estos puntos de medición incluirían timestamps en los mensajes, métricas de procesamiento en los componentes intermedios, y el tiempo de escritura/lectura en la capa final.

Las métricas serán tomadas utilizando Prometheus y mostradas en un tablero de Grafana.

Las métricas a considerar serán:

- Tiempo de ingesta de histórico
- Latencia de procesamiento

3.1.2. Manejo de Datos Históricos

Se medirán especificamente la efectividad con la que el sistema permite consultar datos históricos y reprocesar toda la historia. Las métricas serán tomadas utilizando Prometheus y mostradas en un tablero de Grafana.

- Tiempo de reprocesamiento de la historia completa
- Uso de recursos en reprocesamiento de la historia completa

Se omitirá la implementación de un cambio en el modelo de procesamiento que implique reprocesar la historia completa, pero se dará una propuesta de como hacerlo y se evaluará las implicancias de su puesta en producción manteniendo los dos sistemas funcionando e integrandolos eventualmente.

3.1.3. Costos Operativos

La implementación del sistema en contenedores, permitirá un monitoreo del uso de los recursos del sistema. Además, se plantearán cálculos, utilizando las calculadoras de costo que proveen los sistemas de nube, que permitan dimensionar el costo operativo de estos sistemas.

Las métricas serán tomadas utilizando Prometheus y mostradas en un tablero de Grafana.

Las métricas a considerar serán:

- Uso de memoria RAM
- Uso de CPU
- Uso de disco
- Uso de red

3.2. Stack de Tecnologías a Utilizar

Con el objetivo de comparar objetivamente las arquitecturas Kappa y Delta, se definió un stack tecnológico común que permite mantener condiciones similares en términos de despliegue, monitoreo y operación.

Estas tecnologías fueron seleccionado considerando aspectos como compatibilidad, madurez del ecosistema, integración con otras herramientas, facilidad de operación, disponibilidad de documentación y eficiencia de recursos.

Sistema de Mensajería - Apache Kafka

Apache Kafka fue elegido como el punto de entrada del sistema por ser el estándar de facto en mensajería distribuida en arquitecturas de datos modernas. Su arquitectura distribuida permite alta disponibilidad, tolerancia a fallos, retención configurable y orden garantizado en particiones.

Kafka desacopla la ingesta del procesamiento, permitiendo escalabilidad horizontal y resiliencia ante picos de carga.

Se hicieron pruebas con Apache Pulsar, que promete ser el sucesor de Kafka, pero su adopción en la industria es aún limitada y su ecosistema de herramientas no es tan maduro. A su vez, su documentación presenta inconsistencias que hacen un desafío su integración en escenarios poco convencionales.

Motor de Procesamiento de Flujos - Apache Flink

Flink fue seleccionado por su baja latencia, manejo avanzado de estado y compatibilidad directa con fuentes como Kafka y formatos de tablas analíticas como Apache Paimon.

Su modelo de programación basado en flujos de datos y su integración con catálogos lo hacen ideal tanto para arquitecturas Kappa como Delta.

Además, permite operaciones con garantías *exactly-once* y es capaz de reprocesar eventos desde el principio del log si fuera necesario.

Aunque Apache Spark Structured Streaming podría haber sido considerado, Flink presenta menor latencia y mejor soporte para semánticas de estado complejas.

Almacenamiento de Objetos - MinIO

MinIO fue elegido como la solución de almacenamiento distribuido compatible con la API de S3. Su facilidad de despliegue, rendimiento eficiente y compatibilidad con entornos locales lo posicionan como una alternativa ideal a soluciones propietarias en la nube, permitiendo independencia de proveedor y bajo costo operativo.

Formato de Tabla Analítica - Apache Paimon

Apache Paimon fue seleccionado como formato de tabla debido a su diseño nativo para flujos de datos. Su integración con Flink es directa y permite operaciones de escritura eficientes en escenarios de actualización frecuente. A pesar de tener una comunidad pequeña, su especialización para el procesamiento incremental en tiempo real fue clave para esta elección.

Otras alternativas como Apache Iceberg o Hudi tienen comunidades más grandes y mejor documentación, pero presentan mayores desafíos en integración directa con Flink o no funcionan también para el caso de procesamiento de streaming.

Motor Analítico - Apache Doris

Doris combina procesamiento paralelo masivo (MPP) con consultas OLAP en tiempo real. Su soporte para tablas híbridas y la posibilidad de realizar consultas federadas sin necesidad de duplicar los datos lo convierte en una excelente opción para analítica en tiempo real e histórica. También la integración con Paimon mediante un catálogo lo posiciona como una solución eficiente y simple de operar.

Aunque Druid o Pinot fueron considerados, Doris ofrece capacidades que estas otras tecnologías no tienen. En particular, las consultas federadas.

Orquestación de Contenedores - Docker Compose

El uso de Docker Compose permite una gestión simple, portable y reproducible del entorno experimental. Facilita el aislamiento de componentes, el control de versiones de imágenes y la replicabilidad del experimento.

Esta elección responde a la necesidad de contar con un entorno controlado sin la complejidad que presentan otras herramientas.

Monitoreo y Visualización: Prometheus + Grafana

Prometheus permite recolección y análisis eficiente de métricas, mientras que Grafana proporciona una interfaz amigable y personalizable para visualizar el estado del sistema y comparar el rendimiento entre arquitecturas.

Esta combinación es ampliamente adoptada en entornos productivos, y su integración nativa con sistemas de contenedores y servicios distribuidos la vuelve especialmente útil para este caso.

Además, se aprovecha la capacidad de Grafana para crear paneles de control personalizados para también visualizar los resultados de las consultas analíticas.

3.3. Despliegue

Se realizará un despliegue de ambas arquitecturas manteniendo la misma cantidad de nodos de cada tecnología. Para esto se utilizarán contenedores con la tecnología Docker Compose dentro de una única máquina virtual linux. El sistema donde se desplegará el sistema cuenta con 16 núcleos, 64 GB de RAM y 2 TB de disco duro de estado sólido. El sistema operativo utilizado es Ubuntu 22.04 LTS.

3.3.1. Componentes

Se desplegarán los siguientes componentes:

Componente	**Cantidad**
Broker Apache Kafka	3
Zookeeper	3
Apache Flink Job Manager	1
Apache Flink Task Manager	4
Apache Doris Frontend	1
Apache Doris Backend	3
MinIO	1
Grafana	1
Prometheus	1

Cuadro 3.1: Cantidad de componentes desplegados

El objetivo de esta configuración es proveer de un entorno de alta disponibilidad que a su vez permita escalar horizontalmente y se adecúe al ambiente de pruebas.

3.3.2. Configuración de los componentes

Se utilizarán configuraciones por defecto para cada uno de los componentes desplegados. Sin embargo, se realizán los cambios necesarios para asegurar el correcto funcionamiento de los mismos.

Se deberá asegurar la mayor cantidad de recursos del sistema para cada uno de los componentes, dándole mayor importancia a los componentes que se encargan de la ingestión y procesamiento de datos.

De esta forma, se deberá asegurar el procesamiento ininterrumpido de los datos.

3.4. Mediciones

Se realizarán mediciones de rendimiento y latencia de ambas arquitecturas.

Para esto se tomarán 5 muestras de cada medición y se calculará el promedio y la desviación estándar. Las mediciones se realizarán en el mismo entorno de despliegue y con la misma carga de trabajo. Se utilizará un conjunto de datos sintético que corresponden a 1 año de datos de 32 pacientes, lo que implica alrededor de 7 GB de datos.

Todas las mediciones se harán sobre un entorno recien instalado para asegurar que no haya interferencias entre mediciones. Se desarrollará un script en Python que se encargará de enviar las mediciones a cada uno de los sistemas.

El script se encargará de enviar los datos a través de un productor de Kafka y lo hará tan rápido como sea posible, lo que permitirá medir el impacto que tiene el despliege del sistema en el entorno donde se ejecuta.

3.5. Conjunto de Datos

El conjunto de datos utilizado para evaluar el sistema será de datos sintéticos. Se desarrollará un script en Python que genere datos sintéticos para los sensores de signos vitales.

Estos datos serán generados de acuerdo a la naturaleza de la medición de cada sensor y se interconectarán para simular un flujo de datos continuo. El script generará datos para tres tipos de pacientes: Sanos, Sanos pero que se deterioran con el tiempo, y enfermos pero estables. De esta manera, se podrá evaluar el sistema en diferentes escenarios.

Existen dos razones principales para el uso de datos sintéticos:

- Los conjuntos de datos de sensores disponibles son heterogéneos y muy pequeños; lo que de todos modos implicaría generar datos sintéticos.
- No es necesario contar con datos que muestren tendencias reales, sino que se comporten como lo harían en realidad para demostrar los atributos de calidad del sistema.

Para esto, se definirá un flujo de datos para cada sensor que será interconectado pero definiendo comportamientos especificos que tengan que ver con la naturaleza de su medición. Los signos vitales tomados para esto son:

- Frecuencia respiratoria
- Presión sistólica
- Frecuencia cardíaca
- Temperatura
- Saturación de oxígeno
- Nivel de conciencia en escala APVU

Capítulo 4

Desarrollo

4.1. Monitoreo Remoto de Pacientes

4.1.1. Introducción

Antecedentes

El Monitoreo Remoto de Pacientes ha emergido como una tecnología para mejorar la atención médica moderna, permitiendo la vigilancia continua del paciente fuera de los entornos clínicos tradicionales. Sin embargo, estos sistemas enfrentan desafíos en el mantenimiento de la calidad consistente de datos y la integración de mediciones de múltiples dispositivos con diferentes niveles de confiabilidad y tasas de muestreo.

Planteamiento del Problema

Los sistemas tradicionales de monitoreo de signos vitales frecuentemente enfrentan dificultades con:

- Temporización inconsistente de mediciones entre diferentes signos vitales
- Variación en la confiabilidad y precisión de los dispositivos
- Integración de múltiples fuentes de datos para el mismo signo vital
- Mantenimiento de la validez clínica con datos incompletos

Solución Propuesta

Se propone abordar estos desafíos mediante:

- Fusión de datos multi-dispositivo
- Puntaje de calidad del dato
- Puntaje de frescura del dato
- Capacidad de degradación gradual

4.1.2. Sistema de Puntuación NEWS2

Descripción General de NEWS2

El National Early Warning Score 2 (NEWS2) es una herramienta de evaluación estandarizada utilizada para detectar el deterioro clínico. Evalúa seis parámetros fisiológicos:

- Frecuencia respiratoria
- Saturación de oxígeno
- Presión arterial sistólica
- Frecuencia cardíaca
- Nivel de consciencia
- Temperatura

Desafíos de Implementación Tradicional

NEWS2 fue originalmente diseñado para mediciones manuales periódicas en entornos clínicos. Su adaptación para monitoreo remoto continuo presenta varios desafíos:

- Diferentes frecuencias de medición para diferentes parámetros
- Calidad y confiabilidad variable de las mediciones
- Necesidad de actualizaciones de puntuación en tiempo real
- Manejo de datos faltantes o degradados

4.1.3. gdNEWS2

Concepto y Fundamentos

Se propone aumentar el sistema NEWS2 con el concepto de degradación gradual (graceful degradation) de modo que la puntuación aún sea útil incluso cuando la medición de alguno de los datos de signos vitales no se encuentre presente o no se confíe del todo en la calidad del mismo.

Cada parámetro de NEWS2 tiene un puntaje de calidad y de frescura asociado, que se utilizarán para definir el nivel de confianza que se le tiene a dicho parámetro. De esta forma, se puede calcular una puntuación de alerta temprana incluso cuando no se tienen todos los datos disponibles y brindarle transparencia al profesional de la salud para que tome las decisiones correspondientes en cuanto al tratamiento del paciente.

4.1.4. Formato de Datos de Mediciones Brutas

Los datos serán recibidos en formato JSON, con un esquema común para todos los tipos de mediciones. El esquema general es el siguiente:

```
{
    "measurement_type": "RESPIRATORY_RATE" | "HEART_RATE" |
       "OXYGEN_SATURATION" | "BLOOD_PRESSURE_SYSTOLIC" | "
       TEMPERATURE" | "CONSCIOUSNESS",
    "measurement_timestamp": "datetime",
    "device_id": "string",
    "raw_value": "number",
    "battery": "number",
    "signal_strength": "number"
}
```

Listing 4.1: JSON example

4.1.5. Algoritmo de Puntuación de Calidad

Un nuevo algoritmo de puntuación de calidad debería basarse en la experiencia clínica y la evidencia científica. Para el caso de estudio presentado, se propone un algoritmo sencillo a fin de ilustrar su potencial y mantener limitado el enfoque de este trabajo.

