Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Ingeniería Informática

PROCESAMIENTO Y VISUALIZACIÓN DE DATOS A GRAN ESCALA APLICADO AL COMERCIO ELECTRÓNICO

Autor: Sergio Vicente de las Heras

Director: Jesús Montes Sánchez

Madrid, 16 de junio de 2018

PROCESAMIENTO Y VISUALIZACIÓN DE DATOS A GRAN ESCALA APLICADO AL COMERCIO ELECTRÓNICO

Autor: Sergio Vicente de las Heras
Director: Jesús Montes Sánchez

DATSI
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

16 de junio de 2018

Dedicatoria

Dedicado a mi familia quienes me apoyaron en todo momento durante estos años en la universidad.

Agradecimientos

Quiero agradecer a la Escuela Técnica Superior de Ingenieros Informáticos la formación recibida durante estos dos años de máster además de a mi tutor del trabajo de fin de máster, Jesús Montes Sánchez, que gracias a su apoyo ha hecho posible este proyecto.

También quiero agradecer a Payvision el apoyo y los recursos facilitados para el desarrollo del proyecto y a mis compañeros de equipo quienes me han respaldado en todo momento.

Resumen

Resumen — El aumento en el volumen de información que tienen que manejar diariamente las empresas provoca que procesos tradicionales de tratamiento y almacenamiento no resulten eficientes a la hora de manejar grandes cantidades de datos. Payvision es un proveedor independiente de soluciones de pago, especializado en el procesamiento de pagos internacionales con tarjeta para el mercado de comercio electrónico. La aplicación web de reporting de Payvision tiene como objetivo mostrar información a los usuarios sobre las transacciones de comercio electrónico que han sido procesadas por la plataforma. El proceso de tratamiento y consulta de datos resulta muy lento cuando se tienen que manejar volúmenes grandes de transacciones, lo que provoca que la experiencia de usuario a través de la aplicación web no sea adecuada.

El objetivo de este estudio es diseñar, implementar y evaluar una solución que pueda dar soporte a este volumen de datos, ofreciendo al usuario información de sus transacciones en tiempo real. Maximizando el rendimiento de cada pieza del sistema para conseguir flujo de procesamiento y almacenamiento de información continuo. Mediante un sistema de procesamiento en streaming distribuido como Apache Flink podemos tratar de forma inmediata todas las transacciones que son procesadas por el sistema de pagos de Payvision. El sistema de almacenamiento debe ser capaz de responder de forma eficiente a la hora de realizar búsquedas y consultas. Elasticsearch nos permite disponer de un motor de búsqueda y análisis distribuido de alto rendimiento a través del cual podemos visualizar los datos en tiempo real.

El resultado de este trabajo es un sistema de procesamiento y reporting de transacciones de alto rendimiento, distribuido y eficiente, capaz de proveer un servicio de consulta de transacciones procesadas por el usuario en tiempo real.

Palabras clave — Tiempo real, Big Data, procesamiento en streaming, Apache Flink, rendimiento, Elasticsearch, e-commerce.

Abstract

Abstract — The growing volume of information that companies have to handle daily causes traditional processing and storing processes to be inefficient when handling large amounts of data. Payvision is an independent payment solution provider specialized in global credit card processing for the e-commerce market. The Payvision reporting application shows information about user e-commerce transactions that have been processed by the payment platform. Transforming and visualizing these data can be a slow process when the volume of transactions is considerably big. This has a negative impact on the web application's user experience during navigation.

The main goal of this project is to design, implement and evaluate a solution that can solve this problem, offering to the user near real-time information about processed transactions. This solution maximizes performance of each piece of the system, to achieve a continuous processing and storing transaction flow. Using a stream-processing system like Apache Flink, we can handle all transactions received from the Payvision payment platform as they arrive. In order to get efficient data searches and querying, Elasticsearch —a distributed, search and analytics engine— provide us with a scalable and near real-time search environment.

The result of this project is a distributed, scalable and efficient stream-processing and reporting system, capable to provide a near real-time search platform for e-commerce transactions.

Key words — Real time, Big Data, stream-processing, Apache Flink, performance, Elasticsearch, e-commerce.

Índice general

Índice de figuras

Índice de tablas

1

Introducción y Objetivos

El objetivo del proyecto se centra en el desarrollo de un sistema de procesamiento que pueda afrontar problemas de tratamiento, análisis y consumo de datos provenientes de sistemas externos con tecnologías que permitan el procesamiento distribuido de grandes volúmenes de información. El alcance deseado es la inclusión de este tipo de tecnologías en una compañía real dedicada al comercio electrónico que maneja grandes cantidades de datos, con el objetivo de almacenar, procesar y visualizar grandes cantidades de transacciones de forma eficiente. Payvision es un proveedor independiente de soluciones de pago, especializado en el procesamiento de pagos internacionales con tarjeta para el mercado de comercio electrónico y que ofrece a bancos adquirentes, proveedores de servicio de pago, ISOs[1] y sus comerciantes una plataforma única de procesamiento de pagos segura.

El sistema de reporting de Payvision muestra a los usuarios de la plataforma todas las transacciones económicas asociadas a su cuenta en el sistema, este sistema se compone de una aplicación web que expone las transacciones provenientes del sistema de procesamiento de pagos de Payvision. Todas las transacciones que sean procesadas por la plataforma deben poderse visualizar en el sistema de reporting, además, se muestra información analítica del histórico de pagos mediante un dashboard con diferentes diagramas de puntos y ratios calculados sobre diferentes campos. A través de esta aplicación web se pueden consultar transacciones aplicando numerosos filtros, los más comunes consisten en filtrado por fecha, cuenta de usuario, cantidad monetaria o método de pago utilizado.

Todos estos datos sobre transacciones llegan desde diferentes fuentes, dependiendo del modelo de negocio que exista por detrás. Al tratarse de una solución de pagos *omnichannel* existen numerosos tipos de transacciones, cada una con características diferentes dependiendo del método de pago utilizado. Cada uno de estos tipos de

[1] ISO: Independent Sales Organization

transacciones debe ser tratado para mostrar todas las transacciones de forma homogénea y fácilmente interpretable por los clientes en el sistema de reporting. Para ello, los datos son tratados antes de ser mostrados en la aplicación web, un proceso de ETL (*Extract, Transform and Load*) se encarga de cargar y tratar la información antes de almacenarla para ser consumida por los clientes.

Actualmente numerosas compañías cuentan con sistemas de tratamiento de datos, bien sea para sustentar otros sistemas o para almacenarlos en un sistema de *data warehouse*. Para realizar este tratamiento los procesos de ETL son comunes en la industria, estos procesos consisten en la extracción de los datos desde otros sistemas, transformar los datos (aplicando reglas de negocio, limpiando nulos, filtrando campos, aplicando validaciones, etc.) y finalmente cargando los datos en almacenes de datos o repositorios para alimentar otras aplicaciones. Este proceso puede llegar a ser bastante lento cuando se requiere tratar con grandes volúmenes de datos.

Una vez que los datos han sido procesados se almacenan en un servidor de base de datos para ser consumidos desde la aplicación web, debido a que el volumen de datos cada vez es más grande las consultas, filtros y analíticas que se requieren para alimentar la aplicación que ve el cliente cada vez son más lentas lo que provoca que la experiencia de usuario se vea mermada, convirtiendo la aplicación web en una herramienta poco práctica para el usuario.

Debido al crecimiento de la compañía se espera que el volumen de transacciones que pasen por la plataforma se vea incrementado notablemente durante el próximo año, obligando al sistema a ser capaz de manejar todo ese volumen extra. Al igual que otras partes de la plataforma, el sistema de reporting debe adaptarse a esta situación y para ello es necesario un cambio en la arquitectura que soporte un volumen mucho mayor. Este estudio pretende cambiar el paradigma de procesamiento y almacenamiento de los datos reemplazando antiguos procesos de ETL y sistemas de almacenamiento que no están preparados para manejar adecuadamente grandes volúmenes de información.

No solo se pretende manejar adecuadamente los datos relativos a transacciones, sino que se pretende ofrecer al usuario un sistema de reporting donde pueda ver la información de transacciones en tiempo real. Esto supone cambiar el procesamiento por lotes tradicional por un procesamiento en streaming donde el tratamiento de los datos y su almacenamiento para ser consultados se produzca de forma inmediata una vez recibida la transacción desde el sistema de procesamiento de pagos. Además, se pretende que la experiencia de usuario sea lo más fluida posible pudiendo realizar cualquier tipo de consulta sobre los datos asociados a sus transacciones sin apenas penalización de tiempo, para ello se debe reemplazar el sistema de almacenamiento actual por un motor de búsqueda que ofrezca un rendimiento aceptable y tiempos de respuesta mínimos.

Para ofrecer una solución de este tipo valoraremos las diferentes opciones tecnológicas de las que disponemos hoy en día, tanto para el procesamiento en streaming de transacciones como para el almacenamiento y consulta de estas de forma eficiente. Además, se realizaran pruebas de rendimiento para evaluar el sistema desarrollado y su capacidad de respuesta para diferentes volúmenes de datos.

2

Estado del Arte

Los requisitos básicos cuando trabajamos con datos masivos tanto estructurados como desestructurados son los mismos que cuando se trata con conjuntos de datos de cualquier tamaño, sin embargo, el volumen, la velocidad de procesamiento y las características de estos conjuntos de datos deben tenerse en cuenta a la hora de diseñar nuevas soluciones y a la hora de elegir la tecnología apropiada. Los requisitos computacionales asociados a este nuevo paradigma en términos de rendimiento, capacidad computacional, escalabilidad y valor de este tipo de computación se han expandido abrumadoramente durante los últimos años.

Debido al tipo de información que se procesa en este tipo de sistemas, el reconocimiento de tendencias o cambios en los datos a lo largo del tiempo son incluso más importantes que los valores individuales. Visualizar los datos de forma apropiada puede ser muy valioso a la hora de detectar tendencias u obtener información de estos grandes conjuntos de datos.

2.1. Tecnologías de mensajería

Las diferentes tecnologías de mensajería, conocidas como colas de mensajes, brókers de mensajería o herramientas de mensajería, son utilizadas para proveer una comunicación asíncrona entre sistemas y desacoplar procesos (desacoplando los emisores y receptores), escalado de sistemas o manejo de *backpressure*[1]. Cuando trabajamos con sistemas de procesamiento en streaming e intervienen diferentes sistemas en el proceso es común ver estas herramientas como fuente o destino de datos. Todas ellas sirven para el mismo propósito, pero cada una tiene características diferentes, enfocándose o especializándose

[1]Control del sistema en situaciones de sobrecarga

en aspectos como rendimiento, fiabilidad, arquitectura, etc.

