Comparación de las arquitecturas Kappa y Delta para el monitoreo remoto de pacientes basado en datos de sensores

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

Emiliano Conti -

Tutor: Alejandro Bianchi

Universidad ORT

27 de octubre de 2024

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

Aquí va tu resumen...

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, flujos de datos y casos de uso típicos.

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.

6. Proporcionar recomendaciones para la selección e implementación de arquitecturas de procesamiento de Big Data en el ámbito de la salud digital.

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 medicos. Asi como tambien 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 histórico
    - Modelos predictivos (machine learning)
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. (Tanenbaum & van Steen, 2020)

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 estan caidos, 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 en la industria de IT 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 (conjuntos/activos) 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 (análisis) rentables e innovadoras para mejorar el conocimiento, la toma de decisiones y el control de procesos; todo esto exige (debe ser apoyado por) 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 (incluidas las redes de sensores) y entregar datos en una variedad de formas a diferentes consumidores y dispositivos de datos e información. (Demchenko et al., 2014)

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. (Hueske & Kalavri, 2019)

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 tambien 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.(Muñoz-Escoí et al., 2019)

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 (Muñoz-Escoí et al., 2019)

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 procesamiento de streams es lograr equilibrar la latencia, el costo y la correctitud simultáneamente (Akidau et al., 2015).

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 procesamiento de streams (Hueske & Kalavri, 2019, 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 (Preuveneers et al., 2016).

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 "exactamente una vez" en presencia de fallos es un desafío significativo en el procesamiento de streams (Carbone et al., 2015).

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 (Akidau et al., 2015).

6. **Integración con sistemas batch**

   Combinar eficazmente el procesamiento de streams 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 (Carbone et al., 2015).

2.1.6. Regluaciones en el Uso de Datos: GDPR

2.1.7. Conceptos clave en el procesamiento de streaming

El procesamiento de streaming se refiere al análisis y manipulación de datos en tiempo real a medida que se generan o reciben. Según Carbone et al. (Carbone et al., 2015), los conceptos fundamentales incluyen:

- **Flujo de datos**: Una secuencia potencialmente infinita de registros que llegan continuamente (Akidau et al., 2015).
- **Latencia**: El tiempo entre la llegada de un dato y su procesamiento, crucial para aplicaciones en tiempo real (Akidau et al., 2015).
- **Ventanas**: Mecanismos para agrupar datos en intervalos finitos para su procesamiento (Akidau et al., 2015).
- **Estado**: Información que se mantiene entre eventos para cálculos incrementales (Carbone et al., 2015).
- **Watermarks**: Indicadores de progreso del tiempo en el flujo de datos (Akidau et al., 2015).

2.1.8. Comparación entre procesamiento por lotes y en tiempo real

La elección entre procesamiento por lotes y en tiempo real depende de los requisitos específicos de la aplicación:

Característica	**Procesamiento por lotes**	**Procesamiento en tiempo real**
Latencia	Alta (minutos a horas)	Baja (milisegundos a segundos)
Throughput	Alto	Moderado a alto
Complejidad	Menor	Mayor
Consistencia	Fuerte	Eventual
Uso típico	Análisis histórico, reportes	Monitoreo, alertas, decisiones inmediatas

Cuadro 2.1: Comparación de procesamiento por lotes y en tiempo real

Como sugiere Stonebraker et al. (Stonebraker et al., 2005), el procesamiento en tiempo real es esencial para aplicaciones que requieren decisiones

inmediatas, mientras que el procesamiento por lotes es más adecuado para análisis profundos de grandes volúmenes de datos históricos.

2.1.9. 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 por lotes**: Sistemas tradicionales como Hadoop MapReduce, diseñados para procesar grandes volúmenes de datos estáticos (Dean & Ghemawat, 2008).

2. **Arquitecturas de streaming puro**: Como Apache Storm, enfocadas en el procesamiento en tiempo real pero con limitaciones en la consistencia y exactitud (Toshniwal et al., 2014).

3. **Arquitectura Lambda**: Propuesta por Marz (Marz, 2011), combina procesamiento por lotes y en tiempo real para balancear latencia, throughput y tolerancia a fallos.

4. **Arquitectura Kappa**: Introducida por Kreps (Kreps, 2014), simplifica la Lambda tratando todos los datos como streams.