Se tomarán los identificadores de los dispositivos, que luego serán clasificados en 3 grupos cada uno con un peso asociado:

Clasificación de Dispositivos	**Peso Asociado**
Dispositivos de calidad médica	1.0
Dispositivos de calidad premium	0.7
Dispositivos de calidad de consumo	0.4

Cuadro 4.1: Clasificación de dispositivos y sus pesos correspondientes

Además, se tomará en cuenta la señal de batería y la intensidad de la señal, que serán clasificados en 3 grupos cada uno con un peso asociado:

Nivel de Batería	**Peso Asociado**
Batería a más del 80 %	1.0
Batería entre 80 % y 50 %	0.7
Batería entre 50 % y 20 %	0.6
Batería a menos de 20 %	0.4

Cuadro 4.2: Clasificación de niveles de batería y sus pesos correspondientes

Por último, se tomará en cuenta la intensidad de la señal:

Intensidad de Señal	**Peso Asociado**
Valor de señal de más de 0.8	1.0
Valor de señal entre 0.8 y 0.6	0.8
Valor de señal entre 0.5 y 0.6	0.6
Valor de señal menor a 0.5	0.4

Cuadro 4.3: Clasificación de intensidad de señal y sus pesos correspondientes

De esta manera, el cálculo de la puntuación de calidad se realizará de la siguiente manera:

$$\text{Quality Score} = 0.7\times\text{Device Quality}+0.2\times\text{Battery Quality}+0.1\times\text{Signal Quality} \tag{4.1}$$

En este momento, los valores tanto de los pesos como de los parametros son arbitrarios y se espera que en caso de encontrar útil este acercamiento, futuras iteraciones ajusten estos parámetros o utilicen un criterio diferente para su cálculo.

4.1.6. Algoritmo de Puntuación de Frescura

Para medir la frescura de los datos, se propone un enfoque simple que considera el tiempo transcurrido desde la última medición y el tiempo que le tomó a la medición actual llegar a ser procesada.

Tiempo desde medición hasta procesamiento:

Tiempo de Medición hasta Procesamiento	**Peso Asociado**
Menos de una hora	1.0
Entre una y seis horas	0.9
Entre seis y doce horas	0.7
Entre doce y veinticuatro horas	0.5
Entre veinticuatro y cuarenta y ocho horas	0.3
Otros	0.2

Cuadro 4.4: Clasificación de tiempo desde medición hasta procesamiento

Tiempos entre mediciones:

Tiempo entre Mediciones	**Peso Asociado**
Menos de cuatro horas	1.0
Entre cuatro y ocho horas	0.8
Entre ocho y doce horas	0.6
Entre doce y veinticuatro horas	0.4
Más de veinticuatro horas	0.2

Cuadro 4.5: Clasificación de tiempo entre mediciones

De esta manera, se propone el siguiente algoritmo de puntuación de frescura:

$$\text{Freshness Score} = 0.5 \times \text{Time Since Last Measurement} + 0.5 \times \text{Time Since Measurement} \tag{4.2}$$

4.1.7. Algoritmo de Puntuación de Degradación

Este algoritmo se encargará de calcular la puntuación de degradación de la puntuación NEWS2 en caso de que no se tengan todos los datos disponibles. Simboliza la confianza que se le tiene a la puntuación NEWS2 en base a la calidad y frescura de los datos.

Se propone utilizar la siguiente fórmula para calcular la puntuación de degradación:

$$\text{Degradation Score} = 0.7 \times \text{Quality Score} + 0.3 \times \text{Freshness Score} \tag{4.3}$$

Se calcula este puntaje para cada uno de los parámetros de NEWS2 y se promedia para obtener la puntuación de degradación final.

4.1.8. Algoritmo de Puntuación de NEWS2

El algoritmo de puntuación NEWS2 se basa en la suma de los puntajes de cada uno de los parámetros,

$$\text{NEWS2 Score} = \sum_{i=1}^{n} \text{Parameter Score}_i \tag{4.4}$$

donde n es la cantidad de parámetros que se tienen disponibles. En caso de que no se tenga un parámetro disponible, se asumirá una puntuación de cero para ese parámetro en su lugar. De esta manera, se puede calcular la puntuación NEWS2 incluso cuando no se tienen todos los datos disponibles.

Por otro lado, se propone utilizar la puntuación de degradación para dar contexto sobre la puntuación NEWS2 final. Así, se puede explicar que tan confiable es la puntuación NEWS2 calculada en base a los datos disponibles.

4.2. Implementación

Se implementó le especificación definida en el capítulo anterior de modo tal que para ambas arquitecturas los detalles de implementación sean lo más similares posible. Para esto, se utilizó como lenguaje de procesamiento Flink SQL, que permite desarrollar los trabajos de procesamiento utilizando un lenguaje agnóstico a las plataformas subyascentes.

4.2.1. Pipeline de Procesamiento

El pipeline de procesamiento se encarga de recibir los datos en formato JSON, realizar el procesamiento de los mismos y devolver la puntuación NEWS2 calculada.

Esto se desarrolló de la siguiente manera:

- Recepción de datos en formato JSON mediante un topico de Kafka
- Enriquecimiento de los datos con los puntajes de calidad y frescura
- Enrutamiento de los datos para su procesamiento particular según el signo vital
- Cálculo de la puntuación NEWS2 para cada una de las Componentes
- Unión y agrupación según una ventana de tiempo
- Calculo de valores de agregación de los puntajes de NEWS2 y de degradación

Debido a una limitante en el hardware de procesamiento, se simplificó el calculo de frescura de los datos para no tener en cuenta los anteriores. Esto provocaba que el sistema se quedara sin memoria RAM y no pudiera procesar los datos para ambas arquitecturas. Por lo que se optó por no tener en cuenta los datos anteriores y solo calcular la frescura de los datos con las medidas de tiempo propias de cada registro.

A continuación se presenta un diagrama de flujo del pipeline de procesamiento:

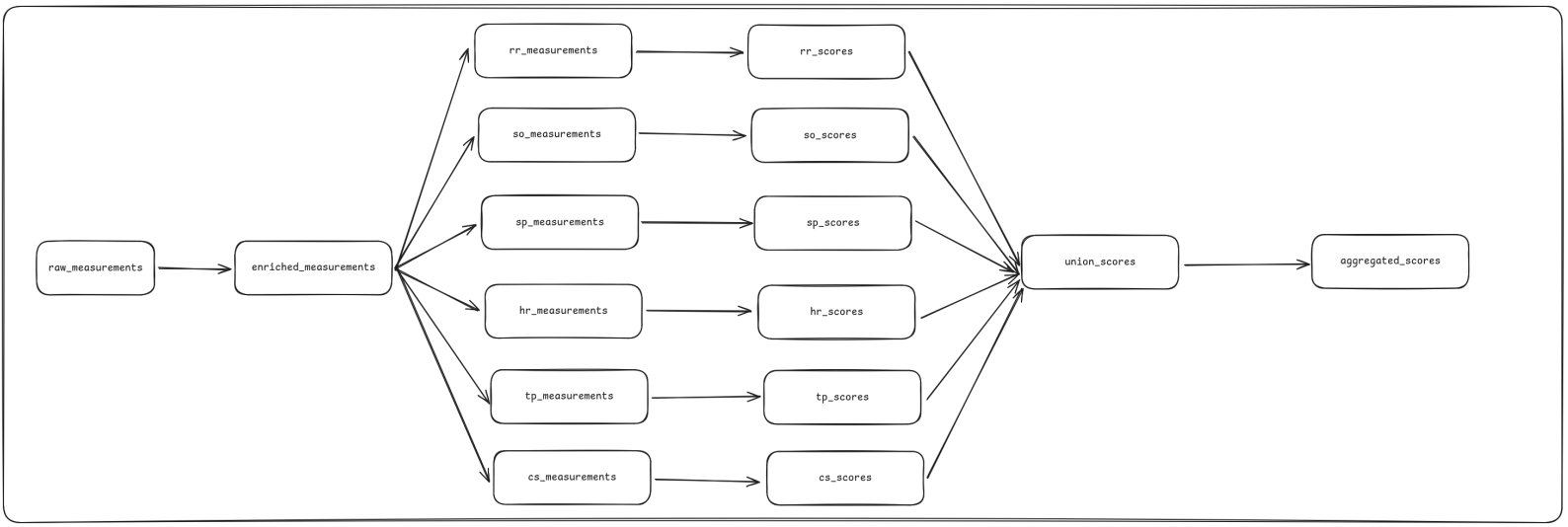

Figura 4.1: Diagrama de flujo del pipeline de procesamiento

4.2.2. Despliegue de Componentes

El despliegue de los componentes se realizó mediante el uso de **Docker Compose**, y se midió según las métricas expuestas por esta herramienta. Por otro lado, para el calculo de costos, se asume un despliegue de alta disponibilidad en la nube de AWS basado en el siguiente diagrama:

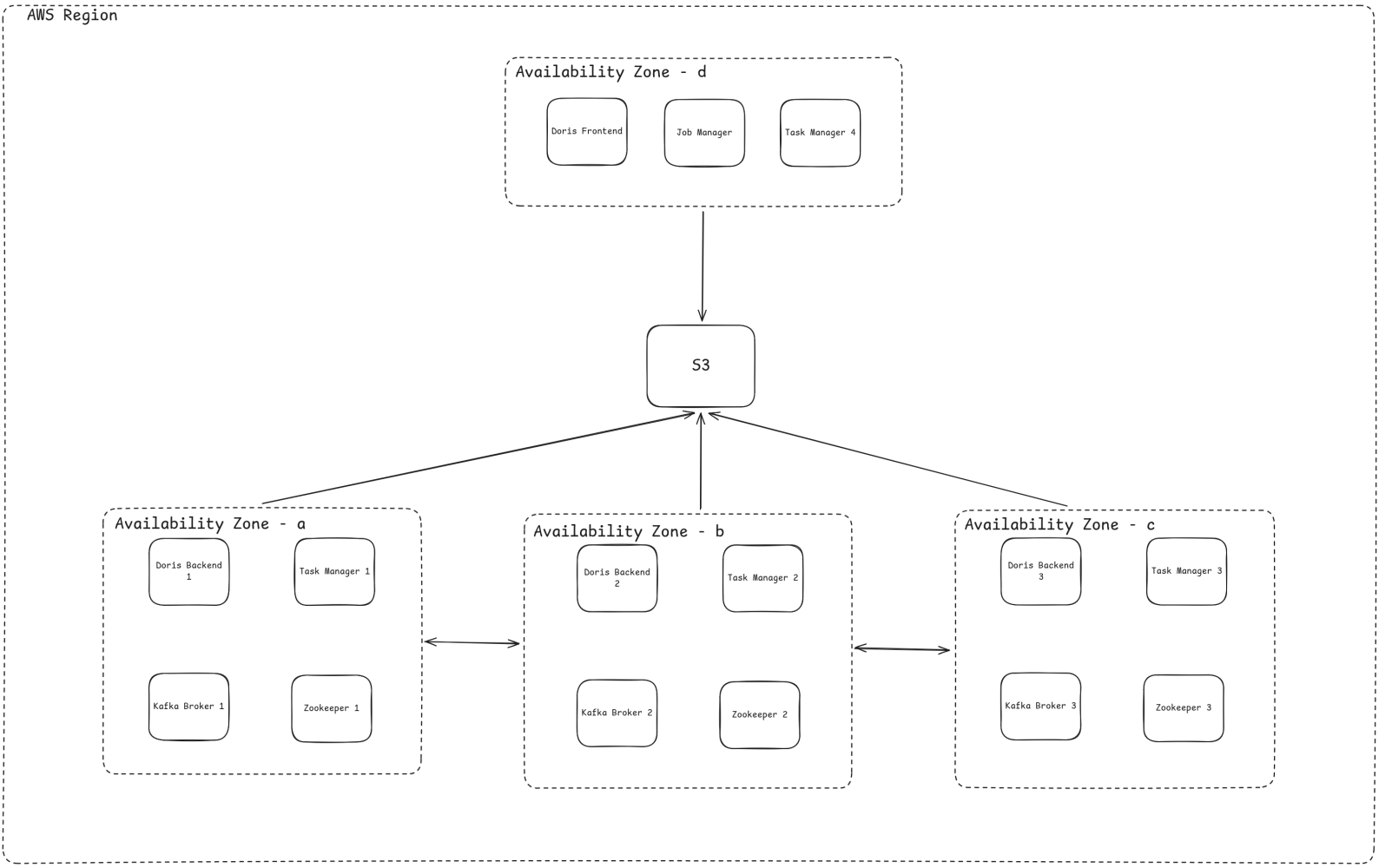

Figura 4.2: Diagrama de despliegue de componentes

La razón de este despliegue es la alta disponibilidad y la tolerancia a fallos, por lo que se despliega en una única región y se dividen los servicios en zonas de disponibilidad para asegurar que si una de ellas falla, el sistema siga funcionando lo mejor posible.

Tres de las zonas de disponibilidad (a, b y c) son idempotentes en cuanto a su funcionamiento, cada una cuenta con un nodo de Kafka, un nodo Zookeeper, un nodo de procesamiento de Flink y un nodo de backend de Doris. La cuarta zona de disponibilidad (d) tiene el nodo de frontend de Doris y el nodo de gestión de Flink; así como también un nodo extra de procesamiento de Flink. No se incluye MinIO en este despliegue porque se utiliza S3 nativo, que se define en una región y esta igualmente comunicado con todas las zonas de disponibilidad.

A su vez, estarían idealmente desplegados mediante un orquestador de contenedores como Kubernetes, utilizando algún servicio como Elastic Kubernetes Serice (EKS).