En la actualidad existen múltiples sistemas de mensajería. Vamos a analizar dos de los sistemas más modernos y populares hoy en día: RabbitMQ y Apache Kafka.

2.1.1. RabbitMQ

RabbitMQ [1] es un bróker de mensajería que implementa varios protocolos de mensajería. Fue uno de los primeros sistemas de mensajería de código abierto en alcanzar un nivel de madurez razonable, librerías de cliente, herramientas para desarrollo y documentación de calidad. Originalmente implementa el protocolo AMQP (Advanced Message Queuing Protocol), lo cual hace posible un escenario multiplataforma y flexible al tratarse de un protocolo soportado por múltiples lenguajes, sin estar ligado a estándares como JMS que limitaban a las aplicaciones que no estuvieran escritas en Java.

RabbitMQ es un sistema de mensajería de propósito general, bueno para para distribuir un mensaje a múltiples consumidores o para distribuir carga de trabajo entre múltiples trabajadores. Ofrece gran variedad de características como entrega fiable, enrutamiento, federación, alta disponibilidad y herramientas de administración. Es utilizado para aplicaciones que requieran consistencia y control sobre los mensajes.

2.1.2. Apache Kafka

Apache Kafka [2] fue desarrollado en Scala e iniciado en LinkedIn como una herramienta para conectar múltiples sistemas.Al tratarse de una arquitectura distribuida, era necesario mejorar características como integración de los datos o procesamiento en streaming en tiempo real, rompiendo las diferencias con enfoques monolíticos a estos problemas. Kafka ha sido bien aceptado por el ecosistema de productos de la Apache Software Foundation, siendo particularmente útil en arquitecturas orientadas a eventos.

Kafka ha sido diseñado y enfocado a escenarios donde exista un procesamiento en streaming. Recientemente ha incorporado Kafka Streams, que se posiciona como alternativa a otras plataformas como Apache Spark, Apache Flink, Apache Beam/Google Cloud Data Flow, etc. Es un sistema de mensajería que encaja bien para flujos con enrutamiento complejo y alta carga de mensajes, además, mantiene un almacenamiento durable de los mensajes dejando el control de consumo a los clientes (pudiendo ser reconsumidos en cualquier momento).

RabbitMQ puede conseguir muchas de las características de Kafka a través software adicional, por ejemplo, suele ser utilizado junto con Apache Cassandra cuando se requiere mantener la historia de mensajes o las aplicaciones que los consuman necesiten una cola 'infinita'. RabbitMQ soporta oficialmente multitud de lenguajes, además, las bibliotecas de cliente son maduras y están bien documentadas.

En términos de seguridad y operaciones RabbitMQ provee mejor soporte, a través del plugin de administración nos provee una API HTTP, UI para administración y

monitorización y herramientas CLI para realizar operaciones. Además del soporte para TLS, RabbitMQ dispone de RBAC (role-based access control) sobre un almacén de datos propio, LDAP o proveedores externos basados en HTTPS y admite la autenticación basada en certificado en lugar de nombre de usuario y contraseña.

Apache Kafka fue diseñado para ofrecer alto rendimiento, por lo que para sistemas con alta carga de mensajes por segundo es la mejor opción. A pesar de que las dos herramientas pueden correr sobre un clúster, Kafka ha demostrado poder manejar volúmenes muy altos de carga con varios millones de mensajes por segundo, RabbitMQ tan solo ha alcanzado entrono al millón de mensajes por segundo utilizando alrededor de treinta nodos [3].

2.2. Sistemas de procesamiento en streaming

El procesamiento en streaming es un paradigma de programación equivalente a la programación orientada a flujos, programación orientada a eventos y programación reactiva, normalmente apoyada en el procesamiento en paralelo de cada una de las entradas de datos al sistema.

El modelo de streaming está pensado para analizar y procesar los datos en tiempo real lo que nos permite obtener resultados inmediatos. Esta es una de sus principales ventajas, pero además existen otras razones por las que utilizar este paradigma:

- En ciertas ocasiones tratamos con datos que no tienen un final, es decir, se van generando continuamente y sería inapropiado tratarlos por lotes ya que su naturaleza es continua. Por ejemplo tratando los eventos en ventanas de tiempo y el mantenimiento de sesiones veremos que nos ofrecen un gran potencial cuando tratamos con este tipo de datos.
- El procesamiento en streaming encaja a la perfección cuando trabajamos con series de tiempo o detección de patrones en tiempo. Esta tarea sería más costosa y menos fluida si trabajamos por lotes.
- Cuando trabajamos con lotes, tenemos que primero construir el lote y luego procesarlo todo de golpe. Esto hace que se creen picos de carga en el sistema mientras que con un procesamiento continuo se requiere menos sobrecarga en momentos aislados de tiempo.
- Existen escenarios donde el volumen de datos es tan grande que no es posible almacenarlos de forma fácil, por lo que el procesamiento de estos datos en streaming sin tener que retenerlos en el sistema agiliza el proceso.
- Finalmente, mucha información se genera de forma continua y no en lotes, por ejemplo transacciones financieras, actividad de usuarios en cualquier sistema y sobre todo eventos generados por sistemas IoT[2]. El procesamiento en streaming es una

[2]IoT: Internet of Things

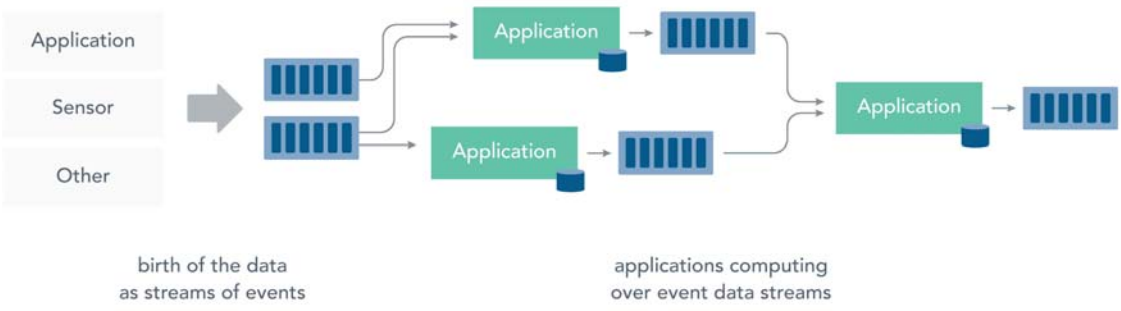


Figura 2.1: Procesamiento en streaming.

forma mucho más natural de trabajar con esta información y existen multitud de casos de uso donde se puede aplicar.

No olvidemos que este modelo no está pensado para todos los casos. Una buena regla para saber cuando debemos usarla es sobre todo cuando no necesitemos hacer múltiples iteraciones sobre los datos, es decir que se puedan tratar o procesar una sola vez sin tener que volver a consultarlos para procesarlos iterativamente. Si es suficiente con un simple tratamiento de los datos y solo se necesita localidad temporal de los datos (el sistema sólo necesita acceder a datos recientes) entonces encaja bien con el modelo de streaming.

Este tipo de sistemas se nutren de eventos generados por sistemas externos. Un caso habitual es el uso de colas de mensajes como fuentes de datos, desde las que se van consumiendo los eventos generados y que van a ser procesados por el sistema.

En 2016 surgió una nueva idea llamada Streaming SQL. Utilizamos SQL para realizar consultas sobre un flujo de datos. En la actualidad hay muchos sistemas que utilizan este mecanismo para obtener información de los datos que están siendo procesados, por ejemplo WSO2 Stream Processor y SQLStreams han soportado SQL sobre streams desde hace más de cinco años, Apache Storm y Apache Flink dieron soporte a SQL en 2016 mientras que Apache Kafka o Apache Samza lo incluyeron en 2017 [4]. Además de facilitar el uso de obtener información de flujos continuos de información facilita a los desarrolladores y usuarios utilizar su conocimiento de consultas SQL para incorporar el análisis de datos de forma sencilla en sus sistemas o aplicaciones.

El análisis de estos datos es lo que realmente nos aporta valor. Podemos dividir las herramientas de análisis de datos en cuatro categorías [5]:

- Análisis descriptivo: Nos informa que está pasando, es habitual utilizar herramientas de reporting o visualización con las que ver cual es el estado en un determinado momento del tiempo. Este tipo de técnicas son las menos sofisticadas.
- Análisis de diagnóstico: Este tipo de técnicas explican porque sucedió algo, se trata de técnicas más avanzadas que permiten profundizar en los datos y explicar que originó cierta situación o la causa raíz de una determinada situación.

- Análisis predictivo: Son las más habituales en la actualidad, este tipo de herramientas nos permiten realizar mediante algoritmos avanzados un pronóstico sobre lo que sucederá en el futuro. A menudo se utilizan técnicas de inteligencia artificial y aprendizaje automático.
- Análisis prescriptivo: Se trata de un paso más al análisis predictivo, nos indica que acciones tomar para lograr ciertos resultados, estas herramientas requieren de técnicas muy sofisticadas de inteligencia artificial, debido a ello son las más difíciles de conseguir.

Apache Hadoop [6] es considerado el framework por excelencia cuando hablamos de Big Data o manejo de datos a gran escala. Internamente utiliza MapReduce como motor de procesamiento, pero desde la versión 2.0 de Hadoop y la introducción de YARN como gestor de recursos es posible ejecutar otros motores o frameworks sobre un clúster de Hadoop. Este es el caso de Apache Spark que se ejecuta aprovechando toda la arquitectura que Hadoop nos ofrece. Eso hace que este tipo de sistemas hayan conseguido gran flexibilidad e interoperabilidad entre ellos.

Todos estos sistemas se basan en la ejecución de ciertas operaciones sobre los datos. Algunos lo hacen procesando los datos por lotes y otros a través de un procesamiento continuo del flujo entrante de información como los sistema de procesamiento en streaming vistos en la sección anterior. Los sistemas de procesamiento por lotes han sido los predominantes en los sistemas tradicionales de Big Data ya que manejan muy bien volúmenes grandes de datos, frecuentemente usados en datos históricos almacenados previamente.