5. **Arquitectura Delta**: Desarrollada por Databricks, combina las ventajas de las arquitecturas Lambda y Kappa, optimizando el procesamiento de datos tanto en batch como en streaming (Armbrust et al., 2020) (Leano, 2020).

2.2. Tecnologias para Streaming en Big Data

2.2.1. Mensajería Distribuida

Las tecnologias 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. (Kleppmann, 2018)

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. (Marz & Warren, 2015)

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 tambien 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.

Comparación

Escalabilidad

- **Apache Kafka:** 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 indefinida.
- **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 y retención ilimitada.
- Kafka ofrece mayor flexibilidad en configuración.
- Servicios gestionados (Kinesis, Event Hubs) tienen encriptación en reposo por defecto, pero retención limitada.
- Pulsar ofrece exactly-once nativo, Kafka lo requiere configurable.

Cuadro 2.2: Comparación de Sistemas de Mensajería

Característica	Apache Kafka	Apache Pulsar	Amazon Kinesis	Azure Event Hubs
Escalabilidad	Alta	Muy alta	Alta	Alta
Retención	Configurable	Ilimitada	365 días máx.	7 días máx.
Garantía entrega	At-least-once, Exactly-once*	Exactly-once	At-least-once	At-least-once
Encript. reposo	Configurable	Sí	Sí (AWS KMS)	Sí
Encript. tránsito	Sí	Sí	Sí	Sí
Throughput	Muy alto (configurable)	Muy alto (configurable)	1MB/s por shard	1MB/s por unidad

2.2.2. 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 microbatches 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 fué 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 cfrece 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.

Cuadro 2.3: Comparativa de Motores de Procesamiento Big Data

	Modelo	**Estado**	**Latencia**	**Throughput**	**Garantías**
Apache Spark	Micro-batches	En memoria	Media	Muy alto	At-least-once
Apache Flink	Streaming	Distribuido	Muy baja	Alto	Exactly-once
Apache Beam	Modelo unificado	Según runner	Variable	Variable	Según runner
Apache Samza	Streaming	Local por partición	Baja	Alto	At-least-once
Apache NiFi	Flujo dirigido	Local por procesador	Media-Alta	Medio	At-least-once
Kafka Streams	Streaming	Distribuido	Baja	Alto	Exactly-once
Pulsar Funcitons	Mensaje	Distribuido	Muy Baja	Alto	Exactly-once

Comparación

2.2.3. 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) fue inicialmente 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.

Sistemas de Archivos Distribuidos

Hadoop

Almacenamiento de Objetos en la Nube

2.2.4. Desafíos

Procesamiento fuera de Orden

Small File Issue

2.3. Arquitectura Lambda

2.3.1. 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 por lotes (batch) y en tiempo real, combinando ambos enfoques para proporcionar una vista completa y actualizada de los datos.

2.3.2. 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 de lotes.
- 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 Logica