4.2.3. Generación de Datos Sintéticos

Se utilizaron datos sintéticos para realizar las pruebas de carga y estrés de ambas arquitecturas. Para la generación de los datos sintéticos se utilizó un script en Python que permite definir perfiles de pacientes con diferentes características, así como también el rango de tiempo para el que se genera la información.

Se generaron datos sintéticos para 32 pacientes a lo largo de un año, con un total de 110.122.654 registros; que implica un archivo CSV de aproximadamente 7GB.

El proceso de generación de datos es el siguiente:

- Se genera un archivo CSV con los datos sintéticos ordenados por signo vital.
- Se separa este archivo CSV en un archivo por paciente y se ordena cada uno.
- Se vuelve a juntar la información de cada paciente de forma ordenada en un único archivo CSV.
- Se envía cada línea del CSV a los nodos Kafka de la cada una de las arquitecturas para su ingestión

Se debieron definir estos pasos debido al tamaño del archivo CSV, ya que al ser tan grande no se puede cargar en memoria. Por otro lado, para simular demoras, se agregó una columna de delay al archivo inicial que luego fue tomada en cuenta al momento de ordenar las filas. El archivo CSV final resultante tiene el siguiente formato:

```
device_id,measurement_type,timestamp,raw_value,battery,signal
DEVICE_002_P0001,RESPIRATORY_RATE,-19.48,16.27,99.59,0.48
DEVICE_002_P0001,TEMPERATURE,-3.46,36.43,99.67,0.72
DEVICE_002_P0001,BLOOD_PRESSURE_SYSTOLIC,1.22,119.03,99.71,0.75
DEVICE_002_P0001,HEART_RATE,3.0,70.31,99.8,0.71
DEVICE_002_P0001,HEART_RATE,173.27,73.08,100.0,0.48
DEVICE_002_P0001,RESPIRATORY_RATE,247.35,16.36,99.32,0.71
```

El archivo de configuración usado tiene la siguiente forma:

```
{
    "time_range": "1_year",
    "patients": [
        {"patient_id": "P0001", "category": "HEALTHY"},
        ...
        {"patient_id": "P0005", "category": "ILL_STABLE"},
        ...
        {"patient_id": "P0009", "category": "HEALTHY_DETERIORATING"},
        ...
    ],
    "devices": [
        {
            "device_id": "MEDICAL_RR",
            "patient_ids": [
                ...
            ],
            "vitals": {
                "RESPIRATORY_RATE": {"measurement_rate": 60}
            },
            "battery": {"initial": 100, "drain_rate": 0.05},
            "signal_strength": {"base": 0.9, "variation": 0.1}
        },
        ...
        {
            "device_id": "CONSUMER_SMARTWATCH_002",
            "patient_ids": [
                ...
            ],
            "vitals": {
                "RESPIRATORY_RATE": {"measurement_rate": 240},
                "BLOOD_PRESSURE_SYSTOLIC": {"measurement_rate": 300},
                "HEART_RATE": {"measurement_rate": 180},
                "TEMPERATURE": {"measurement_rate": 600}
            },
            "battery": {"initial": 100, "drain_rate": 0.02},
            "signal_strength": {"base": 0.6, "variation": 0.2}
        }
    ]
}
```

4.2.4. Visualizaciones

Se implementó un tablero de monitoreo utilizando Grafana, que permite visualizar la latencia, el throughtput y el uso de hardware en tiempo real. Grafana se conecta con Prometheus que es el sistema de monitoreo utilizado para recolectar las métricas de los componentes de la arquitectura.

Figura 4.3: Tablero de Monitoreo

También se implementó en Grafana un tablero de datos por pacientes, que permite visualizar la evolución de los signos vitales de cada paciente a lo largo del tiempo.

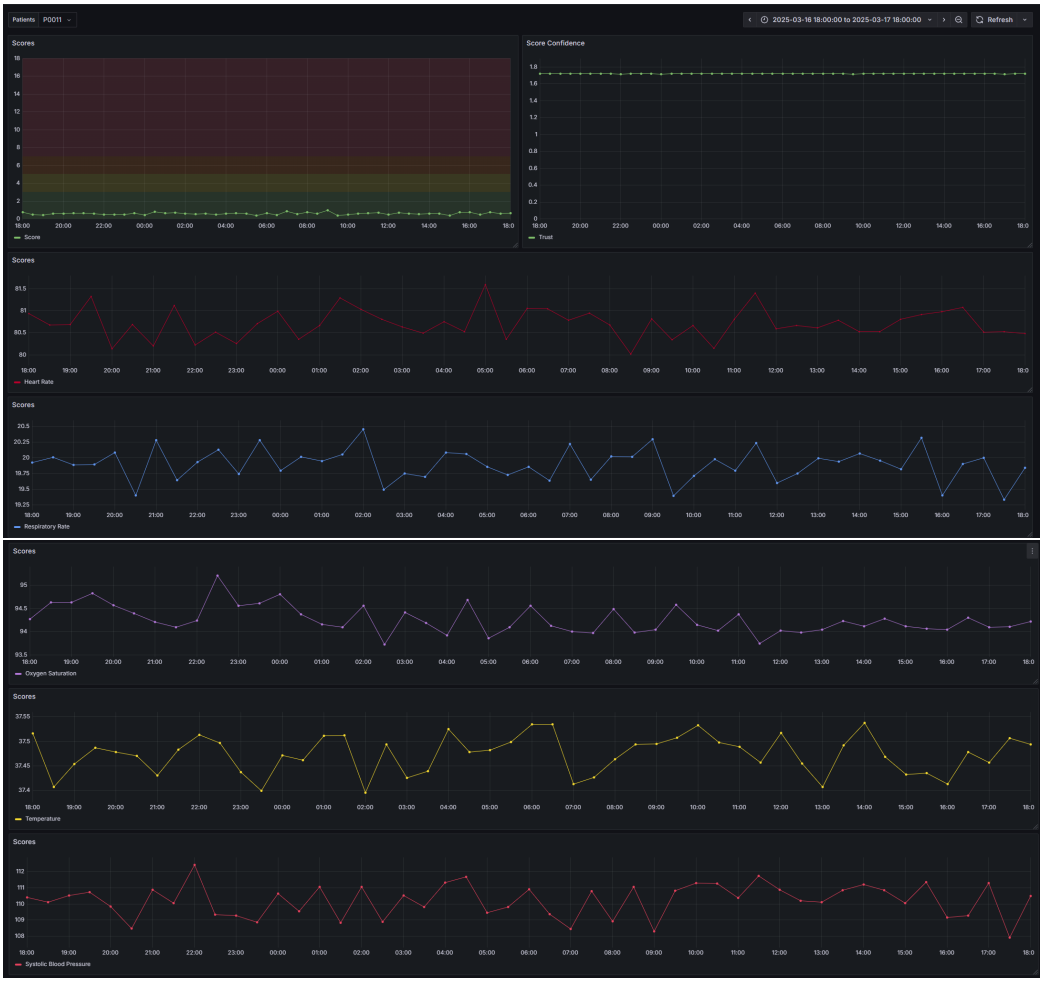

Figura 4.4: Tablero de Visualizaciones

4.2.5. Limitaciones en la Implementación

Se definieron algunas limitaciones en la implementación de la arquitectura Kappa y Delta, que se detallan a continuación:

- No se implementará una solución de Governanza de datos
- No se implementará una solución de Calidad de Datos
- No se implementará una solución de Linaje de Datos
- No se implementará una solución de Gestión de Plataforma
- No se implementaron medidas de cifrado y anonimización de datos

Estas limitaciones se deben a que el objetivo de este trabajo es evaluar las arquitecturas Kappa y Delta, e incluir esas soluciones haría que el trabajo se extienda mucho más allá de lo esperado.

4.3. Arquitectura Kappa

4.3.1. Principios de Diseño

La principal característica de esta arquitectura es su fuerte uso de un registro de eventos inmutable y ordenado cronológicamente que actúa como única fuente de verdad sobre los datos ingresados al sistema.

De esta manera, se logra unificar el procesamiento de datos en batch y streaming tratándolos como un flujo continuo de eventos, eliminando la dualidad de código y reduciendo la complejidad operativa.

El procesamiento de estos datos se realiza mediante motores de procesamiento de eventos que leen este registro, aplican transformaciones determinísticas y generan resultados derivados que pueden recomputarse en cualquier momento desde el inicio del log.

Este principio de reproducibilidad permite regenerar el estado completo del sistema cuando cambian los requisitos o algoritmos de procesamiento, sin necesidad de mantener rutas de código separadas.

Las vistas materializadas son otro principio fundamental, donde los resultados procesados se almacenan en sistemas optimizados para consultas, proporcionando acceso eficiente al estado actual sin necesidad de reprocesar todo el historial de eventos.

4.3.2. Stack Tecnológico

Para la capa de ingesta y transporte de datos, la Arquitectura Kappa implementa **Apache Kafka** como componente central, funcionando no solo como sistema de mensajería sino como la fuente única de verdad y almacén principal de eventos. En Kappa se configura Kafka con períodos de retención extendidos, aprovechando la capacidad de compactación de logs para mantener el historial completo de eventos mientras se optimiza el espacio de almacenamiento. Esto se logra agregando la capacidad de almacenamiento en capas, mediante la cual se pueden mantener los eventos en Object Storage (utilizando **MinIO**), cuando pasa un tiempo definido de mantención en almacenamiento local.

El procesamiento de datos se realiza mediante **Apache Flink**, se despliega en un cluster con un nodo Job Manager y cuatro nodos Task Manager; de forma de distribuir la carga de trabajo lo mejor posible. En este caso, se define como punto de entrada un tópico de Kafka, para procesar los datos en tiempo real y enviarlos a un nuevo topico y continuar con el procesamiento más adelante en la arquitectura.

En el último paso, se guarda el resultado del procesamiento en **Apache Doris**, un motor de análisis de datos distribuido que permite realizar consultas SQL en tiempo real sobre grandes volúmenes de datos con una interfaz basada en MySQL. Este componente permite escalar de forma diferente el acceso a los datos del procesamiento, siendo desplegado como un nodo frontend y tres nodos backend. Estos comparten el trabajo de procesamiento de consultas y almacenamiento de datos, mientras que el frontend se encarga de la distribución de las mismas.