Apache Hadoop es un framework que originalmente estaba pensado para manejar exclusivamente datos por lotes, sin embargo, la arquitectura de Hadoop se ha reestructurado para ofrecer distintas capas o componentes:

- HDFS: Hadoop Distributed File System es la capa que gestiona el almacenamiento y la replicación de los datos a través de los nodos. HDFS asegura que los datos se encuentren disponibles a pesar de fallos en ciertos nodos del sistema.
- YARN: Son las siglas de su nombre en inglés: Yet Another Resource Negotiator. Es el componente que gestiona y coordina los recursos de los que dispone el clúster de Hadoop. Con YARN es posible ejecutar tareas que en versiones anteriores no hubiera sido posible, abre la puerta a la implementación de motores alternativos a MapReduce actuando como una interfaz sobre el clúster de recursos.
- MapReduce: Es el motor nativo de procesamiento en lotes de Hadoop.

El modelo de procesamiento de datos de Hadoop viene del motor de MapReduce. Esta técnica sigue el algoritmo de map, shuffle y reduce utilizando tuplas de clave-valor. El funcionamiento básico consiste en la lectura del dataset distribuido almacenado en el sistema de ficheros HDFS, a cada una de esas piezas distribuidas de información se aplica una determinada acción computacional y se redistribuyen los resultados intermedios en

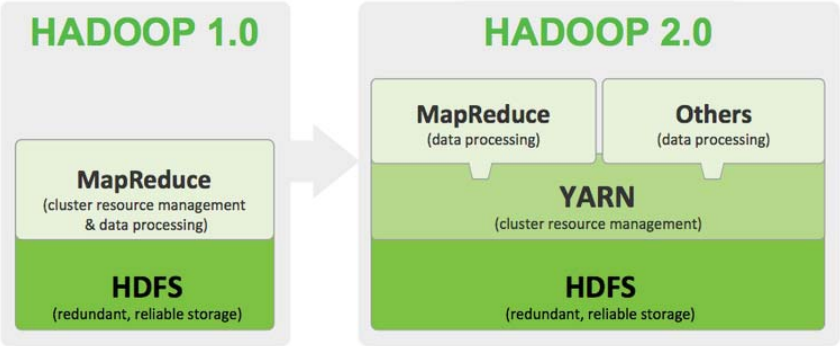


Figura 2.2: Hadoop 1.0 vs. 2.0.

conjuntos agrupados por clave. Finalmente se aplica la función de reducción combinando los resultados de cada procesamiento individual y calculando el resultado final, el cual se almacena de vuelta a HDFS.

Debido a que esta técnica hace un uso extensivo de lectura y escritura por cada tarea hace que tienda a ser un proceso lento. Por otra parte, ya que el almacenamiento es uno de los recursos más abundantes en un servidor hace que sea menos caro que otras alternativas que hagan uso del trabajo en memoria.

Los sistemas de procesamiento en streaming computan los datos según llegan al sistema. Esto requiere de un modelo diferente de procesamiento al que utiliza MapReduce por lotes de datos. Al contrario que los sistemas por lotes donde definimos qué operaciones vamos a aplicar sobre todo el conjunto de datos, el procesamiento en streaming define operaciones sobre elementos individuales a su llegada al sistema.

Los set de datos que manejan este tipo de sistemas son considerados ilimitados o infinitos ya que no tienen un principio ni un final definidos. Esto hace que debamos tener en cuenta algunas implicaciones importantes como que el total de los datos está definido como los datos que hayan llegado hasta el momento y que los datos van a tener localidad temporal.

En estos sistemas los datos se procesan uno a uno (streaming real) o de pocos en pocos (micro-batches) con un mantenimiento mínimo de estado entre registros. En la mayoría de ellos se utiliza en un paradigma de procesamiento funcional.

2.2.1. Apache Storm

Apache Storm [7] es un framework de procesamiento en streaming caracyerizado por su baja latencia. Storm es realmente rápido y es capaz de procesar grandes volúmenes de datos a alta velocidad.

El flujo de ejecución de un aplicación en Storm está diseñado siguiendo una topología en forma de gráfico acíclico dirigido (DAG) con spouts (surtidores) y bolts (consumidores) que actúan como los vértices del gráfico. Los arcos en el gráfico se denominan flujos, dirigen

los datos de un nodo a otro. Básicamente esta topología describe las fases o pasos que cada elemento que entra en el sistema debe seguir. Esta estructura de topología es similar a un flujo de MapReduce, con la diferencia de que los datos se procesan en tiempo real y no en lotes individuales.

Esta topología se compone de:

- Streams: Flujos de datos convencionales, flujo de elementos que continuamente llegan al sistema.
- Spouts: Fuentes de datos, estas pueden ser APIs, colas, etc. que producen datos.
- Bolts: Representan consumidores de datos que se nutren del flujo principal y operan sobre ellos. Estos consumidores están conectados a los generadores de datos y estos a su vez están conectados entre ellos para crear flujos de procesamiento. Esta estructura forma una topología de nodos conectados unos a otros para formar un sistema que gestione el flujo de datos.

La idea principal es simple, definir pequeñas piezas con un comportamiento específico y operaciones sobre los datos que formen un sistema. Storm garantiza procesamiento "at-least-once", esto quiere decir que los elementos no van a sufrir pérdidas una vez entren al sistema. Si lo que buscamos es procesamiento "exactly-once", donde además conseguimos que no se produzcan duplicados, y un procesamiento con mantenimiento de estado tenemos que utilizar una herramienta llamada Trident. Con el uso de Trident añadimos mucha más sobrecarga al proceso, incrementando la latencia, añadiendo estado e implementando un modelo de micro-batches en vez de streaming puro.

Es recomendable utilizar Core Storm, es decir, sin Trident, para no sobrecargar el proceso, sin embargo Trident garantiza que cada elemento es procesado una sola vez y es realmente útil a la hora de detectar duplicados, además es la única opción que tenemos si queremos mantener estado entre eventos.

Storm es una buena opción para sistemas que necesiten un manejo de los datos cerno a tiempo real y cuando se busca una latencia muy baja de procesamiento. Sin embargo, debe tenerse en cuenta que si se desea trabajar con estado estamos añadiendo demasiada sobrecarga (con la utilización de Trident), por lo que veremos que hay alternativas como Apache Flink que pueden ofrecer una latencia muy baja incluso manteniendo estado entre eventos o elementos del flujo de datos.

Si queremos ejecutarlo sobre Hadoop, Storm puede integrarse con Hadoop YARN haciendo que sea posible desplegarlo y ejecutarlo aprovechando las ventajas de un clúster de Hadoop. Además como muchos otros frameworks de este tipo soporta el uso de diferentes lenguajes de programación.

2.2.2. Apache Samza

Apache Samza [8] es un framework de procesamiento en streaming que está fuertemente unido a Apache Kafka. Mientras que Kafka es muy utilizado en sistemas de procesamiento en streaming, Samza está diseñado para aprovechar todas las ventajas de arquitectónicas de Kafka, tolerancia a fallos, buffering y almacenamiento de estado. Samza usa YARN, por lo que es necesario Hadoop para su ejecución (al menos HDFS y YARN).

Samza utiliza la semantica de Kafka para definir la forma en que el flujo de datos es manejado:

- Topics: Cada flujo de datos que entra en Kafka es llamado topic (tema o asunto), básicamente es un stream de datos relacionados entre sí y de características similares y a los que un consumidor se puede subscribir.

- Particiones: Con el objetivo de distribuir un conjunto de datos entre los nodos Kafka divide los mensajes entrantes en particiones. Estas particiones garantizan que mensajes con la misma clave van a la misma partición. Las particiones además garantizan ordenación.

- Brokers: Cada nodo que compone Kafka se denomina broker.

- Productor: Cualquier componente que escribe o suministra datos a Kafka es llamado un productor, estos productores asignan las claves que son utilizadas para las particiones.

- Consumidores: Son componentes que leen datos de los topics generados, estos tienen la responsabilidad de mantener la información de su propio conjunto de datos, por lo que tiene que manejar los registros que procesen en caso que de ocurra un fallo en el sistema.

Debido a que Apache Kafka trabaja con logs inmutables de información Samza maneja streams inmutables, esto quiere decir que cualquier mutación o transformación en los datos crea un nuevo stream sin afectar al stream original.

Además de los sistemas que trabajan puramente en streaming existen sistemas que pueden manejar tanto datos en streaming como por lotes. Este tipo de frameworks simplifican la complejidad de los distintos tipos de procesos proveyendo una interfaz común para trabajar con ambos tipos de datos.

Apache Spark y Apache Flink son dos de los principales frameworks que podemos encontrar actualmente, mientras que otras tecnologías se centran proveer una solución específica a determinados casos de uso estos intentan ser una solución más general para el procesamiento de datos sea cual sea el tipo de estos.

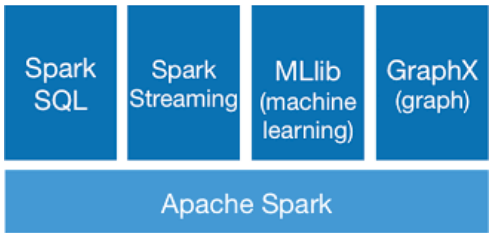


Figura 2.3: Componentes de Apache Spark.

2.2.3. Apache Spark

Apache Spark [9] es un framework avanzado para el procesamiento por lotes de datos con capacidad de ofrecer procesamiento en streaming. Aunque está construido siguiendo los mismos principios de Hadoop se centra principalmente en optimizar la velocidad de procesamiento ofreciendo computación y manejo de datos en memoria.

Puede desplegarse y ejecutarse de forma independiente en clúster aunque se suele desplegar sobre un clúster de Hadoop sustituyendo al motor de MapReduce.

Al contrario que MapReduce, Spark realiza todo el flujo de procesamiento de datos en memoria, tan solo interactuando con la capa de almacenamiento para la carga inicial de datos y para la persistencia final de los datos, manejando todos los resultados intermedios en memoria.

Además se realizar todo el procesamiento en memoria, Spark nos provee optimización complementaria analizando todo el flujo de operaciones mediante la creación de un grafo acíclico dirigido (DAG) y coordinando todas las tareas de forma inteligente.

Para la ejecución de operaciones en memoria por lotes Spark utiliza un modelo de datos que llama Resilient Distributed Datasets, o RDDs, para trabajar con los datos. Se trata de estructuras de datos inmutables que existen en memoria y que representan colecciones de datos. Toda operación que se ejecuta sobre un RDD produce otro RDD. Estos RDDs son usados por Spark para mantener tolerancia a fallos sin la necesidad de persistir en disco después de cada operación.

Spark está compuesto por diferentes piezas cada una con un objetivo específico, Spark Core es el núcleo de todo el sistema, encargado de proveer control y planificación de tareas distribuidas y funcionalidad básica de I/O, todo esto expuesto a través de una API para diferentes lenguajes y basada en la abstracción RDD para el manejo de datos.