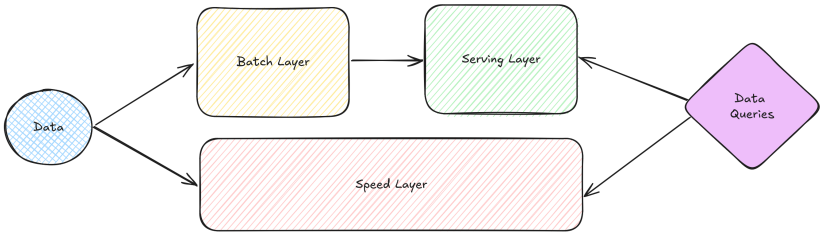

Figura 2.1: Diagrama de la Arquitectura Lambda

Implementacion Tipica

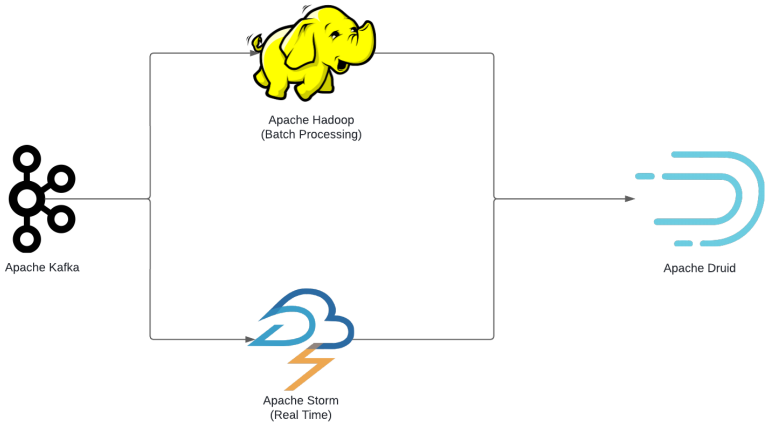

Figura 2.2: IMplementacion de la Arquitectura Lambda

2.3.3. 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 por lotes 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.

2.3.4. Debilidades

- **Complejidad**: La implementación y mantenimiento pueden ser complejos debido a la duplicación de lógica en las capas de lotes y velocidad.
- **Latencia**: El procesamiento por lotes 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.

2.3.5. Conclusiones

La Arquitectura Lambda ofrece una solución robusta para el procesamiento de Big Data, combinando las ventajas del procesamiento por lotes y en tiempo real. Aunque presenta desafíos en términos de complejidad, latencia y costo.

2.4. Arquitectura Kappa

2.4.1. Descripción General

La Arquitectura Kappa es un patrón de arquitectura de procesamiento de datos propuesto por Jay Kreps en 2014 como una simplificación de la Arquitectura Lambda. Su objetivo principal es unificar el procesamiento por lotes y en tiempo real en un único flujo de datos, eliminando la necesidad de mantener códigos separados para estos dos tipos de procesamiento. La Arquitectura Kappa se basa en la premisa de que todo es un flujo de datos (Stream) y que el reprocesamiento se puede lograr simplemente reproduciendo este flujo desde el principio.

2.4.2. 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.

Vista Logica

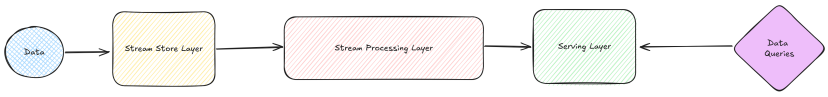

Figura 2.3: Diagrama de la Arquitectura Kappa

Implementacion Tipica

Figura 2.4: Implementacion de la Arquitectura Kappa

2.4.3. Capacidades

La Arquitectura Kappa ofrece varias capacidades clave:

- **Simplificación**: Al unificar el procesamiento batch y en tiempo real, reduce la complejidad del sistema.
- **Consistencia**: Garantiza la coherencia entre los resultados del procesamiento en tiempo real y el reprocesamiento.
- **Escalabilidad**: Se adapta fácilmente al crecimiento del volumen de datos.
- **Reprocesamiento**: Permite actualizaciones sencillas de la lógica de procesamiento mediante el reprocesamiento del stream.
- **Latencia reducida**: Proporciona resultados en tiempo real con menos latencia que Lambda para la mayoría de los casos de uso.

2.4.4. Debilidades

A pesar de sus ventajas, la Arquitectura Kappa tiene algunas limitaciones:

- **Complejidad tecnologica**: Requiere tecnologias con caracteristicas muy especificas en la capa de Stream Store.
- **Dependencia del almacenamiento**: Requiere un sistema de almacenamiento capaz de retener grandes volúmenes de datos históricos.
- **Complejidad**: Algunos análisis complejos pueden ser más difíciles de implementar en un modelo puramente basado en Streams.

2.4.5. Conclusiones

La Arquitectura Kappa representa una alternativa interesante en el diseño de sistemas de procesamiento de datos, ofreciendo una solución elegante para unificar el procesamiento batch y en tiempo real. Su enfoque en el procesamiento de streams como paradigma único simplifica la arquitectura general y reduce la complejidad del mantenimiento del código.

Mientras que es ideal para muchos casos de uso modernos de procesamiento de datos, especialmente aquellos que requieren resultados en tiempo real y flexibilidad en la actualización de la lógica de procesamiento, puede no ser la mejor opción para todos los escenarios.