4.3.3. Vista de Componentes

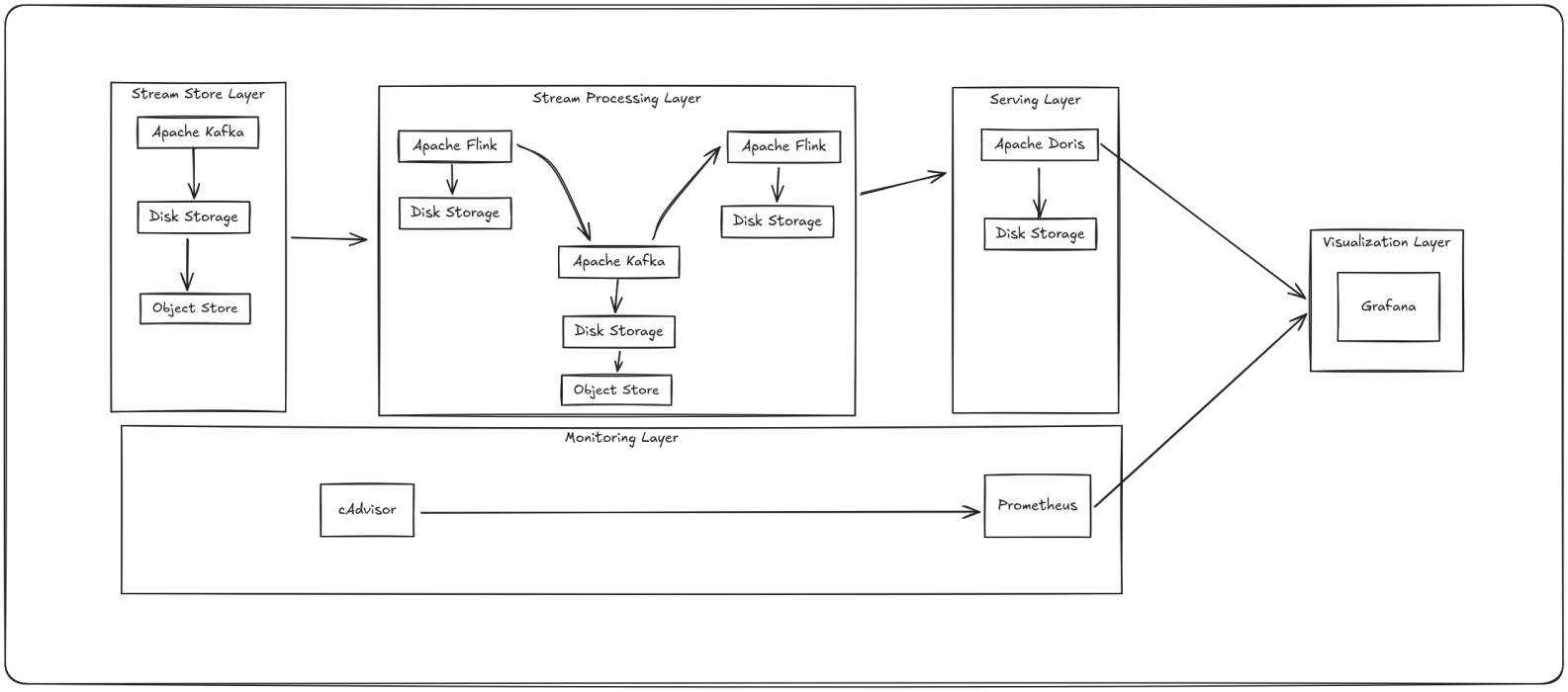

Figura 4.5: Diagrama de la Arquitectura Kappa

4.3.4. Flujo de Procesamiento

El siguiente es un ejemplo de uno de los trabajos de procesamiento de datos desarrollados:

```
SET 'execution.runtime-mode' = 'streaming';
SET 'execution.checkpointing.mode' = 'EXACTLY_ONCE';
SET 'table.local-time-zone' = 'UTC';
SET 'execution.checkpointing.interval' = '60000';
SET 'execution.checkpointing.timeout' = '30000';
SET 'state.backend' = 'hashmap';
SET 'table.exec.state.ttl' = '300000';
SET 'parallelism.default' = '4';
```

```
-- Raw measurements table with original timestamps and device metrics
CREATE TABLE raw_measurements (
    measurement_timestamp TIMESTAMP(3),
    measurement_type STRING,
    raw_value STRING,
    device_id STRING,
    battery DOUBLE,
    signal_strength DOUBLE,
    ingestion_timestamp TIMESTAMP(3) METADATA FROM 'timestamp' VIRTUAL,
    WATERMARK FOR measurement_timestamp AS measurement_timestamp - INTERVAL '10' SECONDS
) WITH (
    'topic' = 'raw.measurements',
    'connector' = 'kafka',
    'properties.bootstrap.servers' = 'kafka-1:19091,kafka-2:19092,kafka-3:19093',
    'format' = 'json',
    'json.timestamp-format.standard' = 'ISO-8601',
    'scan.startup.mode' = 'latest-offset'
);
```

```
CREATE TABLE enriched_measurements (
    measurement_type STRING,
    `value` DOUBLE,
    device_id STRING,
    patient_id STRING,

    -- Weights
    quality_weight DOUBLE,
    freshness_weight DOUBLE,

    -- Timestamps
    measurement_timestamp TIMESTAMP(3),
    ingestion_timestamp TIMESTAMP(3),
    enrichment_timestamp TIMESTAMP(3) METADATA FROM 'timestamp'
        VIRTUAL,
    WATERMARK FOR measurement_timestamp AS measurement_timestamp -
        INTERVAL '10' SECONDS
) WITH (
    'topic' = 'enriched.measurements',
    'connector' = 'kafka',
    'properties.bootstrap.servers' = 'kafka-1:19091,kafka-2:19092,
        kafka-3:19093',
    'format' = 'json',
    'json.timestamp-format.standard' = 'ISO-8601',
    'scan.startup.mode' = 'latest-offset'
);
```

```
    -- Insert with quality and freshness calculations
    INSERT INTO enriched_measurements
    SELECT
        measurement_type,
        CAST(raw_value AS DOUBLE) AS `value`,
        device_id,
        REGEXP_EXTRACT(device_id, '.*_(P\d+)$', 1) AS patient_id,

        -- Quality components
        CAST((
            CASE
                WHEN device_id LIKE 'MEDICAL%' THEN 1.0
                WHEN device_id LIKE 'PREMIUM%' THEN 0.7
                ELSE 0.4
            END * 0.7 +
            CASE
                WHEN battery >= 80 THEN 1.0
                WHEN battery >= 50 THEN 0.8
                WHEN battery >= 20 THEN 0.6
                ELSE 0.4
            END * 0.2 +
            CASE
                WHEN signal_strength >= 0.8 THEN 1.0
                WHEN signal_strength >= 0.6 THEN 0.8
                WHEN signal_strength >= 0.4 THEN 0.6
                ELSE 0.4
            END * 0.1
        ) AS DECIMAL(7,2)) AS quality_weight,

        -- Combined freshness calculation
        CASE
            WHEN TIMESTAMPDIFF(HOUR, measurement_timestamp,
                ingestion_timestamp) <= 1 THEN 1.0
            WHEN TIMESTAMPDIFF(HOUR, measurement_timestamp,
                ingestion_timestamp) <= 6 THEN 0.9
            WHEN TIMESTAMPDIFF(HOUR, measurement_timestamp,
                ingestion_timestamp) <= 12 THEN 0.7
            WHEN TIMESTAMPDIFF(HOUR, measurement_timestamp,
                ingestion_timestamp) <= 24 THEN 0.5
            WHEN TIMESTAMPDIFF(HOUR, measurement_timestamp,
                ingestion_timestamp) <= 48 THEN 0.3
            ELSE 0.2
        END AS freshness_weight,

        -- Timestamps
        measurement_timestamp,
        ingestion_timestamp
    FROM raw_measurements;
```

Como se puede ver, FLink SQL permite tratar a los tópicos de Kafka como tablas, pudiendose así leer y escribir sobre ellos. Esto permite realizar un procesamiento de datos en tiempo real, enriquecerlos y enviarlos a otro tópico de Kafka para su posterior procesamiento.

Para esta arquitectura se utilizaron dos conectores diferentes de Kafka. El primero, visto en los ejemplos, permite leer y escribir pero no modificar. Por otro lado, para las agregaciones, se utilizó **upsert-kafka** que agrega la semántica de actualización y borrado de mensajes, que es muy útil para cuando se necesita un procesamiento incremental de la información, como es el caso de las agregaciones. Aunque cabe destacar que la potencia de Flink permite que se pueda hacer esto incluso para otros destinos de datos como se verá más adelante para Paimon. Todo esto sin cambiar el código del trabajo de procesamiento.

```
CREATE TABLE scores (
    patient_id STRING,
    window_start TIMESTAMP(3),
    window_end TIMESTAMP(3),

    respiratory_rate_value DOUBLE,
    oxygen_saturation_value DOUBLE,
    blood_pressure_value DOUBLE,
    heart_rate_value DOUBLE,
    temperature_value DOUBLE,
    consciousness_value DOUBLE,

    respiratory_rate_score DOUBLE,
    oxygen_saturation_score DOUBLE,
    blood_pressure_score DOUBLE,
    heart_rate_score DOUBLE,
    temperature_score DOUBLE,
    consciousness_score DOUBLE,

    respiratory_rate_trust_score DOUBLE,
    oxygen_saturation_trust_score DOUBLE,
    blood_pressure_trust_score DOUBLE,
    heart_rate_trust_score DOUBLE,
    temperature_trust_score DOUBLE,
    consciousness_trust_score DOUBLE,

    measurement_timestamp TIMESTAMP(3),
    ingestion_timestamp TIMESTAMP(3),
    enrichment_timestamp TIMESTAMP(3),
    routing_timestamp TIMESTAMP(3),
    scoring_timestamp TIMESTAMP(3),
    union_timestamp TIMESTAMP(3),
    WATERMARK FOR union_timestamp AS union_timestamp - INTERVAL '10'
        SECONDS,
    PRIMARY KEY (patient_id, window_start) NOT ENFORCED
) WITH (
    'connector' = 'upsert-kafka',
    'topic' = 'scores',
    'properties.bootstrap.servers' = 'kafka-1:19091,kafka-2:19092,
        kafka-3:19093',
    'key.format' = 'json',
    'value.format' = 'json'
);
```

```
INSERT INTO scores
SELECT * FROM (
    WITH unions as (
        ...
    )
    SELECT
        patient_id,
        window_start,
        MAX(window_end) as window_end,

        MAX(CASE WHEN measurement_type = 'RESPIRATORY_RATE' THEN `
            value` END) as respiratory_rate_value,
        MAX(CASE WHEN measurement_type = 'OXYGEN_SATURATION' THEN `
            value` END) as oxygen_saturation_value,
        MAX(CASE WHEN measurement_type = 'BLOOD_PRESSURE_SYSTOLIC'
            THEN `value` END) as blood_pressure_value,
        MAX(CASE WHEN measurement_type = 'HEART_RATE' THEN `value` END
            ) as heart_rate_value,
        MAX(CASE WHEN measurement_type = 'TEMPERATURE' THEN `value`
            END) as temperature_value,
        MAX(CASE WHEN measurement_type = 'CONSCIOUSNESS' THEN `value`
            END) as consciousness_value,

        ...

        MIN(measurement_timestamp) AS measurement_timestamp,
        MIN(ingestion_timestamp) AS ingestion_timestamp,
        MIN(enrichment_timestamp) AS enrichment_timestamp,
        MIN(routing_timestamp) AS routing_timestamp,
        MIN(scoring_timestamp) AS scoring_timestamp,
        CURRENT_TIMESTAMP as union_timestamp
    FROM TABLE(
        TUMBLE(
            TABLE unions,
            DESCRIPTOR(measurement_timestamp),
            INTERVAL '1' MINUTES
        )
    ) AS unions
    GROUP BY patient_id, window_start
) as t;
```

Por último, se guarda el resultado del procesamiento en **Apache Doris** directamente desde Flink. Para esto, es necesario que la tabla en Doris haya sido creada previamente y además definir un nombre con el que llamarla en el trabajo de procesamiento. Luego, se puede insertar los datos y Flink y Doris acordarán la forma de hacerlo. Según las pruebas realizadas, esto se hace en batches. El tiempo, entre que se terminó de procesar y fue insertado en Doris no fue posible de medir ya que no se encontraró una forma de definir la fecha de inserción real.

```
CREATE TABLE doris_gdnews2_scores (
    patient_id STRING,
    window_start TIMESTAMP(3),
    window_end TIMESTAMP(3),

    -- AVG Raw measurements
    ...

    -- Raw NEWS2 scores
    ...
    news2_score DOUBLE,

    -- Trust gdNEWS2 scores
    ...

    news2_trust_score DOUBLE,

    -- Timestamps
    measurement_timestamp TIMESTAMP(3),
    ingestion_timestamp TIMESTAMP(3),
    enrichment_timestamp TIMESTAMP(3),
    routing_timestamp TIMESTAMP(3),
    scoring_timestamp TIMESTAMP(3),

    flink_timestamp TIMESTAMP(3),
    aggregation_timestamp TIMESTAMP(3),
    PRIMARY KEY (patient_id, window_start) NOT ENFORCED
) WITH (
    'connector' = 'doris',
    'fenodes' = '172.20.4.2:8030',
    'table.identifier' = 'kappa.gdnews2_scores',
    'username' = 'kappa',
    'password' = 'kappa',
    'sink.label-prefix' = 'doris_sink_gdnews2',
    'sink.properties.format' = 'json',
    'sink.properties.timezone' = 'UTC'
);
```

```
INSERT INTO doris_gdnews2_scores
SELECT *
FROM gdnews2_scores;
```

4.4. Arquitectura Delta

4.4.1. Principios de Diseño

Todo el almacenamiento de datos a largo plazo se realiza en un formato de tabla abierta, que combina archivos en formato Parquet con un registro de transacciones. Esto garantiza propiedades ACID y permite operaciones confiables en entornos distribuidos.

Al utilizar object storage, se logra una separación clara entre los recursos de procesamiento y los de almacenamiento, lo que permite escalarlos de manera independiente.

Además, múltiples clientes pueden acceder a los mismos datos de forma simultánea sin interferencias, incluso utilizando herramientas diferentes, siempre que sean compatibles con el formato subyacente.

El formato Parquet, al ser columnar, permite ejecutar consultas SQL de manera eficiente y es compatible con la mayoría de los motores de análisis modernos.

El procesamiento de datos se gestiona como un flujo continuo de eventos, donde el motor de procesamiento utiliza el log de transacciones como fuente de verdad para mantener el estado de los datos. Esto permite unificar el procesamiento batch y streaming en una misma arquitectura. Como efecto secundario de esto, todos los datos son guardados para cada etapa del flujo de procesamiento. Esto significa que están disponibles para su fácil consumo en caso de que se quieran analizar; pero como contraparte, consumen más espacio de almacenamiento ya que se almacena potencialmente varias veces el mismo dato (aunque enriquecido).

Además, el sistema se encarga automáticamente de optimizaciones como la compactación de archivos pequeños y la gestión eficiente de metadatos, asegurando un rendimiento óptimo sin intervención manual.

Por último, al estar basado en estándares abiertos, el sistema evita el vendor lock-in y permite integración con diversas herramientas de BI, machine learning y ETL.

4.4.2. Stack Tecnológico

Para la capa de ingesta y transporte de datos se implementó **Apache Kafka**, un sistema de mensajería distribuido que proporciona alta durabilidad, replicación y garantía en el orden de los eventos. Kafka actúa como el punto de entrada de la arquitectura, permitiendo desacoplar la ingesta de datos del procesamiento y asegurando una capa de buffer que absorbe picos de tráfico mientras mantiene los datos disponibles para su consumo. En este caso, Kafka se despliega en un cluster de tres nodos, con un factor de replicación de tres y un factor de partición de tres. Por otro lado, se definió un tiempo de retención de mensajes acotado, en este caso de 7 días, para evitar la acumulación de datos pero a su vez asegurar la disponibilidad de los mismos para su procesamiento.