- Spark SQL: este componente introduce el concepto de DataFrames, una abstracción de datos que provee soporte para trabajar con datos estructurados o semiestructurados. Además provee un DSL para manejar DataFrames en Scala, Java y Python y soporte para consultas SQL.

- Spark Streaming: a través de este componente conseguimos un mecanismo para trabajar con flujos de datos continuos, mediante el uso de mini-lotes de datos.

- MLlib Machine Learning: es un framework sobre Spark Core que nos permite realizar tareas de machine learning de forma distribuida. Muchos de los algoritmos de machine learning y estadística han sido implementados y vienen incluidos en esta librería.
- GraphX: es un framework de procesamiento de grafos distribuido, basado en RDDs. Provee dos APIs, una para la implementación de algoritmos paralelos y otra más parecida al estilo de trabajo de MapReduce. Además cuenta con soporte completo para gráficos de propiedades.

Spark está diseñado para trabajar con sets de datos por lotes. Para ofrecer características similares a las que nos ofrecen los sistemas de procesamiento en streaming Spark implementa el concepto llamado micro-batches tratando flujos de datos continuos como una serie de lotes muy pequeños que pueden ser tratados mediante la implementación nativa de lotes.

La razón principal de usar Spark y no el motor de MapReduce tradicional de Hadoop es velocidad. Spark es capaz de realizar tareas de procesamiento de datos mucho más rápido debido a su estrategia de computación en memoria y a su avanzado planificador de grafos acíclicos dirigidos. Además una de sus mayores ventajas es su versatilidad. Spark puede ser desplegado por sí solo en un clúster independiente o sobre un clúster de Hadoop existente.

A pesar de que es posible aprovechar el sistema de procesamiento por lotes para manejar flujos de datos en streaming con el manejo de micro-batches de datos, esta no es la mejor alternativa para sistemas que requieran una latencia muy baja ya que cuando tiene que tratar con volúmenes grandes de datos se añade una sobrecarga extra a la hora de descargar el buffer de datos que conduce a un aumento significativo de la latencia.

Debido a que la memoria RAM es generalmente más cara que el almacenamiento en disco el coste de su uso es más caro que el de los sistemas basados en el uso de disco, sin embargo, incrementa dramáticamente el tiempo de realización de tareas que podrían tardar horas si no se realizan en memoria.

2.2.4. Apache Flink

Apache Flink [10] es un framework de procesamiento en streaming que también puede manejar tareas por lotes, simplemente considerando flujos de datos como un flujo finito y por lo tanto tratando el procesamiento por lotes dentro del procesamiento en streaming.

El enfoque de "stream-first" en este tipo de sistemas se conoce como arquitectura Kappa[3], en contraste con la ampliamente conocida arquitectura Lambda (donde las tareas por lotes son utilizadas como el principal método de procesamiento siendo los streams utilizados como complemento y con resultados tempranos pero poco afinados). En la

[3]Jay Kreps, *Questioning the Lambda Architecture*, véase https://www.oreilly.com/ideas/questioning-the-lambda-architecture, 2014.

arquitectura Kappa se simplifica el modelo orientando todo al streaming. Esto es posible gracias a que los motores de procesamiento en streaming se han vuelto más sofisticados.

Flink procesa los datos entrantes como un flujo continuo de información y procesando independientemente cada una de las partes que llegan al sistema. Gracias a su API con la que que manejar streams (DataStream API) es posible trabajar fácilmente con con flujos infinitos de datos y aplicar operaciones sobre ellos.

Los principales elementos de los que se compone Flink son:

- Streams: o flujos, son conjuntos de datos inmutables que fluyen a través del sistema
- Operadores: se trata de funciones que operan sobre los datos para producir otros streams de datos
- Sources: o fuentes de datos, son el punto de entrada de estos al sistema, estas pueden ser un fichero, escuchando a través de la red, una cola de mensajes, etc.
- Sinks: son el lugar donde los datos son dejados por el sistema una vez que el procesamiento en streaming finaliza, pueden ser ficheros, una base de datos, otro sistema de procesamiento de datos, etc.

Para poder recuperarse en caso de fallo del sistema, Flink, hace copias de determinados instantes durante el procesamiento, a través de estos datos se puede recuperar el estado de procesamiento de un momento anterior. Dependiendo del tipo de complejidad del problema Flink puede trabajar manteniendo un estado entre los datos que llegan al sistema y forman parte del flujo de datos.

Además Flink dispone de distintas nociones de tiempo con las que manejar más apropiadamente los datos que nos llegan:

Tiempo de procesamiento

Se refiere al instante de tiempo en el que se ejecuta una operación sobre un determinado dato. Cuando trabajamos con este tipo de tiempo, todas las operaciones relacionadas y en las que intervenga el componente de tiempo usarán el reloj del sistema como referencia de tiempo. Este tipo de tiempo es el más simple de manejar y no requiere coordinación entre streams y máquinas de procesamiento, nos ofrece el mejor rendimiento y la menor latencia, sin embargo en sistemas asíncronos y distribuidos de procesamiento no es determinista, ya que es susceptible de la velocidad a la que los eventos llegan al sistema (por ejemplo una cola de mensajes) y la velocidad a la que los datos fluyen por el sistema.

Tiempo del evento

Es el tiempo que cada evento o dato individualmente tiene asociado y que por lo general se establece cuando el dato se generó. Suele venir en sus metadatos y puede

ser leído por el sistema para dar una referencia de tiempo más ajustada. Por ejemplo podemos definir una ventana de procesamiento para un determinado rango de tiempo y todos los eventos que satisfagan la condición serán incluidos independientemente de cuando los elementos llegaron al sistema, ya que el instante de tiempo a tener en cuenta es el asociado al propio evento. Debido a que solo tiene en cuenta el tiempo del evento es posible corregir eventos que lleguen desordenados, retrasados o repetidos. Al alterar el tiempo que estamos manejando en el sistema debemos decirle al sistema cual es el tiempo con el que debe trabajar, esto lo hacemos definiendo marcas de tiempo a partir de los eventos entrantes, es decir, el tiempo va pasando de forma artificial y es definido por el tiempo de los eventos que llegan al sistema.

Tiempo de ingestión

Es el tiempo en el que el evento llega al sistema. Nada más ser extraído de la fuente de datos se le asigna el instante de tiempo actual y que será utilizado en las operaciones en las que se tenga en cuenta el tiempo del evento. Conceptualmente este tipo de tiempo se encuentra en un punto intermedio entre el tiempo del evento y el tiempo de procesamiento, comparado con el tiempo de procesamiento nos da resultados más previsibles pero más costosos, comparado con el tiempo del evento es menos robusto ya que no puede manejar eventos desordenados y tratarlos apropiadamente pero nos quita el trabajo de ir definiendo marcas de tiempo pudiendo utilizar el tiempo original del sistema.

Apache Flink es actualmente una de las mejores alternativas cuando se trata de frameworks de procesamiento en streaming. A pesar de que Spark ofrece un componente para procesamiento en streaming no es apropiado para determinados casos de uso debido a su arquitectura orientada a micro lotes.

El enfoque de stream-first de Flink ofrece baja latencia, alta carga de datos, y procesamiento en tiempo real elemento por elemento. Además Flink maneja varias variables por sí solo, por ejemplo maneja de forma independiente su propia memoria en vez de dejárselo al recolector de basura nativo de Java por motivos de rendimiento, tampoco requiere cambios y ajustes manuales cuando las características de los datos a procesar cambian como sucede en Spark.

Como se ve en la figura 2.4 Apache Flink es capaz de ofrecer un rendimiento muy superior a Apache Storm realizando una tarea de conteo de elementos distribuida.

Flink puede operar bien con otros componentes. Es posible ejecutarlo sobre un clúster de Hadoop, se integra con YARN, HDFS y sistemas como Kafka de forma simple. Flink puede incluso ejecutar tareas destinadas a su ejecución en otros sistemas de procesamiento como Hadoop y Storm a través de librerías que los hacen compatibles. También ofrece una interfaz web donde manejar tareas y ver el estado del sistema, donde además es posible visualizar el plan de ejecución para las tareas por ejecutar.

Con un rango muy amplio de opciones de procesamiento para sistemas con un manejo extenso de datos masivos y Big Data podemos concluir que para sistemas donde solo

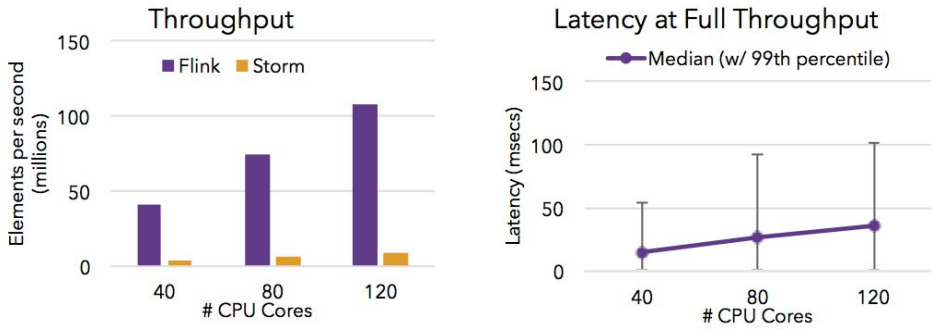


Figura 2.4: Rendimiento de Apache Flink vs. Apache Storm [11].

se necesita procesamiento por lotes de datos y no es vital el tiempo, Hadoop es una buena opción al tratarse de una alternativa más sencilla y barata en términos hardware. Cuando se trata de procesamiento sin estado y en streaming con una necesidad de baja latencia Storm. Es una buena alternativa. Si trabajamos con datos en streaming y por lotes, Spark nos ofrece alto rendimiento en operaciones sobre lotes y tiene un soporte muy amplio, librerías y herramientas que podemos utilizar. Flink nos ofrece por otro lado verdadero procesamiento en streaming y soporte para trabajar en lotes de datos, está altamente optimizado y puede ejecutar tareas escritas para otras plataformas similares, además nos provee baja latencia de procesamiento. Flink está en sus primeras etapas de adopción pero se está imponiendo a sistemas como Spark en determinados casos de uso [12].