2.5. Arquitectura Delta

2.5.1. Fundamentos y principios de diseño

2.5.2. Componentes clave

2.5.3. Manejo de datos en la arquitectura Delta

2.5.4. Ventajas y limitaciones

2.5.5. Casos de uso ideales

Capítulo 3

Metodología

3.1. Criterios de Evaluación para Arquitecturas de Streaming

Para evaluar y comparar las arquitecturas Kappa y Delta en el contexto del monitoreo remoto de pacientes, se considerarán los siguientes criterios:

3.1.1. Latencia de Procesamiento

- Tiempo de respuesta para el procesamiento de datos en tiempo real
- Capacidad para manejar picos de datos sin aumentar significativamente la latencia

3.1.2. Escalabilidad

- Capacidad para manejar un aumento en el volumen de datos
- Facilidad de agregar recursos computacionales según sea necesario
- Rendimiento bajo diferentes cargas de trabajo

3.1.3. Consistencia de Datos

- Garantía de consistencia entre datos en tiempo real y datos históricos
- Manejo de datos fuera de orden o retrasados

3.1.4. Tolerancia a Fallos

- Capacidad de recuperación ante fallos del sistema
- Prevención de pérdida de datos en caso de interrupciones

3.1.5. Manejo de Datos Históricos

- Eficiencia en el acceso y análisis de datos históricos
- Capacidad para reprocesar datos históricos cuando sea necesario

3.1.6. Costo Operativo

- Requisitos de hardware y software
- Costos de mantenimiento y operación a largo plazo

3.1.7. Seguridad y Cumplimiento Normativo

- Capacidad para cifrar datos en tránsito y en reposo
- Cumplimiento con regulaciones de protección de datos en salud (por ejemplo, HIPAA)

3.1.8. Rendimiento en Análisis Complejos

- Capacidad para realizar análisis en tiempo real de múltiples fuentes de datos
- Eficiencia en la ejecución de modelos de machine learning

Estos criterios servirán como base para una evaluación exhaustiva y objetiva de las arquitecturas Kappa y Delta en el contexto del monitoreo remoto de pacientes, permitiendo una comparación detallada y fundamentada.

Capítulo 4

Desarrollo

Contenido del capítulo...