El procesamiento de datos se realiza mediante **Apache Flink**, se despliega en un cluster con un nodo Job Manager y cinco nodos Task Manager; de forma de distribuir la carga de trabajo lo mejor posible. En este caso, se define como punto de entrada un tópico de Kafka, para luego continuar procesando los datos, no utilizando Kafka sino aprovechando las capacidades de **Apache Paimon**. Este adopta un enfoque log-structured para las escrituras, lo que lo hace especialmente eficiente para cargas de trabajo de streaming con alta frecuencia de actualizaciones.

Este permite tratar una tabla de datos como un flujo continuo de eventos, funcionando de forma efectiva como una cola de mensajes, pero con la ventaja de tener los datos materializados en un almacenamiento persistente y barato.

Por último, este formato de almacenamiento permite ser leido por **Apache Doris**, un motor de análisis de datos distribuido que permite realizar consultas SQL en tiempo real sobre grandes volúmenes de datos con una interfaz basada en MySQL.

Para esto, tanto como para el uso de Flink, se necesita definir un catálogo de tablas. Normalmente, esto podría hacerse utilizando Apache Hive, pero se optó por simplificar el sistema tanto como sea posible, y dado que no se necesitaba interactuar con un sistema existente; por lo que el catálogo se almacenó en Object Storage.

Esto último se logró mediante el uso de **MinIO**, un servidor de almacenamiento de objetos de código abierto que permite almacenar datos de forma segura y eficiente, y que además es compatible con el protocolo S3 de Amazon Web Services.

Esta combinación tecnológica permite implementar efectivamente los principios de la Arquitectura Delta, donde los datos fluyen desde las fuentes a través de Kafka, son procesados por Flink, almacenados en diferentes capas mediante Paimon sobre MinIO, y finalmente consultados a través de Doris, manteniendo en todo momento las propiedades ACID y permitiendo el procesamiento continuo así como análisis retrospectivos sobre datos históricos.

4.4.3. Vista de Componentes

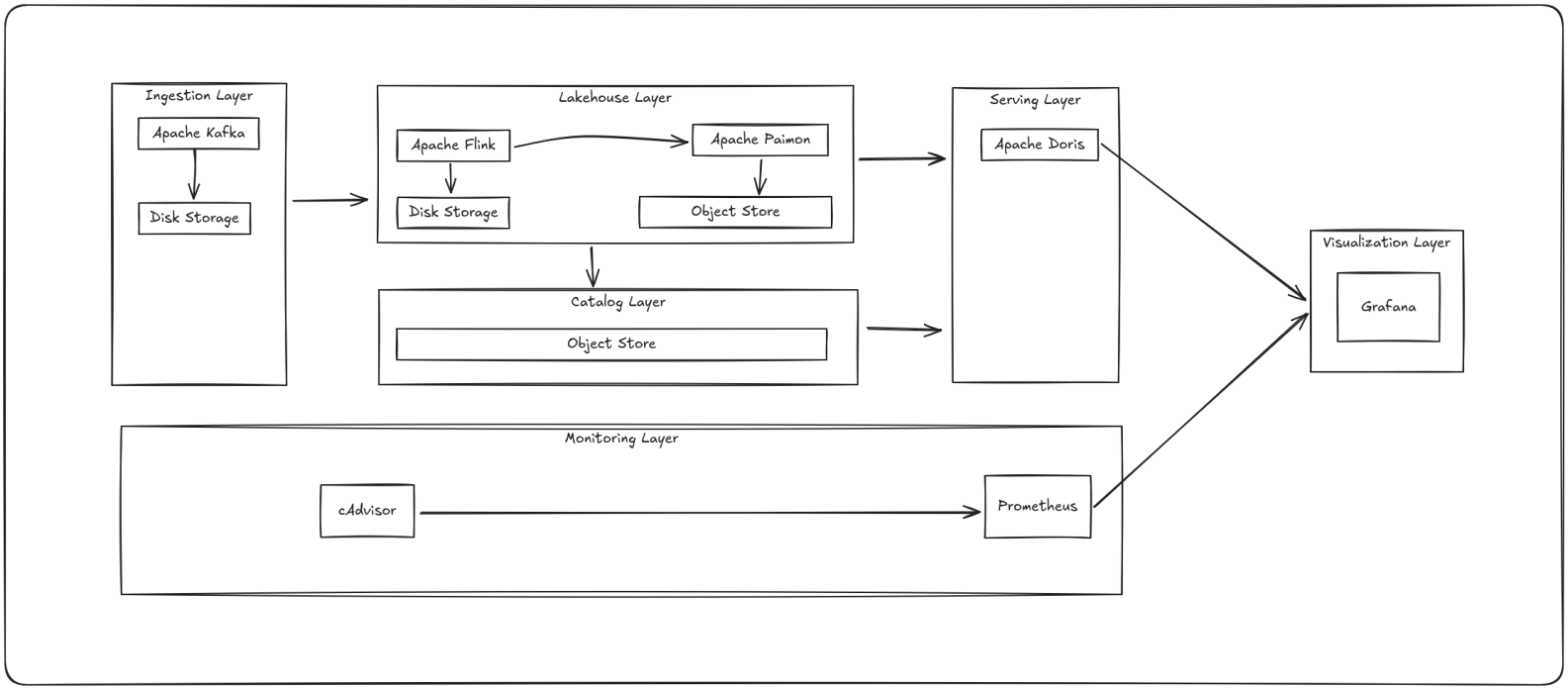

Figura 4.6: Diagrama de la Arquitectura Delta

4.4.4. Flujo de Procesamiento

El siguiente es un ejemplo de uno de los trabajos de procesamiento de datos desarrollados:

```
SET 'execution.runtime-mode' = 'streaming';
SET 'execution.checkpointing.mode' = 'EXACTLY_ONCE';
SET 'table.local-time-zone' = 'UTC';
SET 'execution.checkpointing.interval' = '60000';
SET 'execution.checkpointing.timeout' = '30000';
SET 'state.backend' = 'hashmap';
SET 'table.exec.state.ttl' = '300000';
SET 'parallelism.default' = '2';

CREATE CATALOG paimon WITH (
    'type' = 'paimon',
    'warehouse' = 's3://datalake/paimon',
    's3.endpoint' = 'http://minio:9000',
    's3.access-key' = 'minioadmin',
    's3.secret-key' = 'minioadmin',
    's3.path.style.access' = 'true',
    'location-in-properties' = 'true'
);
```

```
CREATE TABLE paimon.delta.raw_measurements (
    measurement_timestamp TIMESTAMP(3),
    measurement_type STRING,
    raw_value STRING,
    device_id STRING,
    battery DOUBLE,
    signal_strength DOUBLE,
    ingestion_timestamp TIMESTAMP(3),
    WATERMARK FOR measurement_timestamp AS measurement_timestamp -
        INTERVAL '10' SECONDS
);
```

```
CREATE TABLE default_catalog.default_database.raw_measurements (
    measurement_timestamp TIMESTAMP(3),
    measurement_type STRING,
    raw_value STRING,
    device_id STRING,
    battery DOUBLE,
    signal_strength DOUBLE,
    ingestion_timestamp TIMESTAMP(3) METADATA FROM 'timestamp' VIRTUAL,
    WATERMARK FOR measurement_timestamp AS measurement_timestamp - INTERVAL '10' SECONDS
) WITH (
    'topic' = 'raw.measurements',
    'connector' = 'kafka',
    'properties.bootstrap.servers' = 'kafka-1:19091,kafka-2:19092,kafka-3:19093',
    'format' = 'json',
    'json.timestamp-format.standard' = 'ISO-8601',
    'scan.startup.mode' = 'latest-offset'
);
```

```
INSERT INTO paimon.delta.raw_measurements
SELECT * FROM default_catalog.default_database.raw_measurements;
```

Este código muestra algunas de las características principales del uso de Flink SQL en conjunto con Paimon. Las primeras tres reglas son las estándares para un procesamiento de streaming con manejo de mensajes en tiempo real y que además utilice una hora estándar para tener sincronizadas las marcas de tiempo de todos los sistemas que intervienen.

Luego se define el 'execution.checkpointing.interval' que es de los parámetros más importantes para el procesamiento de streaming, ya que define cada cuanto tiempo se guardan los estados intermedios de los datos procesados. Para el caso de Paimon particularmente, marca cada cuanto se impactan los datos procesados en el almacenamiento, por lo que define que tan pronto estarán disponibles estos para su consumo.

Esto afecta directamente a la latencia, por lo que es algo que se tiene que balancear fuertemente con los recursos del sistema para evitar retrasos en el procesamiento.

Algo importante a destacar es que si bien en este caso se define el catalogo en cada uno de los scripts, esto no debería ser necesario pues Flink permite definir mediante configuración sus catálogos disponibles. Sin embargo, no fue posible configurarlo de esta manera, por lo que se optó por definirlo en cada uno de los scripts.

Un último detalle a destacar, es que se define el paralelismo en 2 de modo tal que se pueda aprovechar al máximo las particiones del tópico de Kafka. Lo ideal sería definirlo en 3, pero esto no es posible por una limitante de hardware en cuanto a memoria en el nodo que ejecuta el trabajo de procesamiento.

Esta primer inserción de datos que se encarga de recibir los datos en formato JSON desde el tópico de Kafka y almacenarlos en el formato de Paimon es una de las dos grandes diferencias de esta arquitectura respecto a la anterior. La segunda diferencia es que en este caso no es necesaria la inserción en Doris, ya que esta es sólo una herramienta que se usa para acceder a los datos pero no los almacena. Esto hace que el flujo de procesamiento sea mas simple y liviano al no haber un tercer componente que tenga que procesar información.

4.5. Decisiones Técnicas de Arquitectura

En esta sección se detallan las principales decisiones de diseño arquitectónico tomadas durante la implementación del sistema, con foco en los aspectos de alta disponibilidad, interoperabilidad, simplicidad operativa y escalabilidad futura.

4.5.1. Cluster de Kafka y ZooKeeper

Se configuró un clúster de Kafka con 3 brokers y 3 nodos de ZooKeeper. Esta arquitectura permite:

- Alta disponibilidad y tolerancia a fallos de nodos individuales
- Replicación de particiones con `replication.factor = 3`
- Tolerancia a la pérdida de hasta un nodo en cada capa (brokers o ZooKeeper)

Esta configuración es apropiada para entornos productivos y garantiza durabilidad y disponibilidad en la capa de mensajería.

4.5.2. Despliegue de Apache Doris

Se implementaron 3 nodos Backend (BE), permitiendo la distribución de datos y procesamiento paralelo. Sin embargo, sólo se desplegó una instancia de Frontend (FE), lo que representa un punto único de falla (*Single Point of Failure*). En un entorno de producción, se recomienda utilizar múltiples nodos FE en configuración maestro-seguidor para garantizar disponibilidad continua del servicio de consultas y acceso al catálogo.

4.5.3. Catálogo de Datos

Se optó por utilizar un catálogo basado en sistema de archivos (object storage) para el manejo de tablas analíticas, sin integración con Hive Metastore. Esta decisión simplificó la infraestructura y redujo la complejidad operativa. Si bien esta elección limita la interoperabilidad entre motores de procesamiento, resulta adecuada para soluciones encapsuladas donde no se requiere acceso simultáneo desde múltiples tecnologías.

4.5.4. Coordinación de Procesamiento

Se utilizó una única instancia de JobManager de Apache Flink. Esta configuración es suficiente para entornos controlados, pero representa una limitación en cuanto a tolerancia a fallos. En escenarios reales, se recomienda implementar Flink en modo de alta disponibilidad, utilizando múltiples JobManagers (activo y pasivos), un backend compartido para almacenamiento de estado (por ejemplo, S3), y un coordinador de liderazgo como ZooKeeper.

4.5.5. Justificación General

Las decisiones tomadas responden a un enfoque de prototipo funcional, priorizando la validación de las arquitecturas Kappa y Delta en condiciones controladas. Se buscó un equilibrio entre fidelidad técnica y simplicidad de despliegue, permitiendo reproducibilidad y flexibilidad sin incurrir en la complejidad de un entorno productivo completo.

Capítulo 5

Resultados

5.1. Expectativas Iniciales

Inicialmente, se esperaba que la arquitectura Delta proporcionara una mayor flexibilidad y escalabilidad en comparación con la arquitectura Kappa. Sobre todo porque a nivel publicitario, en los últimos tiempos la industria de ingeniería de datos se ha encargado de promover la arquitectura Delta, en particular los Data Lakehouse, como la solución definitiva para el procesamiento de datos en tiempo real y batch.

Por lo que a priori, se esperaba que Delta fuera más eficiente en términos de rendimiento y costos, y en simplicidad de implementación y mantenimiento.

Por otro lado, dado que Delta utiliza de forma extensiva el Object Storage, se esperaba una pérdida de rendimiento, y en particular más latencia, que su contraparte. Sin embargo, no se esperaba una diferencia demasiado grande ya que muchas veces se propone utilizar un Data Lakehouse como un sistema de almacenamiento de datos en tiempo real.