2.3. NoSQL y almacenamiento de datos distribuido

Las bases de datos NoSQL (no sólo SQL) proveen un mecanismo para el almacenamiento de información y un modelo diferente de base de datos comparado con las tradicionales bases de datos tabulares o relacionales. Este tipo de bases de datos existen desde la década de los sesenta pero no fue hasta principios del sigo veintiuno cuando empezaron a popularizarse y a adquirir el nombre de "NoSQL". Impulsadas por la Web 2.0 y el incremento de la cantidad de datos que se manejan en sistemas de Big Data y aplicaciones de tiempo real han reemplazado en estos campos a las bases de datos relacionales debido a su alta escalabilidad. Se denominan no sólo SQL para recalcar el hecho de que también soportan lenguajes de consulta tipo SQL.

Los datos no son almacenados en estructuras fijas y no es habitual el uso de operaciones como JOIN, tampoco garantizan ACID (atomicidad, consistencia, aislamiento y durabilidad), pero a cambio de estas limitaciones proveen gran rendimiento, un diseño simple, más control para proveer alta disponibilidad y alta escalabilidad horizontal (lo cual es complicado en bases de datos relacionales).

Existen varios modelos de bases de datos NoSQL, los más extendidos son [13]:

Bases de datos clave-valor y orientadas a columnas

Las bases de datos clave-valor son el tipo más básico de bases de datos no relacionales. Cada elemento es almacenado como una clave que lo identifica y un valor asociado a esta. El conjunto de datos puede ser expresado como una colección de tuplas clave-valor. Este tipo de bases de datos proveen gran rendimiento a la hora de búsquedas por clave o rangos de claves, este modelo de consultas y actualizaciones esta muy optimizado. Si quisiéramos realizar búsquedas sobre otros valores tendríamos que crear y mantener indices adicionales para obtener un rendimiento apropiado. En las bases de datos orientadas a columnas estas son la unidad básica y consisten en un par clave valor, una familia de columnas sería el equivalente a una tabla, cada fila tiene un único identificador y tantas columnas como necesite para almacenar información. Cada fila puede tener un número de columnas diferente pero la familia debe ser declarada antes de usarse ya que típicamente representan la forma en la que se van a almacenar en disco. Básicamente en estos dos tipos de bases de datos existe una única forma eficiente de acceder a los datos, a través de su clave. Esto nos aporta un rendimiento abrumador pero esta pensado para casos de uso limitados y pueden conllevar un coste adicional de desarrollo para consultas más complejas.

Ejemplos: HBase, Redis, Cassandra, Aerospike, Dynamo

Bases de datos documentales

Las bases de datos orientadas a documentos son un subtipo del almacenamiento clave-valor. La diferencia radica en la forma en la que los datos son procesados; en este tipo de bases de datos se tiene en cuenta la estructura interna del documento a diferencia de las bases de datos clave-valor. Nos proveen la capacidad de consulta de cualquier campo dentro del documento encargándose del procesamiento e indexación de los datos cuando se añaden al sistema. Algunas bases de datos como MongoDB proveen un amplio conjunto de opciones de indexación para optimizar la forma de consultar los datos, como indices para texto, para información geoespacial, indices compuestos, etc. Algunos sistemas como Elasticsearch incluso analizan los datos y ofrecen un motor de búsqueda y de análisis en tiempo real.

Ejemplos: Elasticsearch, CouchDB, MongoDB, Couchbase

Bases de datos orientadas a grafos

Una base de datos en grafo utiliza estructuras de datos en grafo formados por un número finito de elementos y relaciones entre ellos para realizar consultas semánticas con nodos, aristas y propiedades como forma de representación y almacenamiento de datos. Determinados análisis que relacionen tipos o entidades son muy eficientes en este tipo de sistemas.

Ejemplos: Neo4j, OrientDB, InfiniteGraph, Sones

Además de estos existen otros muchos tipos de bases de datos NoSQL como bases de

datos multivalor, orientadas a objetos, tabulares, multimodelo, etc.

2.3.1. Elasticsearch

Elasticsearch [14] es un motor de búsqueda basado en Lucene [15]. Provee un motor de búsqueda de texto completo, distribuido y con capacidad de multi-tenencia con una interfaz web y almacén de datos orientado a documentos JSON. Esta desarrollado en Java, es de código abierto y bajo licencia Apache. Actualmente es el motor de búsqueda más popular en entornos empresariales seguido por Apache Solr [16], también basado en Lucene.

Apache Lucene es una librería que provee un motor de búsqueda de texto completo de alto rendimiento escrita en Java. Es muy apropiada para cualquier aplicación en la que se requiera búsquedas de texto completo ya que ofrece indexado escalable y de alto rendimiento sobre los datos (documentos) y una sencilla API a través de la cual realizar consultas. Lucene provee búsquedas sobre documentos. Un documento se compone de una colección de campos, cada uno de estos campos consiste en un nombre y uno o más valores. Estos campos pueden ser de tipo binario, numérico o de texto. Lucene provee múltiples formas de partir el texto en tokens. A partir de éstos Lucene es capaz de retornar un conjunto de documentos ordenados por relevancia dependiendo de los parámetros introducidos en la consulta.

Elasticsearch nos ofrece una búsqueda escalable, cercana al tiempo real y distribuida, lo que significa que los índices (o conjuntos de datos) pueden dividirse en partes y cada una de esas partes puede tener cero o varias réplicas. Los datos relacionados entre sí son almacenados en el mismo índice, el cual consiste en una o más partes y cero o varias réplicas. Una vez que un índice se ha creado, el número de partes en las que se divide no puede ser modificado.

En las siguientes secciones se detallan algunas propiedades de bases de datos NoSQL y cómo Elasticsearch las implementa:

No transaccional

A pesar de que Lucene si tiene el concepto de transacción, Elasticsearch no lleva a cabo transacciones en el sentido típico, por ejemplo, no hay manera de realizar rollback de una transacción que se ha ejecutado o no se puede indexar varios documentos en la misma transacción y esperar que se indexen todos o ninguno. Lo que se lleva a cabo internamente es una escritura previa (write-ahead-log) de las operaciones que se van a realizar para asegurar la durabilidad de las operaciones evitando realizar una transacción de Lucene con el coste que ello conlleva. La visibilidad de los cambios se produce cuando un indice se refresca, lo que sucede una vez por segundo.

Elasticsearch esta diseñado para ser rápido, llevar a cabo transacciones distribuidas conllevaría demasiado trabajo, aceptando que algunos de los datos que nos devuelva

pueden no tener los últimos cambios y que todo el mundo ve los mismos datos, Elasticsearch puede servir los datos desde caches lo que nos ofrece un rendimiento mucho mayor.

Esquema flexible

Elasticsearch no requiere que se especifique un esquema antes de insertar datos en el sistema, es capaz de inferir los tipos de un documento en formato JSON. El sistema maneja especialmente bien tipos numéricos, booleanos e instantes de tiempo o timestamps, para ingerir los datos de texto dispone de múltiples analizadores disponibles.

A pesar de que no es necesario definir un esquema de datos, es recomendable definir ciertas directrices para ayudar al sistema a llevar a cabo búsquedas y/o analíticas más eficientes.

Relaciones y restricciones

Elasticsearch es una base de datos orientada a documentos, el grafo completo de objetos que se quiere consultar necesita ser indexado, antes de indexar documentos estos deben ser *desnormalizados*, lo que incrementa el rendimiento de las consultas. Para llevar a cabo esta desnormalización (y evitar así el uso de uniones o JOINs) entre documentos los datos deben ser almacenados con toda la información asociada que se quiera consultar, esto hace que se requiera más espacio para su almacenamiento y que se dificulte su modificación.

Este tipo de bases de datos orientadas a documentos están diseñadas para almacenar la información de forma optimizada para su consulta.

Robustez

Una base de datos debe ser robusta, especialmente cuando se trata de sistemas vitales para el funcionamiento de la plataforma que se quiera construir. Idealmente una consulta costosa debería ser posible de ser cancelada para no interferir en el correcto funcionamiento del sistema. Desafortunadamente, Elasticsearch no es capaz de manejar excepciones de tipo `OutOfMemory`.

Elasticsearch esta pensado para ser rápido por lo que se debe proveer suficiente cantidad de memoria en clústeres de entornos productivos y no realizar operaciones de las que no sepamos que requisitos de memoria van a necesitar. Debido al tipo de sistema de almacenamiento del que trata y su especialización en rendimiento se asume que la cantidad de memoria disponible es abundante.

Distribuido

Elasticsearch esta diseñado para ser distribuido y ser fácilmente escalable para manejar cantidades masivas de datos. Debido a la naturaleza de los sistemas distribuidos implica que hay que ser capaces de manejar situaciones de error en una o varias partes del sistema. Los diferentes sistemas de bases de datos ofrecen diferentes cualidades, Eric Brewer sostiene que es imposible para un sistema de almacenamiento distribuido ofrecer mas de dos de estas tres garantías: consistencia, disponibilidad y tolerancia al particionado[4]. Elasticsearch es un sistema que ofrece consistencia y tolerancia al particionado, si tienes un sistema de solo lectura es posible conseguir un sistema con disponibilidad y tolerancia al particionado teniendo una configuración que requiera los mínimos nodos maestros posibles para evitar requerir un cuórum.

Cuando se trata de escalado, un índice puede ser dividido en una o más particiones, este debe ser particionado teniendo en cuenta el volumen que va a soportar. Cuantos más nodos se añadan al clúster, Elasticsearch se encarga de distribuir y redistribuir las particiones. Elasticsearch es muy sencillo de escalar.

Seguridad

Elasticsearch no dispone de métodos de autenticación ni autorización en su versión más simple. Se debe tener en cuenta que cualquiera que pueda conectar con el clúster de Elasticsearch tiene permisos para realizar cualquier acción. Sin embargo a través de X-Pack Security es posible securizar un clúster. Este paquete se puede utilizar para proteger los datos con métodos de autenticación pero además nos ofrece otras cualidades como comunicación segura, control de acceso basado en roles, filtro de IP, y auditoría.

X-Pack Security nos permite proteger un clúster Elasticsearch:

- Previniendo accesos no autorizados mediante control de acceso por contraseña, control de acceso basado en roles y filtro de IP.
- Preservando la integridad de los datos transmitidos mediante autenticación de las mensajes y encriptación SSL/TLS.
- Manteniendo un registro de auditoría con información sobre las acciones que se realicen sobre los datos.