4.1. Implementación Arquitectura Kappa

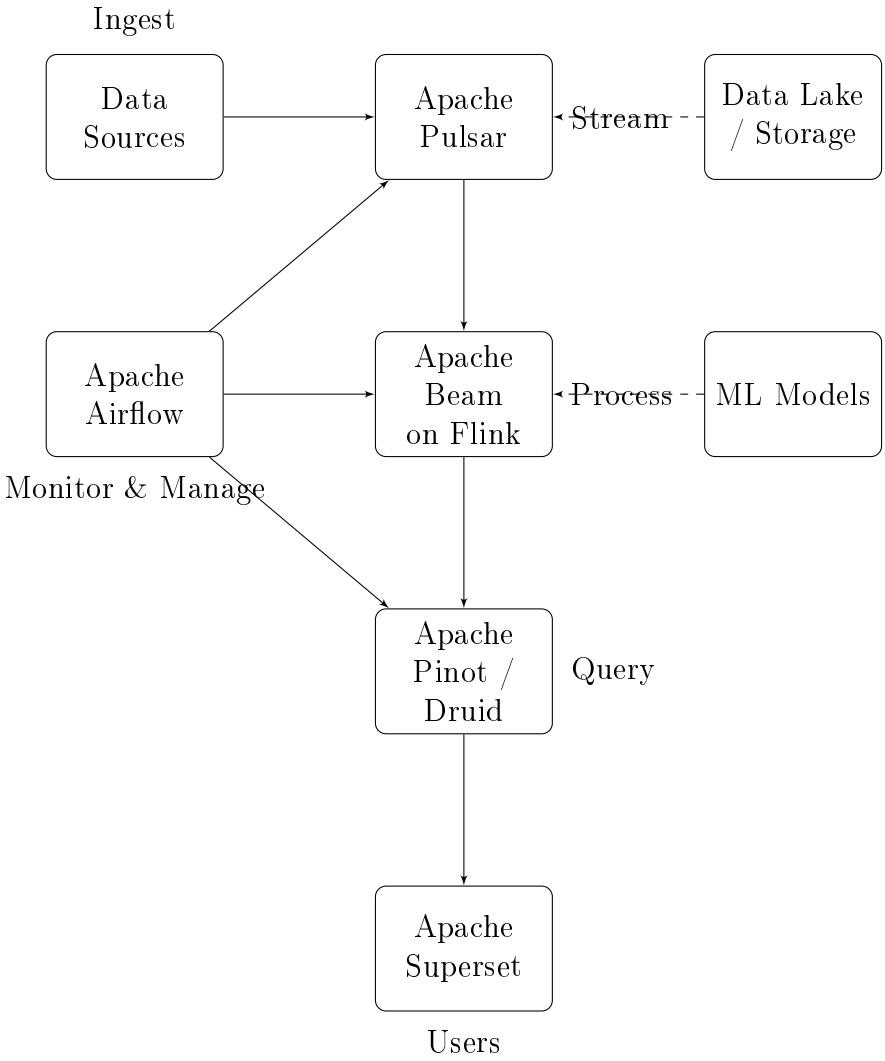

Figura 4.1: Pulsar-Centric Kappa Architecture

Capítulo 5

Resultados

Contenido del capítulo...

Capítulo 6

Conclusiones

Contenido del capítulo...

Apéndice A

Primer Anexo

Contenido del anexo...

Bibliografía

Stonebraker, M., Çetintemel, U., & Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, *34*(4), 42-47.

Dean, J., & Ghemawat, S. (2008). MAPREDUCE: SIMPLIFIED DATA PROCESSING ON LARGE CLUSTERS. *Communications of the ACM*, *51*(1), 107-113. https://research.ebsco.com/linkprocessor/plink?id=1a5fefb3-a714-300d-bc4c-94603fe83a6f

Marz, N. (2011). *How to beat the CAP theorem* [Accessed on 2024-10-08]. Nathan Marz's Blog. Consultado el 8 de octubre de 2024, desde http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

Demchenko, Y., Laat, C. D., & Membrey, P. (2014). Defining architecture components of the Big Data Ecosystem. *2014 International Conference on Collaboration Technologies and Systems (CTS)*, *1*(2), 104-112.

Kreps, J. (2014). *Questioning the Lambda Architecture*. Consultado el 7 de octubre de 2024, desde https://www.oreilly.com/radar/questioning-the-lambda-architecture/

Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al. (2014). Storm@ twitter. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 147-156.

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., et al. (2015). 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*, *8*(12), 1792-1803.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, *36*(4).

Marz, N., & Warren, J. (2015). *Big Data: Principles and best practices of scalable realtime data systems* (1st). Manning Publications Co.

Preuveneers, D., Berbers, Y., & Joosen, W. (2016). SAMURAI: A batch and streaming context architecture for large-scale intelligent applications and environments. *Journal of Ambient Intelligence, Smart Environments*, *8*(1), 63-78. https://research.ebsco.com/linkprocessor/plink?id=afac9a51-b38e-3302-b299-be23cea08349

Kleppmann, M. (2018). *Designing Data-Intensive Applications* (1.ª ed., Vol. 1). O'Reilly Media Inc.

Hueske, F., & Kalavri, V. (2019). *Stream Processing with Apache Flink : Fundamentals, Implementation, and Operation of Streaming Applications* (First edition). O'Reilly Media.

Muñoz-Escoí, F. D., Juan-Marín, R. d., García-Escrivá, J.-R., Mendívil, J. R. G. d., & Bernabéu-Aubán, J. M. (2019). CAP Theorem: Revision of Its Related Consistency Models. *Computer Journal*, *62*(6), 943-960. https://research.ebsco.com/linkprocessor/plink?id=3508200f-f883-34a0-a32c-3dea6a6209e0

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., undefinedwitakowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., ... Zaharia, M. (2020). Delta lake: high-performance ACID table storage over cloud object stores. *Proc. VLDB Endow.*, *13*(12), 3411-3424. https://doi.org/10.14778/3415478.3415560

Leano, H. (2020, 20 de noviembre). *Delta vs. Lambda: Why Simplicity Trumps Complexity for Data Pipelines*. Databricks. https://www.databricks.com/blog/2020/11/20/delta-vs-lambda-why-simplicity-trumps-complexity-for-data-pipelines.html

Tanenbaum, A. S., & van Steen, M. (2020). *Distributed Systems: Principles and Paradigms* (Third edition). Pearson Education, Inc.