Por último, dado el apogeo de los Data Lakehouse se suponía una facilidad en el ensamblado y funcionamiento conjunto de los distintos componentes de la arquitectura, asumiendo que hubieran soluciones estandarizadas y abiertas que permitieran la integración de los distintos componentes de forma sencilla y rápida.

5.2. Comparación Técnica

Desde el punto de vista técnico se eligió realizar una comparación basandose en el común denominador tecnológico para ambas arquitecturas. Esto implica que se realizó una optimización profunda para aprovechar al máximo las capacidades de ambas. Sin embargo, se pudieron ver las características más notables en los ecosistemas tecnológicos en los que se basan sus implementaciones.

Un caso interesante es Kafka. Kappa surge del mismísimo creador de Kafka y por lo tanto, aprovecha al máximo las capacidades de baja latencia de esta tecnología. Aún años después de introducida, es la tecnología de mensajería y streaming más popular; habiendo cosechado una enorme comunidad. Esto permitió que para ambas arquitecturas se pudiera implementar un cluster altamente distribuido, con las mejores prácticas posibles.

Es de destacar que inicialmente se intentó utilizar Apache Pulsar ya que es otra plataforma que cumple con los mismos casos de uso que Kafka y promete mayor escalabilidad. Sin embargo, no existe suficiente documentación para llegar rápidamente a la misma calidad con la que se llegó al usar Kafka. Otra desventaja es en su uso en conjunto con Flink ya que los conectores existentes no funcionaban correctamente. Es importante mencionar también, que la comunidad de Kafka es tremendamente activa y en el tiempo que se desarrolló esta tesis mejoraron los protocolos de comunicación entre nodos por lo que Apache Zookeeper dejó de ser necesario en las últimas versiones.

También Flink fue un caso problemático ya que si bien promete unas muy buenas prestaciones para el caso de uso de streaming, su comunidad es muchísimo más pequeña que la de la alternativa más clara (Apache Spark) lo que implicó mucho tiempo de investigación inicial y poco de profundización en la misma. Teniendo en cuenta esto, a menos que el caso de uso específico requiera un procesamiento de streaming puro y tiempos de latencia del nivel de milisegundos, se recomienda utilizar Spark ya que es más fácil de usar y tiene una comunidad mucho más activa.

Doris por su parte, es una tecnología relativamente nueva. Sumado a esto, la mayoría de la documentación está en chino, lo que dificultó mucho su implementación. Sin embargo, las capacidades que ofrece son muy interesantes y prometedoras. Parece tener una comunidad bastante activa y en crecimiento, que apuesta por la adopción y evolución de esta tecnología. Sin embargo, no se pudo encontrar una comunidad de usuarios en inglés que permita una rápida adopción y resolución de problemas. Por lo que tampoco se pudo utilizar las últimas versiones de la misma y por ende se perdió la posibilidad de poner a prueba las últimas mejoras que se han ido introduciendo.

A nivel tecnológico Apache Paimon, fue un hallazgo inesperado pero afortunado ya que es una herramienta que funciona nativamente con Flink (y de hecho nace como un subproyecto de esa comunidad). Esto permitió que si bien la comunidad es pequeña y también en su mayoría en chino, fuera fácilmente integrable para la arquitectura Delta. También su caso de uso más importante era exactamente el que se buscaba en este caso en un formato de tabla analítica.

De no haberlo encontrado, se hubiera tenido que utilizar Apache Iceberg, que es la alternativa más popular y que está tomando más relevancia en el mundo de la ingenería de datos. De todas maneras, en base a las pruebas realizadas inicialmente, Apache Hudi hubiera sido la mejor opción por su enfoque en streaming de datos. Su mayor problema es que tiene una comunidad mucho más pequeña que Iceberg (aunque muchisimo más grande que Paimon), que no es tan fácil de integrar con Flink y que por su parte también presenta complejidades en su despliegue ya que no es solamente un formato de tabla analítica sino que se proveen de forma separada servicios de Lakehouse que se deben integrar.

El catálogo de metadatos fue otro punto importante a tener en cuenta. En este caso, al no necesitar integración con otras herramientas, se optó por un catálogo en sistema de archivos. Sin embargo, normalmente sería preferible por cuestiones de integración exponer un servicio de metadatos como Apache Hive o AWS Glue. Este es otro aspecto muy emergentes, sobre todo en el mundo de los Data Lakehouse, donde se están desarrollando estándares abiertos para la interoperabilidad entre distintas herramientas. Sin embargo, todas las iniciativas existentes son extremadamente nuevas y no están maduras; o no son soportadas por algún otro componente de la arquitectura. Tal fue el caso de Project Nessie que no tenía una forma de ser integrado a Apache Doris para su uso; y que de todas maneras será absorbido por Apache Polaris en un futuro.

Desde el punto de vista de su componentes técnicos y debido al enfoque de utilizar las mismas tecnologías para ambas, las arquitecturas son extremadamente similares y no presentan ventajas significativas una sobre otra (más allá de que para utilizar Paimon en Flink es necesario incluir algunos JAR adicionales). De hecho, una vez resuelta la implementación de la arquitectura Kappa (que fue la primera en implementarse), fue trivial implementar la arquitectura Delta.

5.3. Aspectos Operativos

Se realizó una prueba de carga de ambas arquitecturas con datos sintéticos de 32 pacientes a lo largo de un año, con un total de 110.122.654 registros; que implica un archivo CSV de aproximadamente 7GB.

El hardware donde se realizó la prueba cuenta con 64GB de RAM, 16 núcleos de CPU y 2 TB de disco duro de estado sólido, y se utilizó Docker Compose para la ejecución de las pruebas. Se realizaron 5 pruebas de carga para cada arquitectura, y se registraron los tiempos de carga, latencia y uso de recursos. Variaron no solo los tiempos de carga del total del conjunto de datos, sino también el uso de recursos.

Un aspecto importante a destacar es que cada una de las pruebas de carga se realizó en un entorno limpio, es decir, se realizó una reinstalación completa del sistema operativo y de los componentes de la arquitectura. Esto se hizo para evitar que el sistema operativo o los componentes de la arquitectura afecten los resultados de las pruebas.

Un resultado importante a destacar es que la arquitectura Kappa no pudo ser ejecutada en su totalidad en ninguna de las pruebas. Esto debido a que el sistema operativo se quedó sin espacio en disco, lo que provocó que el sistema se detuviera. La cantidad de datos que se pudieron guardar, en promedio, fue del 61 % del total de los datos. Para la arquitectura Delta, se logró cargar el 100 % de los datos en todas las pruebas.

5.3.1. Throughput

El throughput se refiere a la cantidad de datos procesados por unidad de tiempo. En este caso, se midió en base a los mismos límites en el uso de recursos, la cantidad de datos que podían ser ingestados por el sistema hasta completar la carga total o fallar. Para ambas arquitecturas y en cada prueba de carga se registró una ingesta de registros por segundo promedio estable.

Para **Delta** se pudieron ingestar en un promedio de 1300 registros por segundo; y la carga completa llevó 88438 segundos. Para **Kappa** se pudieron ingestar en un promedio de 790 registros por segundo; y la carga del 61 % de los datos llevó 84879 segundos.

La carga de datos en Kappa no pudo completarse completamente ya que en todas las instancias en que se realizó, el hardware usado para el despliegue de la arquitectura consumió toda la memoria de su disco duro.

Se entiende entonces que a nivel de gestión (al menos de disco), Delta es más eficiente que Kappa. A su vez, a nivel de cantidad de mensajes que se pueden consumir por segundo, Delta fue dos veces más eficiente que Kappa.

5.3.2. Latencia

La latencia se refiere al tiempo que tarda un dato en ser procesado por el sistema. En este caso, se midió en base a los mismos límites en el uso de recursos, la cantidad de tiempo que tardó un dato en pasar por todo el flujo de datos.

La latencia se midió en segundos y los resultados fueron los siguientes:

Arquitectura	**Mínima**	**Máxima**	**Promedio**
Kappa 1	-1	333	0.198
Kappa 2	-1	305	0.179
Kappa 3	-1	326	0.176
Kappa 4	-1	329	0.177
Kappa 5	-1	318	0.169
Delta 1	90	611	180
Delta 2	90	609	180
Delta 3	90	609	180
Delta 4	90	610	180
Delta 5	90	610	180

De esta forma promediando los resultados de las 5 pruebas de carga de cada arquitectura:

Arquitectura	**Mínima**	**Máxima**	**Promedio**	**Desviación**
Kappa	-1	322	0.18	0.01
Delta	90	610	180	0

Es interesante ver los resultados. Ya que la latencia mínima de Kappa es negativa. Esto se debe a que el conector de kafka que se usa, no actualiza el campo **aggregation_timestamp** cuando se actualiza el registro porque llegan nuevos datos. Esto muestra que también muestra un problema de diseño del trabajo de inserción al calcular los datos pero no se encontró una solución al mismo.

Por otro lado, se registroó el tiempo de latencia de los registros que se insertaron en el sistema; sin embargo, debido al conector de Flink con Doris, este proceso se realiza en batch, por lo que nuevamente es posible que el registro de tiempo de inserción sea anterior al de procesamiento.

Por su parte, Delta muestra una latencia de varias magnitudes superior a Kafka. Esto es esperable debido al uso extensivo de Object Storage; ya que las lecturas en disco en conjunto con la transferencia por red suelen ser mucho más lentas que las lecturas en memoria. Sin embargo, es importante destacar que la latencia máxima prmedio de Delta es de 610 segundos, lo que implica que en el peor de los casos, el sistema tardó 10 minutos en procesar un dato. Esto es un tiempo aceptable para la mayoría de los casos de uso en los que se implementan flujos de datos; aunque no se puede decir que lo sea para un sistema de procesamiento de datos en tiempo real.

Se puede ver también que ambas arquitecturas son bastante estables en cuanto a latencia, ya que la desviación estándar es baja. Sin embargo, Delta parecería ser más consistente en sus resultados. Sin embargo, esto puede deberse a que la escala de la latencia en la que se maneja Kappa es mucho más baja que la de Delta.

Se analizó también la distribución de latencia de ambos sistemas y también en todas las pruebas de carga se obtuvieron resultados similares. Un ejemplo de estas distribuciones se presentan a continuación:

Rango (segundos)	**Registros**	**Porcentaje (%)**
<1	9962080	81,48
1-10	2264616	18,52
10-60	60	0,0005
60-180	17	0,00014
180-300	17	0,00014
>300	2	1,63575E-05

Cuadro 5.3: Distribución de Latencia en Kappa

Rango (segundos)	Registros	Porcentaje (%)
90-120	122476	0,76
120-150	132107	0,82
150-180	7582878	47,31
180-210	7892041	49,24
210-240	299165	1,87
240-270	1	6,24 E-06
270-300	11	6,86 E-05
330-360	9	5,61 E-05
390-420	3	1,87 E-05
420-450	1	6,24 E-06
600-630	1	6,24 E-06

Cuadro 5.4: Distribución de Latencia en Delta

Más del 80 % de los registros en Kappa tienen una latencia menor a 1 segundo, lo que implica que el sistema es capaz de procesar los datos en tiempo real. Sin embargo, la latencia máxima de Kappa es de 333 segundos, lo que implica que en el peor de los casos, el sistema tardó 5 minutos en procesar un dato (aunque solo para dos casos).

Esto es un tiempo aceptable para la mayoría de los casos de uso en los que se implementan flujos de datos; sobre todo porque un sólo punto de latencia no es un problema para el sistema en su conjunto. Hay también mas de un 18 % de registros que tienen una latencia de entre 1 y 10 segundos, que si bien son tiempos aceptables para la mayoría de los casos de uso, son mayores que los esperados para un sistema de procesamiento de datos en tiempo real.

Dada la cantidad de registros, se podría decir que aquellos puntos de latencia mayores a 10 segundos son outliers. Aunque hay que considerar que para Kappa no se pudo cargar el 100 % de los datos, por lo que no se puede asegurar que esta distribución se mantenga en el tiempo.

Por su parte, Delta tiene casi su totalidad de registros entre 150 y 210 segundos, que no es un tiempo aceptable para un sistema en tiempo real, pero si significa que es bastante consistentes con sus tiempos y su distribución está bien definida. Además, para Delta si se pudo cargar el 100 % de los datos, por lo que se puede asegurar que esta distribución se mantenga en el tiempo.

5.3.3. Uso de Recursos

El uso de recursos se refiere a la cantidad de recursos que utiliza el sistema para procesar los datos. En este caso, se midió en base a los mismos límites en el uso de recursos, la cantidad máxima de recursos que utilizó el sistema para procesar los datos.

Al igual que con la latencia, para todas las pruebas de carga los resultados fueron similares. Por lo que se presenta una tabla con los resultados de una de estas pruebas.