Elasticsearch no suele ser usado como única forma de almacenamiento, sino que suele ser utilizado en conjunción con otras bases de datos, por ejemplo, un sistema de base de datos con un enfoque más fuerte en integridad sobre los datos, corrección y robustez y transacionalmente actualizable el cual va actualizando de forma asíncrona el modelo en Elasticsearch, siendo este utilizado para realizar consultas de los datos. Sin embargo

[4]Eric Brewer, *Towards Robust Distributed Systems*, véase https://people.eecs.berkeley.edu/ brewer/cs262b-2004/PODC-keynote.pdf, 2000.

puede ser utilizado como sistema principal de almacenamiento cuando las limitaciones descritas no suponen un problema. Un buen ejemplo es Logstash, un sistema de logs donde cada entrada es escrita una vez y leída múltiples veces, sin necesidad de transacciones ni restricciones de integridad.

Existen otros sistemas que proveen sistemas de búsqueda de texto completo como Postgres, MySQL, MongoDB, Riak, etc. La principal diferencia con estos sistemas es el rendimiento que ofrece. Como se ha mencionado Elasticsearch realiza un uso extensivo de caches sin preocuparse de control de concurrencia y otras limitaciones asociadas a sistemas relacionales o que implementen integridad relacional [17].

3

Especificación de requisitos

El aumento del número de transacciones que procesa Payvision diariamente se ha visto incrementado en los últimos meses, y además se espera que este volumen se vea incrementado drásticamente durante este año. Las herramientas de procesamiento y visualización de estos datos deben poder manejar grades volúmenes de información. Para ello es necesario mejorar el rendimiento del sistema para que pueda manejar estos volúmenes de información de forma eficiente.

3.1. Dominio del problema

El sistema de reporting de Payvision muestra a los usuarios información sobre las transacciones que han sido procesadas a través de la plataforma, tanto transacciones de comercio electrónico, point of sale (POS) o pagos alternativos. Debido al incremento del volumen de transacciones en los últimos años el sistema de procesamiento y normalización de datos es cada vez es más lento. Este proceso consiste en una lectura diaria de un fichero que contiene todas las transacciones que se han producido durante el día anterior.

El problema reside en la gran cantidad de tiempo que dedican nuestros procesos de ETL en procesar este archivo y sobre todo cuando es necesario volver a reprocesar todo el histórico debido a nuevos requerimientos para mostrar nuevos datos o corrección de los actuales. Este procesamiento consiste en la limpieza y análisis de las transacciones que llegan de las diferentes fuentes; el modelo de seguridad, la identificación de transacciones y los detalles asociados a cada transacción deben ser controlados por el ETL para ser almacenados uniformemente en la base de datos. Además, las consultas y filtros que se solicitan desde la aplicación son bastante lentas debido a la cantidad de datos que se tienen que manejar, en especial agregados y ordenaciones.

El volumen de este fichero crece diariamente ya que el volumen de transacciones que procesa la empresa es cada vez mayor. Debido a cambios estratégicos de la compañía está previsto que el volumen de transacciones POS aumente de forma brusca durante los próximos meses, por lo tanto se está reestructurando la arquitectura y la forma de manejar el procesamiento de éstas. Con lo que respecta a la aplicación de reporting está planificado que cambie la fuente de consumo de transacciones a una cola de mensajes en la que los sistemas de procesamiento van a ir dejando las transacciones que el sistema vayan procesando.

Este cambio de origen de datos hace que se plante un cambio en la forma de gestionar las transacciones para su posterior visualización en la aplicación web, al tratarse de un flujo continuo de datos y no un fichero (procesamiento en batch) se propone utilizar un sistema de procesamiento en streaming para que la ingestión y visualización de transacciones se produzca en tiempo real.

Para ofrecer a nuestros clientes (merchant y resellers) un sistema más eficiente y que puedan ver sus transacciones de forma más inmediata y fluida se propone el cambio de paradigma a la hora del proceso de ETL utilizando tecnologías que puedan procesar en streaming todas las transacciones que se vayan produciendo y su posterior consulta y filtrado.

3.1.1. Descripción de la situación actual

El sistema actual es capaz de recibir un fichero diario con todas las transacciones que se produjeron el día anterior y almacenarlas de forma estructurada en la base de datos. Si no hay demasiada carga y no se requiere procesar toda la historia el sistema puede funcionar con un procesado una vez al dia sin demasiado impacto ya que no hay (de momento) un requisito de inmediatez en la presentación de transacciones.

La base de datos consiste en un modelo relacional con lo que se consigue alta integridad de los datos, si bien debido a su naturaleza algunas consultas son demasiado pesadas y afectan negativamente a la experiencia de usuario en la web. Este tipo de consultas es cada vez más un requisito fundamental a la hora de mostrar los datos.

En el caso de que haya un problema en el código o se necesite mostrar información que actualmente no se esté tratando es necesario volver a reprocesar todo el histórico de transacciones, algo que con el sistema actual requiere días. Si la carga de datos aumenta todas estas operaciones se van a volver imposibles de realizar en un tiempo apropiado.

3.1.2. Modelos de procesos de negocio actuales

El sistema actual permite visualizar las transacciones que han sido procesadas por la plataforma de pagos de Payvision. Los merchants o resellers pueden ver las transacciones asociadas a su cuenta y el balance de transacciones que han sido aprobadas y cuáles rechazadas además de otro tipo de operaciones como devoluciones o movimiento de dinero

de vuelta al cliente.

A continuación se detallan los principales actores del sistema, aquellos que guardan una relación con este y que demandan cierta o gran parte de la funcionalidad:

- Merchant y resellers: Persona o compañía que utiliza la aplicación de reporting para consultar la actividad financiera y de transacciones relacionada con su negocio.
- Personal de soporte: Persona involucrada en dar soporte a los clientes de la aplicación en cuestiones técnicas o de negocio.
- Product owner: Persona encargada del producto, responsable de conocer las necesidades de negocio y las necesidades de los clientes. Debe transmitir las necesidades de los clientes y la funcionalidad a desarrollar.
- Desarrolladores de la aplicación: Personas encargadas del desarrollo software de la aplicación.
- Personal de ventas: Personas encargadas de la venta del producto.

Actualmente cuando un merchant o reseller quiere consultar la información relacionada con las transacciones que ha procesado con nuestro proveedor de pagos se conecta a la aplicación de reporting web, en ella puede ver información sobre el estado de las transacciones, balances, facturas, etc. En la aplicación existen distintas secciones donde se diferencian las transacciones que son de tipo POS y las que son de tipo Ecommerce.

Cuando usuarios internos de la compañía necesitan crear usuarios para nuevos clientes pueden hacerlo a través de la aplicación. Existen diferentes tipos de roles que pueden darse cuando se crea un usuario nuevo: soporte, merchant, reseller, usuario interno o desarrollador. Dependiendo del tipo de rol que un usuario tenga va a tener permiso para ver más o menos información y va a tener más o menos acceso a los datos y a trabajar con ellos. Además de crear, un usuario con permisos de administración va a poder modificar y deshabilitar usuarios.

Una peculiaridad de ciertos usuarios es que se le puede asignar un permiso a través del cual va a poder ver partes de la aplicación que aún están en fase de pruebas y por lo tanto pueden ir previsualizando secciones o funcionalidad aún no expuesta.

La infraestructura con la que cuenta la aplicación es simple. Por una parte existen dos servidores en los cuales está instalada la aplicación web, una máquina offline donde se produce la ingestión de transacciones provenientes de los sistemas internos de procesamiento de transacciones y una máquina dedicada a contener el servidor de base de datos. Todas estas máquinas están conectadas entre sí en una red interna, para exponer la aplicación al exterior existe un balanceador de carga expuesto a internet y que balancea la carga a un servidor web u otro.

La aplicación web se compone de dos componentes principales, una Single Page Application implementada en AngularJS y una API REST de la que obtenemos los datos

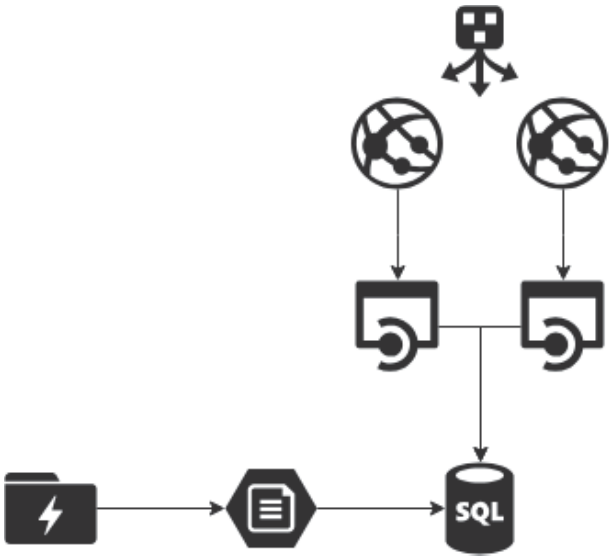


Figura 3.1: Entorno tecnológico actual.

desde la SPA. Estas dos aplicaciones están alojadas en un servidor web IIS en un entorno Windows Server. Con respecto al servidor de base de datos, se trata de un servidor SQL Server 2014 en una máquina Windows Server 2012.

3.1.3. Necesidades de negocio

Los objetivos de negocio se centran en que el sistema sea capaz de ingestar todas las transacciones que se procesen en la plataforma de pagos y poder visualizarlas en la aplicación web además de consultar y filtrar los datos de forma fluida a pesar del volumen de datos con el que se está trabajando.

Cuando el sistema se encuentre totalmente desarrollado el volumen de transacciones que la aplicación va a poder gestionar se va a ver incrementando exponencialmente, especialmente se pretende optimizar la consulta de estas. Las transacciones que van a recibirse deben poder ser tratadas en el menor tiempo posible para que la vista a través de la web sea lo más cercana a tiempo real posible.

No solo se pretende trabajar con los datos de forma fluida sino que además se espera que los datos estén disponibles para su visualización de forma inmediata una vez lleguen al sistema, para ello el proceso de normalizado e inserción de los datos en el sistema de almacenamiento debe ser lo más fluido posible, y por supuesto, la consulta de estos debe ser altamente eficiente.

La aplicación web muestra las transacciones para cada usuario, normalmente con un filtro de fechas. Además, como se puede ver en la figura 3.2, la aplicación dispone de un dashboard donde se puede visualizar información de las transacciones que han sido procesadas y ofrece un análisis su estado en el rango de fechas seleccionado mediante diferentes técnicas de visualización, desde diagramas de puntos para visualizar el volumen

de transacciones procesado a diagramas de barras o indicadores con ratios de aceptación y tipo de tarjeta o método de pago utilizados. Este dashboard lleva a cabo diferentes consultas a la base de datos. Dependiendo del tipo de consulta el tiempo de respuesta puede llegar a ser bastante alto por lo que la experiencia de usuario no es satisfactoria si la página dedica demasiado tiempo a cargar todos los datos. El objetivo es que la aplicación responda de forma fluida y por tanto los tiempos de respuesta deben ser mínimos.