Para **Kappa** se presentan los siguientes resultados:

Componente	**Uso de CPU**	**Memoria**	**Almacenamiento**	**Transferencia**
Job Manager	134 %	1.8 GB	-	54 GB
Task Manager 1	260 %	6.2 GB	-	230 GB
Task Manager 2	244 %	6.3 GB	-	252 GB
Task Manager 3	272 %	6.2 GB	-	243 GB
Task Manager 4	256 %	6.4 GB	-	240 GB
Kafka 1	250 %	8.2 GB	234 GB	255 GB
Kafka 2	282 %	8.4 GB	234 GB	255 GB
Kafka 3	268 %	8.5 GB	234 GB	255 GB
Doris Frontend	370 %	2.4 GB	-	0.55 GB
Doris Backend 1	374 %	4.2 GB	0.234 GB	0.1 GB
Doris Backend 2	345 %	4.2 GB	0.234 GB	0.1 GB
Doris Backend 3	356 %	4.3 GB	0.234 GB	0.1 GB
MinIO	127 %	4.2 GB	384 GB	386 GB

Para **Delta** se presentan los siguientes resultados:

Componente	**Uso de CPU**	**Memoria**	**Almacenamiento**	**Transferencia**
Job Manager	250 %	3.5 GB	-	5.9 GB
Task Manager 1	208 %	9.7 GB	-	86.1 GB
Task Manager 2	166 %	7.6 GB	-	45.2 GB
Task Manager 3	194 %	8.5 GB	-	40.2 GB
Task Manager 4	206 %	7.5 GB	-	10.9 GB
Kafka 1	234 %	2.7 GB	38 GB	101.8 GB
Kafka 2	274 %	2.7 GB	38 GB	101.9 GB
Kafka 3	176 %	2.7 GB	38 GB	102.1 GB
Doris Frontend	365 %	3.6 GB	-	1.1 GB
Doris Backend 1	331 %	4.1 GB	0 GB	0.2 GB
Doris Backend 2	351 %	4.1 GB	0 GB	0.2 GB
Doris Backend 3	340 %	4.1 GB	0 GB	0.2 GB
MinIO	127 %	3.8 GB	8.4 GB	374 GB

Acorde a lo esperado, Kappa requiere un mayor uso de recursos para los nodos de Kafka que Delta. Por su lado, Delta requiere un mayor uso de recursos para los nodos de procesamiento. También para el nodo de frontend de Doris. Aunque es notablemente menor el uso de memoria que requiere de los nodos Kafka.

El uso de CPU es similar en las dos arquitecturas, aunque Delta tiene más diferencias entre nodos del mismo tipo. Probablemente sea debido a un desbalanceo en el paralelismo de los trabajos de procesamiento.

La sorpresa más grande es el uso de almacenamiento y de transferencia de red. Kappa requiere un uso de almacenamiento mucho mayor que Delta, y también un uso de transferencia de red mucho mayor. Esto se debe a que Delta guarda sus datos en Parquet, lo que lo hace muchísimo más eficiente que el formato de Kafka. Incluso, en el caso de Delta, todas las transformaciones de los datos están guardadas en Object Storage y aún así el uso de almacenamiento es notablementemenor que en Kappa.

5.3.4. Resultados

A nivel de throughput, Delta es más eficiente que Kappa. No solo eso sino que es más fácil de operar y consume menos recursos. A nivel de latencia, Kappa es más eficiente que Delta. Sin embargo, la latencia máxima de Delta es aceptable para la mayoría de los casos de uso. En cuanto a uso de memoria y procesamiento, ambas son similares. Aunque Delta es muchísimo mas eficiente en el uso de almacenamiento y transferencia de red.

5.4. Costos

5.4.1. Suposiciones

Se calcularán los costos basados en el despliegue de cada arquitectura en AWS.

La arquitectura desplegada en AWS consiste en:

- Cluster de EKS (Elastic Kubernetes Service) distribuido en 4 Availability Zones (a, b, c, d)
- AZ-a: Doris Backend 1, Task Manager 1, Kafka Broker 1, Zookeeper 1
- AZ-b: Doris Backend 2, Task Manager 2, Kafka Broker 2, Zookeeper 2
- AZ-c: Doris Backend 3, Task Manager 3, Kafka Broker 3, Zookeeper 3
- AZ-d: Doris Frontend, Job Manager, Task Manager 4
- Almacenamiento S3 nativo (sustituyendo MinIO)
- Interconexión entre zonas de disponibilidad

El costo de EKS en AWS es de $0.10 por hora por cada nodo del cluster. Por lo que el costo mensual base es de U$S 72.00.

Se asume además que todos los nodos experimentan una carga constante igual a la mayor carga registrada en las pruebas. Esto asegura que se mantenga la disponibilidad del sistema.

Por otro lado, Zookeeper no fue incluido en la comparación de costos operativos ya que no hace una diferencia. Sin embargo, será incluido en el costo total de ambas arquitecturas.

Se asume que para todos los nodos, salvo Job Manager y Doris Frontend hay un 25 % de uso de red en la misma zona de disponibilidad y un 75 % de uso de red entre zonas de disponibilidad. El Job Manager, por el despliegue que definimos, tendrá un 100 % de uso de red en zonas de disponibilidad distintas. A su vez Doris Frontend tendrá un 100 % de uso de red hacia internet.

5.4.2. Costos de la Arquitectura Kappa

Para el caso de Kappa, se asumirá que tanto el almacenamiento como el uso de red medidos corresponden al 61 % del valor real si se hubiera podido completar la carga de datos completa.

De esta forma, se consideran instancias que cumplan estos requisitos.
El caso más complejo es el de Kafka, que requiere un muy buen acceso a disco y una buena red. Y por otro lado, necesita que si se cae un nodo no se pierdan datos, ya que es nuestra fuente de verdad.

Para cumplir los patrones de almacenamiento vistos en la prueba se propone para Kafka el uso de una instancia r6i.xlarge con 4vCPU y 32GB de RAM, que permite asegurar que los nodos no estén sobrecargados. Además se le deberá agregar un volúmen persistente EBS de 1TB para asegurar que el nodo pueda almacenar los datos necesarios.

Para el caso de Doris, se utilizará una instancia c5d.xlarge para el frontend (que viene con un disco) y una c5.x2large para el backend ya que requieren un buen uso de CPU y de red. Además, el backend debe tener un volúmen persistente de al menos 50 GB para asegurar que pueda almacenar los datos necesarios y estos no se pierdan.

Para el caso de Flink, se utilizará una instancia r5.xlarge para los nodos Task Manager ya que proveen un muy buen rendimiento de memoria y una c5.large para el Job Manager.

Para el caso de Zookeeper, se utilizará una instancia m5.large que asegura un buen funcionamiento de este sistema.

Componente	Cantidad	Instancia	Costo	Total
Job Manager	1	c5d.large	$69,12	$69,12
Task Manager	4	r5d.large	$181,44	$725,76
Kafka Broker	3	r6i.xlarge	$144,72	$434,16
Zookeeper	3	m5.large	$69,12	$207,36
Doris Frontend	1	c5d.xlarge	$144,72	$144,72
Doris Backend	3	c5.2xlarge	$244,8	$734,4

Cuadro 5.7: Costo de Kappa en instancias

Componente	Cantidad	Servicio	Tamaño	Costo	Total
Job Manager	1	Incluido	50 GB	$0	$0
Task Manager	4	Incluido	50 GB	$0	$0
Kafka Broker	3	EBS gp3	1 TB	$398,72	$1196,16
Zookeeper	3	Incluido	50 GB	$0	$0
Doris Frontend	1	Incluido	50 GB	$0	$0
Doris Backend	3	EBS gp3	50 GB	$4	$12
S3	N/A	S3	1TB	$23,55	$23,55

Cuadro 5.8: Costo de Kappa en almacenamiento

Componente	Cantidad	Salida	Entre AZ	Internet	Total
Job Manager	1	90 GB	$0.90	$0	$0.9
Task Manager	4	420 GB	$12,6	$0	$12,6
Kafka Broker	3	420 GB	$9,45	$0	$9,45
Doris Frontend	1	1 GB	$0	$0.09	$0.09
Doris Backend	3	0,2 GB	$0.0045	$0.00	$0.0045
S3	N/A	700 GB	N/A	N/A	$7

Cuadro 5.9: Costo de Kappa en uso de red

Componente	Costo
Instancia	$2315,52
Almacenamiento	$1231,71
Red	$30,04
Total	**$3577,27**

Cuadro 5.10: Costo de Kappa

5.4.3. Costos de la Arquitectura Delta

Para el caso de Delta, las instancias son similares a Kappa, aunque su menor uso de Kafka permite reducir el costo en el tipo de instancia. Por otro lado, como la fuente de verdad es el almacenamiento en S3, tampoco se necesitan tener volúmenes persistentes ni para Kafka ni para Doris.

Para cumplir con estos requisitos se propone el uso de una instancia c5d.xlarge para Kafka que cuenta con 4 vCPU, 8 GB RAM y 100 GB de disco efímero.

Para el resto de las instancias se utilizarán las misma que para Kappa. S3 es un caso especial, ya que gracias a la optimización de Paimon con el uso de Parquet se necesita menos espacio para almacenar los datos y se pueden reducir los requisitos.

Componente	**Cantidad**	**Instancia**	**Costo**	**Total**
Job Manager	1	c5d.large	$69,12	$69,12
Task Manager	4	r5d.large	$181,44	$725,76
Kafka Broker	3	c5d.xlarge	$138,24	$414,72
Zookeeper	3	m5.large	$69,12	$207,36
Doris Frontend	1	c5d.xlarge	$144,72	$144,72
Doris Backend	3	c5.2xlarge	$244,8	$734,4

Cuadro 5.11: Costo de Delta en instancias

Componente	**Cantidad**	**Servicio**	**Tamaño**	**Costo**	**Total**
Job Manager	1	Incluido	50 GB	$0	$0
Task Manager	4	Incluido	50 GB	$0	$0
Kafka Broker	3	Incluido	50 GB	$0	$0
Zookeeper	3	Incluido	50 GB	$0	$0
Doris Frontend	1	Incluido	50 GB	$0	$0
Doris Backend	3	Incluido	50 GB	$0	$0
S3	N/A	S3	50 GB	$1,15	$1,15

Cuadro 5.12: Costo de Delta en almacenamiento

Componente	**Cantidad**	**Salida**	**Entre AZ**	**Internet**	**Total**
Job Manager	1	6 GB	$0.06	$0	$0.06
Task Manager	4	86 GB	$2,58	$0	$2,58
Kafka Broker	3	102 GB	$2,295	$0	$2,295
Doris Frontend	1	1,1 GB	$0	$0.099	$0.099
Doris Backend	3	0,2 GB	$0.0045	$0.00	$0.0045
S3	N/A	374 GB	N/A	N/A	$3,74

Cuadro 5.13: Costo de Delta en uso de red

Componente	**Costo**
Instancia	$2288,16
Almacenamiento	$1,15
Red	$8,78
Total	**$2298,09**

Cuadro 5.14: Costo de Delta

5.4.4. Resultados

El costo de delta es un **64 %** del costo de Kappa. Esto se debe sobre todo a la nececsidad que tiene Kappa de que Kafka sea su fuente de verdad. Y si bien el almacenamiento en capas que ofrece Kafka permite reducir costos al almacenar sus segmentos en Object Storage, aún así necesita de mucha cantidad de disco para funcionar correctamente en un entorno de alta disponibilidad.

Por su lado, Delta hace un uso extensivo de Object Storage en combinación con el formato Parquet, lo que reduce tanto la necesidad de almacenamiento como el tráfico de red; volviéndose una solución mucho más económica.

En cuanto al costo de las instancias, no existen diferencias significativas entre las arquitecturas.

5.5. Análisis de Resultados

El análisis de resultados permitió evidenciar que las arquitecturas Kappa y Delta presentan características, ventajas y limitaciones diferenciadas, las cuales deben ser consideradas cuidadosamente según el contexto de aplicación.

La arquitectura Kappa demostró ser altamente eficiente para el procesamiento en tiempo real, logrando latencias inferiores al segundo en más del 80 % de los casos. Esta capacidad la posiciona como la opción más adecuada en escenarios donde la reacción inmediata es crítica, como en el monitoreo de pacientes en unidades de cuidados intensivos o emergencias. No obstante, esta eficiencia viene acompañada de un mayor consumo de recursos, una complejidad operativa superior y una necesidad considerable de almacenamiento, debido a su dependencia de Kafka como fuente de verdad.

Por otro lado, la arquitectura Delta evidenció una mayor escalabilidad y una mejor eficiencia en el uso de recursos, especialmente en almacenamiento y transferencia de datos, gracias al uso de formatos columnar como Parquet y almacenamiento en Object Storage. Sin embargo, sus tiempos de latencia (aunque estables) alcanzaron valores promedio de alrededor de 180 segundos, lo que la hace inadecuada para casos que requieren inmediatez. Aun así, esta latencia puede considerarse aceptable en contextos como el monitoreo ambulatorio, donde las decisiones clínicas no dependen de una reacción instantánea.

En cuanto a throughput, Delta logró duplicar la capacidad de ingesta de Kappa, procesando hasta 1300 registros por segundo frente a los 790 de Kappa. Asimismo, logró completar el 100 % de la carga de datos, a diferencia de Kappa, que presentó fallos por limitaciones en el uso de disco durante el despliegue.