3.1.4. Descripción de los sistemas que desarrollar

Al tratarse de un cambio total en la lógica de ingestión de transacciones tenemos que enfrentarnos al desarrollo completo de toda la parte de backend. Los principales componentes que se deben reemplazar son el sistema de ETL y la forma de almacenar los datos, también se ve afectado el código de la API desde la que consume la aplicación web los datos.

El sistema de ingestión que reemplazará al actual ETL va a consistir en un servicio que procesa los datos que reciba desde la cola de mensajes en streaming, la lógica de tratamiento de los datos se va a ver alterada debido a que el sistema de almacenamiento también va a cambiar por lo que la forma de insertar y tratar los datos cambia de paradigma completamente (índices, consultas, tablas ya no tienen la misma naturaleza y por lo tanto hace que se plantee un cambio organizativo en la forma de almacenar la información)

Además el nuevo sistema de procesamiento en streaming requiere una infraestructura nueva por lo que a nivel de hardware también es necesario la creación de nuevas máquinas. Lo mismo ocurre con la base de datos, al tratarse de un sistema distribuido el número de servidores se va a incrementar para poder hacer consultas más eficientes.

3.2. Requisitos del sistema que desarrollar

3.2.1. Requisitos generales

- El sistema debe ser capaz de tratar y almacenar las transacciones que le lleguen desde el sistema de procesamiento de transacciones de Payvision en tiempo real.

- La consulta de datos desde la web o a través de la API debe de ser lo más fluida posible con tiempos de respuesta que no excedan de más de dos segundos.

- Se requiere que las técnicas de análisis de datos sean eficientes y permitan trabajar con volúmenes masivos de datos.

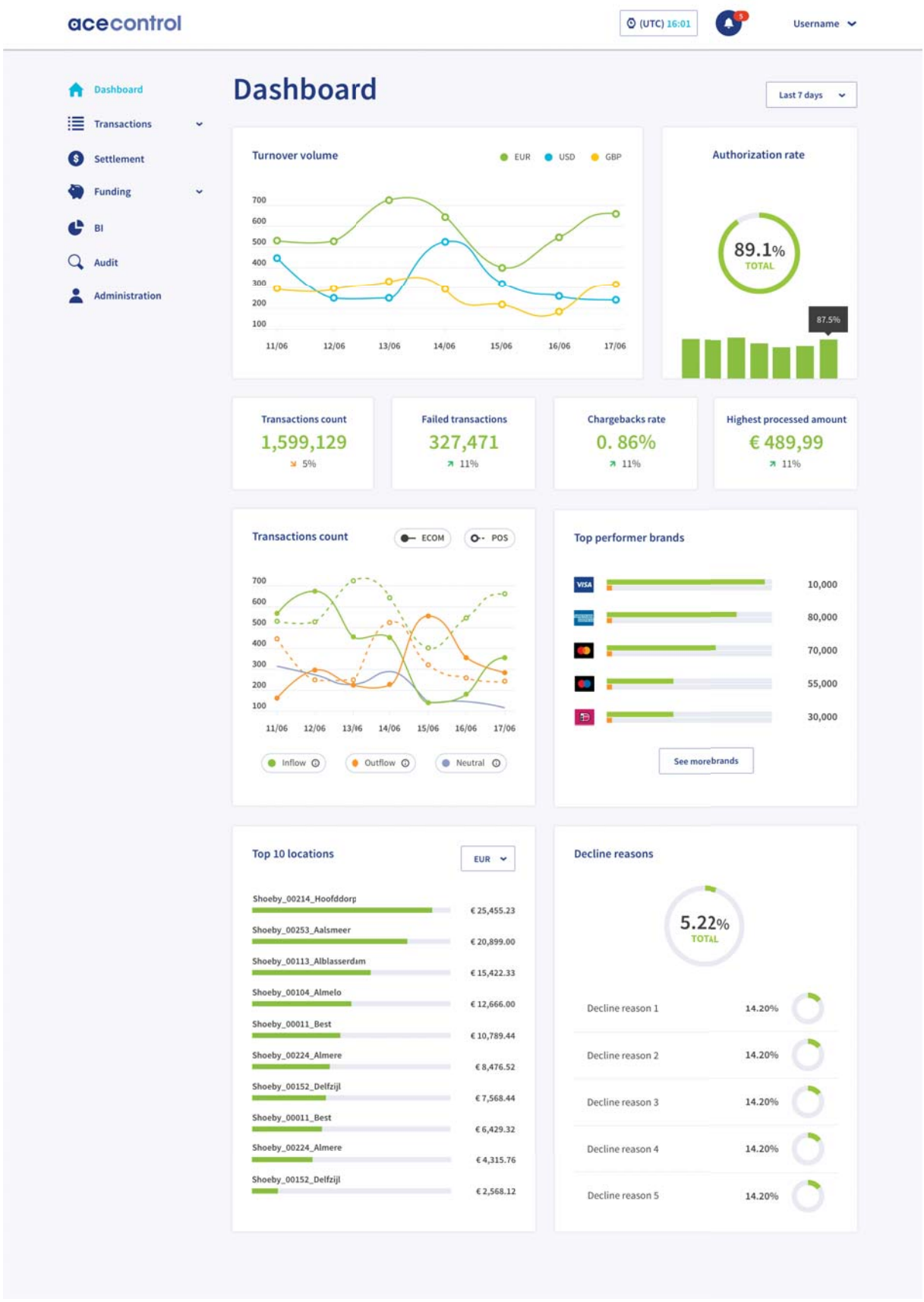


Figura 3.2: AceControl dashboard.

3.2.2. Requisitos funcionales del sistema

- RF-01 - Obtención de transacciones de comercio electrónico: Se deberá poder consultar todas las transacciones de este tipo asociadas a uno o varios merchants.
- RF-02 - Obtención de transacciones de point of sale (POS): Se deberá poder consultar todas las transacciones de este tipo asociadas a uno o varios merchants.
- RF-03 - Obtención de transacciones de pagos alternativos: Se deberá poder consultar todas las transacciones de este tipo asociadas a uno o varios merchants.
- RF-04 - Filtro básico de transacciones por fecha: Para cada tipo de transacción se debe poder aplicar un filtro básico por rango de fechas
- RF-05 - Filtro avanzado de transacciones: Se debe poder aplicar un filtro avanzado con capacidad de filtrar por nombre de la cuenta, localización de la transacción, estado, tipo de transacción, tipo de pago, marca del método de pago, moneda y cantidad monetaria.
- RF-06 - Agregado de transacciones: Para determinados paneles es necesario hacer un agregado del número de transacciones de un determinado tipo para uno o varios merchants.
- RF-07 - Agregado de transacciones por cantidad monetaria: Para determinados paneles es necesario hacer un agregado de la cantidad monetaria para proporcionar información y balance de un determinado conjunto de transacciones.
- RF-08 - Disponibilidad en tiempo real de las transacciones: Según lleguen al origen de datos las transacciones provenientes de Payvision deben ser consumidas por el sistema y deben estar disponibles para su consulta en tiempo real.
- RF-09 - Análisis y resumen sobre transacciones en un rango de fechas: Obtención de datos para alimentar un dashboard que resumirá el número de transacciones, volumen monetario y conteo de transacciones con un determinado estado por intervalos de tiempo durante el último més y el último año.

3.2.3. Requisitos no funcionales del sistema

Requisitos de fiabilidad

- RNF-01 - El sistema deberá estar siempre disponible, sin bajar de un 99 % del tiempo de correcto funcionamiento.

Requisitos de usabilidad

- RNF-02 - El sistema constará de una API REST con endpoints que cumplan adecuadamente este protocolo y como se espera.

- RNF-03 - Las respuestas de error deben ser apropiados con el protocolo.

Requisitos de eficiencia

- RNF-04 - El sistema debe ser capaz de procesar 5 millones de transacciones diarias, un mínimo de 60 por segundo.
- RNF-05 - Cada transacción que sea procesada debe estar disponible para su consulta en menos de 1 minuto.
- RNF-06 - El sistema debe ser capaz de operar adecuadamente con hasta 1000 consultas concurrentes.

Requisitos de seguridad

- RNF-07 - Todas las comunicaciones externas entre servidores de datos, aplicación y cliente del sistema deben estar encriptadas.

4

Evaluación de riesgos

Los riesgos asociados a este proyecto están relacionados con la probabilidad de que una amenaza pueda interferir en el desarrollo de éste y por tanto producir un retraso o incluso el fracaso del mismo.

4.1. Identificación de riesgos

Existe cierta incertidumbre o riesgos que se pueden convertir a lo largo del proyecto en problemas o imprevistos que provoquen que no alcancemos los objetivos planeados. Para evitarlo necesitamos prever y anticipar estas situaciones para evitar que lleguen a convertirse en problemas graves que impidan el desarrollo del proyecto.

A continuación exponemos los riesgos que pueden afectar al proyecto:

Tabla 4.1: Identificación de riesgos

Riesgo	**Tipo de riesgo**	**Descripción**
Estimación errónea	Proyecto y producto	El desarrollo de la funcionalidad y el tamaño de los requisitos se ha subestimado
Volumen de datos erróneo	Proyecto, producto y negocio	El volumen de las transacciones real es mayor al estimado
Volumen de usuarios erróneo	Proyecto, producto y negocio	El volumen de usuarios real es mayor al estimado

Cambios en requerimientos elevado	Proyecto, producto y negocio	El número de cambios en los requerimientos del producto es cambiante durante el proyecto
Software actual no reutilizable	Proyecto y producto	Poca posibilidad de reutilización software existente en cuanto a la lógica de tratamiento de transacciones
Nuevas fuentes de datos	Proyecto y producto	Nuevas fuentes de datos con las que la plataforma se debe integrar
Política de normalización de transacciones cambiante	Proyecto, producto y negocio	Datos relacionados con las transacciones que se deben manejar en el sistema no definidos completamente o cambiantes
Formación escasa en las nuevas tecnologías	Proyecto y producto	Formación escasa en tecnologías de procesamiento en streaming y bases de datos NoSQL
Documentación incompleta de especificaciones de datos	Proyecto y producto	Especificación de datos de las transacciones entrantes incompleta y lógica de normalizado confusa
Componentes software nuevos con los que integrarse	Proyecto y producto	Nuevos componentes software con los que interactuar y no utilizados antes
Velocidad de procesamiento baja	Proyecto y producto	Velocidad de procesamiento baja y rendimiento menor al esperado
Velocidad de consulta baja	Proyecto y producto	Velocidad de consulta de datos baja y rendimiento menor al esperado de la base de datos
Especificación de endpoints incompleta	Proyecto y producto	Listado de funcionalidad en la API incompleto

4.2. Caracterización de las amenazas

Tabla 4.2: Caracterización de las amenazas

Descripción	**Vulnerabilidad**
Fallo eléctrico o fallos en las comunicaciones con otros equipos	Baja
Pérdida de información temporal o indefinida por avería en un dispositivo	Alta

Avería de los dispositivos que impidan su utilización	Media
Fallos de las comunicaciones internas con el servidor web	Media
Caída del servidor por factores ajenos a la empresa	Baja
Desactualización del software que produzca errores	Media
Errores en la administración del servidor	Baja
Borrado accidental de información importante	Media
Pérdida de información por fallo del sistema	Baja
Ficheros corruptos por fallo en el software	Baja
Abandono de la empresa	Media
Baja prolongada o alta frecuencia	Media

4.3. Estimación de riesgos

Para la estimación de riesgos se tiene en cuenta el impacto que cada uno de estos supondrían para el desarrollo de éste, teniendo en cuenta incluso si podrían conllevar la cancelación del mismo. Se puede calcular a partir de la probabilidad que tiene cada una de estas situaciones de suceder y el impacto que este supondría, es decir probabilidad por impacto.