Desde la perspectiva económica, el despliegue de Delta implicó un costo 36 % menor que el de Kappa. Esta diferencia se debe principalmente a la menor necesidad de almacenamiento persistente y al menor tráfico de red requerido.

Dado lo anterior, la elección entre Kappa y Delta dependerá del caso de uso específico.

Para un sistema de salud integral, una estrategia híbrida que combine ambas arquitecturas se perfila como una solución óptima. En este enfoque, Delta puede actuar como fuente de verdad y repositorio histórico, mientras que un subsistema basado en Kappa podría operar en paralelo para los casos donde la baja latencia sea un requerimiento indispensable.

Esta estrategia se asemeja conceptualmente a la arquitectura Lambda, aunque enfocada en el procesamiento continuo de datos y utilizando tecnologías actuales como Doris, que permite consultas federadas sobre los datos almacenados en ambas arquitecturas; y Kafka que permite el ruteo de mensajes a ambos sistemas a la vez paraque cada uno lo procese a la velocidad que pueda.

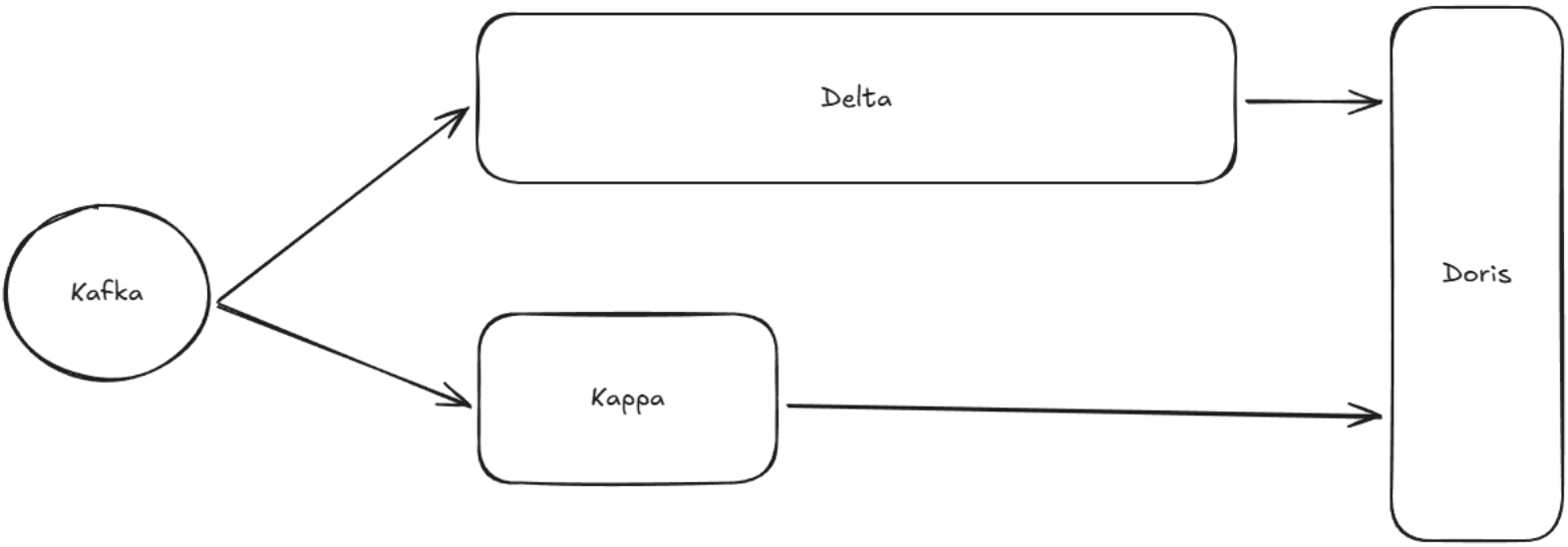

Figura 5.1: Arquitectura combinada de un sistema de salud integral

Capítulo 6

Conclusiones

6.1. Conclusiones Generales

El objetivo de este trabajo fue comparar las arquitecturas Kappa y Delta en el contexto del monitoreo remoto de pacientes mediante sensores. Se realizó un análisis exhaustivo de ambas arquitecturas, considerando sus componentes, flujos de datos y características técnicas.

Se definieron métricas y criterios para la comparación objetiva de ambas arquitecturas, y se implementaron en un caso de uso simulado. Se implementó un sistema de monitoreo remoto de pacientes utilizando ambas arquitecturas, y se realizaron pruebas de rendimiento y funcionalidad. Los resultados obtenidos mostraron que ambas arquitecturas tienen ventajas y desventajas en diferentes contextos.

La arquitectura Kappa es más adecuada para el procesamiento de datos en tiempo real, con baja latencia, pero requiere más recursos y es más compleja de implementar. La arquitectura Delta, por otro lado, es más escalable y eficiente en el uso de recursos, pero tiene tiempos de latencia más altos, lo que la hace menos adecuada para el procesamiento en tiempo real.

En un sistema de salud integral, se podría combinar ambas arquitecturas, utilizando Delta como fuente de verdad y un subsistema basado en Kappa para el monitoreo intrahospitalario, donde la latencia es crítica. Esta combinación permitiría aprovechar las ventajas de ambas arquitecturas y mejorar la eficiencia del sistema en su conjunto.

6.2. Trabajo Futuro

Este trabajo puede abrir las puertas a futuras investigaciones en el área de monitoreo remoto de pacientes y análisis de datos en tiempo real. Se pueden explorar diversas direcciones de investigación:

- **Optimización de Parámetros:** Investigar la optimización de ventanas de tiempo y tasas de degradación para mejorar la validez de las mediciones.
- **Mejora de la Evaluación de Calidad:** Desarrollar métricas de calidad más precisas y protocolos de validación para nuevos tipos de dispositivos.
- **Validación Clínica:** Comparar la efectividad del sistema gdNEWS2 con métodos tradicionales en entornos clínicos.
- **Optimización del Sistema:** Investigar parámetros de rendimiento y capacidades de integración para mejorar la eficiencia del sistema.
- **Tecnologías Emergentes:** Explorar el uso de tecnologías emergentes como inteligencia artificial y aprendizaje automático para mejorar la toma de decisiones en tiempo real.
- **Interoperabilidad:** Investigar la interoperabilidad entre diferentes sistemas y dispositivos para mejorar la integración de datos.
- **Seguridad y Privacidad:** Evaluar la seguridad y privacidad de los datos en el contexto del monitoreo remoto de pacientes.
- **Experiencia del Usuario:** Investigar la experiencia del usuario y la usabilidad de las interfaces de monitoreo remoto.
- **Governanza de Datos:** Desarrollar políticas y prácticas para la gobernanza de datos en el contexto del monitoreo remoto de pacientes.
- **Despliegue Contínuo:** Explorar las diferentes herramientas y sus limitaciones en el despliegue continuo de sistemas de streaming.
- **Comparación con otras arquitecturas:** Comparar la arquitectura Delta con otras arquitecturas de procesamiento de datos o arquitecturas híbridas.
- **Evaluación en un entorno real:** Realizar una evaluación del sistema en un entorno real, con datos reales y pacientes reales.

6.3. Reflexiones Finales

Este trabajo me ha permitido explorar y comparar las arquitecturas Kappa y Delta en el contexto del monitoreo remoto de pacientes. Esta es un área en constante evolución y crecimiento, y la implementación de estas arquitecturas puede tener un impacto significativo en la atención médica.

Además, desde el punto de vista académico, este trabajo me ha permitido profundizar en el análisis de datos en tiempo real. Esto implicó la investigación, aprendizaje y aplicación de diferentes herramientas y tecnologías que no sólo aplican a este trabajo sino que podré aplicar en el ámbito profesional.

Este proceso ha sido desafiante porque he tenido que aprender a utilizar herramientas y tecnologías que no conocía previamente y que no están tan difundidas como aquellas con las que trabajo habitualmente.

Sin embargo, esto me ha permitido adquirir nuevas habilidades y conocimientos que serán valiosos en mi carrera profesional.

Por último, ha sido un proceso gratificante y espero poder seguir profundizando en el esta área en el futuro.

Apéndice A

Repositorio de código

Se definireron tres repositorios de código para el desarrollo de la arquitectura Kappa y Delta.

El primero de ellos es el repositorio de la arquitectura Kappa, que contiene el código de procesamiento de datos y la configuración de los componentes.

El segundo es el repositorio de la arquitectura Delta, que contiene el código de procesamiento de datos y la configuración de los componentes.

El tercero es el repositorio del generador de datos sintéticos que se utilizó para realizar las pruebas de carga y estrés.

El código de cada uno de los repositorios se encuentra disponible en el siguiente enlace:

- `https://github.com/Rekeyea/Tesis-Kappa`
- `https://github.com/Rekeyea/Tesis-Delta`
- `https://github.com/Rekeyea/Tesis-SynthDS`

Bibliografía

[1] T. Akidau, R. Bradshaw, C. Chambers y otros, «The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,» *Proceedings of the VLDB Endowment*, vol. 8, n.º 12, págs. 1792-1803, 2015. DOI: 10.14778/2824032.2824076.

[2] M. Armbrust, T. Das, L. Sun y otros, «Delta lake: high-performance ACID table storage over cloud object stores,» *Proceedings of the VLDB Endowment*, vol. 13, n.º 12, págs. 3411-3424, 2020. DOI: 10.14778/3415478.3415560.

[3] T. Arora, V. Balasubramanian, A. Stranieri y otros, «Modified Early Warning Score (MEWS) Visualization and Pattern Matching Imputation in Remote Patient Monitoring,» *IEEE Access*, vol. 12, págs. 74 784-74 794, 2024. DOI: 10.1109/ACCESS.2024.3396274.

[4] P. Carbone, A. Katsifodimos, S. Ewen y otros, «Apache Flink: Stream and Batch Processing in a Single Engine,» *IEEE Data Engineering Bulletin*, vol. 38, 2015.

[5] V. Dankan Gowda, D. Joshi, C. Senthil Kumar y otros, «Impact of IoT on Remote Patient Monitoring and Advancements in Telemedicine,» en *2024 Second International Conference on Intelligent Cyber Physical Systems and Internet of Things (ICoICI)*, 2024, págs. 319-326. DOI: 10.1109/ICoICI62503.2024.10696558.

[6] J. Dean y S. Ghemawat, «MapReduce: Simplified Data Processing on Large Clusters,» *Communications of the ACM*, vol. 51, n.º 1, págs. 107-113, 2008. DOI: 10.1145/1327452.1327492.

[7] Y. Demchenko, C. De Laat y P. Membrey, «Defining architecture components of the Big Data Ecosystem,» en *2014 International Conference on Collaboration Technologies and Systems (CTS)*, vol. 1, Minneapolis, MN, USA: IEEE, 2014, págs. 104-112. DOI: 10.1109/CTS.2014.6867550.

[8] F. Hueske y V. Kalavri, *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*, 1.ª ed. Sebastopol, CA: O'Reilly Media, 2019.

[9] M. Kleppmann, *Designing Data-Intensive Applications*, 1.ª ed. O'Reilly Media Inc., 2018.

[10] J. Kreps. «Questioning the Lambda Architecture.» (2014), dirección: https://www.oreilly.com/radar/questioning-the-lambda-architecture/ (visitado 7 de oct. de 2024).

[11] H. Leano. «Delta vs. Lambda: Why Simplicity Trumps Complexity for Data Pipelines,» Databricks. (20 de nov. de 2020), dirección: https://www.databricks.com/blog/2020/11/20/delta-vs-lambda-why-simplicity-trumps-complexity-for-data-pipelines.html (visitado 7 de oct. de 2024).

[12] N. Marz. «How to beat the CAP theorem.» (2011), dirección: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html (visitado 8 de oct. de 2024).

[13] N. Marz y J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*, 1.ª ed. USA: Manning Publications Co., 2015.

[14] F. D. Muñoz-Escoí, R. de Juan-Marín, J.-R. García-Escrivá y otros, «CAP Theorem: Revision of Its Related Consistency Models,» *The Computer Journal*, vol. 62, n.º 6, págs. 943-960, 2019. DOI: 10.1093/comjnl/bxy142.

[15] D. Preuveneers, Y. Berbers y W. Joosen, «SAMURAI: A batch and streaming context architecture for large-scale intelligent applications and environments,» *Journal of Ambient Intelligence, Smart Environments*, vol. 8, n.º 1, págs. 63-78, 2016. DOI: 10.3233/AIS-160365.

[16] M. Stonebraker, U. Çetintemel y S. Zdonik, «The 8 requirements of real-time stream processing,» *ACM SIGMOD Record*, vol. 34, n.º 4, págs. 42-47, 2005. DOI: 10.1145/1107499.1107504.

[17] A. S. Tanenbaum y M. van Steen, *Distributed Systems: Principles and Paradigms*, 3.ª ed. Pearson Education, Inc, 2020.

[18] A. Toshniwal, S. Taneja, A. Shukla y otros, «Storm@ twitter,» en *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, págs. 147-156. DOI: 10.1145/2588555.2595641.