Tabla 4.3: Estimación de riesgos

Riesgo	**Probabilidad**	**Impacto**
Problemas financieros o de cancelación del proyecto	Baja	Muy alto
Funcionalidad y tamaño de los requisitos subestimados	Media	Alto
Cambios en los requerimientos definidos	Media	Medio
Volumen de datos mayor al esperado	Media	Medio
Rendimiento de la plataforma bajo	Media	Alto
Pérdida de personal o falta de conocimientos específicos	Baja	Alto
Problemas en la integración entre componentes	Media	Alto

4.4. Plan de riesgos

El riesgo es el efecto asociado a la ocurrencia de una situación o evento que puede darse durante el desarrollo del proyecto. Ante cada situación que conlleve un riesgo se debe planificar una serie de acciones que mitiguen o solventen estos imprevistos.

Tabla 4.4: Plan de riesgos

Riesgo	**Estrategia**
Problemas financieros o de viabilidad del proyecto	Preparación de un documento breve para la dirección de la compañía con las ventajas que supondría para el negocio y el volumen de datos esperado
Cambios en los requisitos inesperados	Valorar el impacto que supone al proyecto, estudiar su viabilidad y posibilidad de incorporación
Pérdida de personal o problemas tecnológicos	Registrar el estado del proyecto en cada fase y documentar el trabajo completado, si es necesario se dedicará tiempo a formación impactando lo menos posible en el plazo de entrega del producto
Volumen de datos elevado o rendimiento bajo de la plataforma	Al tratarse de tecnologías en clúster se solicitará la incorporación de nuevas máquinas para distribuir la carga

5

Análisis y diseño

Desde la llegada de las transacciones al sistema hasta que el usuario las ve en la interfaz web debe de pasar el mínimo tiempo posible, podemos decir que el principal objetivo del sistema es alcanzar un alto rendimiento a la hora de ingestar y mostrar datos. Para conseguir este objetivo se deberá cambiar la forma en la que las transacciones son gestionadas por el sistema, tanto la forma de ingestión como la forma de almacenar los datos. Al tratarse de una fuente de datos en forma de cola de mensajes se pueden ir consumiendo las transacciones en tiempo real sin esperar a procesos en batch como en la actualidad se está haciendo. Además para optimizar la consulta de datos con un gran volumen de transacciones se va a optar por cambiar la tecnología de almacenamiento de un modelo relacional a una base de datos NoSQL y con especial enfoque en la optimización de la búsqueda y filtrado de datos.

El motor de búsqueda de la base de datos deberá ser capaz de realizar operaciones de análisis de datos y visualización lo que nos ayudará mucho a la hora de mostrar información valiosa para el usuario en la interfaz web. En la figura 5.1 podemos ver un esquema del entorno que vamos a desarrollar, como se puede apreciar la fuente de datos en forma de cola de mensajes va a alimentar al sistema de procesamiento en streaming, el cual limpiará y preparará los datos, que finalmente se irán almacenando en el clúster de Elasticsearch. La aplicación web consumirá los datos del clúster el cual nos proporcionará una velocidad de consulta y búsqueda de transacciones mucho mayor que el servidor de SQL Server actual.

Con el objetivo de justificar el cambio tecnológico referente al sistema de almacenamiento llevaremos a cabo, en secciones posteriores, una comparación en términos de rendimiento de ambos sistemas con diferentes volúmenes de datos, desde un millón a cien millones de transacciones y procederemos a ejecutar diferentes tipos de consultas habituales sobre los datos para medir el tiempo de respuesta del servidor.

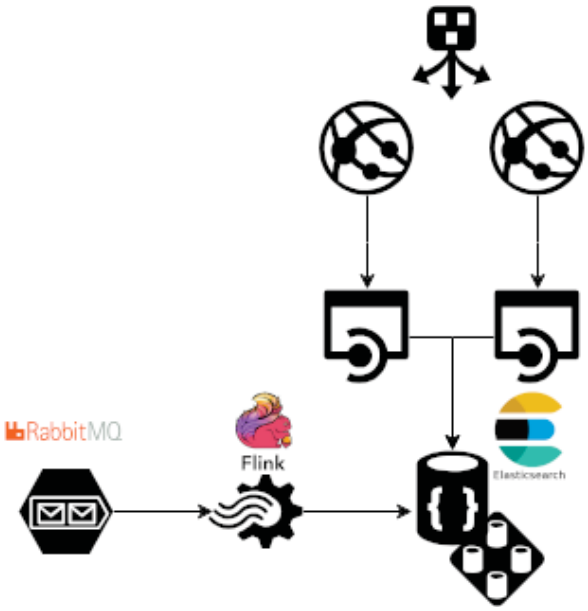


Figura 5.1: Entorno tecnológico propuesto.

5.1. Origen y tipo de fuente de datos

Toda transacción es un evento en el sistema. No son constantes, es decir, tienen carácter aleatorio, su volumen puede variar a lo largo del día y debemos ser capaces de manejarlas según lleguen al sistema. Las transacciones procesadas por la plataforma de pagos de la compañía son comunicadas a la aplicación de reporting para que esta muestre esta información a los clientes de la plataforma. Como la comunicación entre ambos sistemas es asíncrona se decidió utilizar un bróker de mensajería para hacernos llegar todas las transacciones y dejarlas listas para ser consumidas en cualquier momento desde nuestro lado.

La tecnología que vamos a utilizar para el sistema de mensajería es RabbitMQ, un bróker de mensajería que ofrece fiabilidad, enrutamiento flexible, clustering, alta disponibilidad, etc. A pesar de que Apache Kafka este más ligado al modelo de programación en streaming la compañía ha decidido utilizar para este caso RabbitMQ, debido a sus características de fiabilidad de mensajes y aseguramiento de entrega ya que por ejemplo podría reentregar un mensaje si el consumidor fallara. El volumen manejado es perfectamente soportado por un solo nodo de RabbitMQ ya que no se pretende superar los veinte mil mensajes por segundo y por cola. Si fuera necesario añadir más servidores podría escalar sin problemas.

Apache Flink dispone de un conector para RabbitMQ. Se basa en el cliente de Java para RabbitMQ, y hace posible integrar una cola de mensajería de RabbitMQ en el sistema como fuente de datos. Además este conector provee varias formas de consumir mensajes pudiendo configurar el nivel de fiabilidad y garantía de permanencia de los mensajes:

- Exactamente-uno: Para conseguir garantía de exactamente un mensaje se debe

disponer de *checkpointing* habilitado, uso de ids de correlación y configurar la fuente sin ningún nivel de paralelismo.

- Al-menos-uno: Cuando *checkpointing* esta habilitado, pero no se lleva a cabo el uso de ids de correlación o la fuente de datos tiene más de un nivel de paralelismo el sistema solo puede ofrecer garantía del al menos un mensaje.

- Sin garantía: Cuando *checkpointing* no esta habilitado, no se lleva a cabo el uso de ids de correlación y la fuente de datos tiene más de un nivel de paralelismo no se puede ofrecer ningún tipo de garantía sobre los mensajes.

Cabe decir que Apache Flink tiene mejor integración con Apache Kafka ya que las dos tecnologías forman parte del mismo ecosistema de productos de la Apache Software Foundation. Si el sistema tuviera que escalar de forma drástica migrar a Apache Kafka sería la mejor opción. Otro punto a tener en cuenta, pero que queda fuera del alcance de este proyecto, es el caso de manejar los eventos dependiendo de la marca de tiempo asociada. Si esto fuera un requisito del sistema de procesamiento el uso de *checkpoints* y recuperación del histórico para ser reprocesado encajaría mejor con Kafka debido a la persistencia de mensajes en la cola haciendo posible recuperarlos todas las veces que sea necesario, teniendo en cuenta el límite de tiempo que los mensajes deban perdurar antes de ser borrados.

5.2. Modelo de programación y procesamiento distribuido

El modelo de programación que se desea seguir es un flujo de procesamiento en streaming el cual vaya tratando todas las transacciones que lleguen desde la cola de mensajes y las vaya almacenado en el clúster de Elasticsearch. Para conseguir este modelo vamos a utilizar Apache Flink, con el que vamos a obtener un flujo continuo de procesamiento de transacciones. A medida que las transacciones se vayan procesando van a ser almacenadas en el clúster de Elasticsearch con lo que conseguimos que el tiempo desde que llegan al sistema hasta que están disponibles para ser visualizadas sea mínimo.

Gracias al modelo de procesamiento de Apache Flink podemos mantener estado entre transacciones, manejar de orden de llegada o crear de ventanas de ejecución. Esto nos permite manejar las transacciones de forma mucho más cercana a su naturaleza aleatoria ya que al fin y al cabo son eventos que se producen a lo largo del tiempo y que pueden llegar al sistema de forma desordenada. El alcance del proyecto no llega a exprimir todas las posibilidades de Apache Flink a la hora de manejar eventos pero si servir como punto de partida para futuras versiones que implementen nueva funcionalidad como establecer relaciones entre transacciones, reprocesamiento del histórico de eventos y *checkpointing* en caso de fallo.

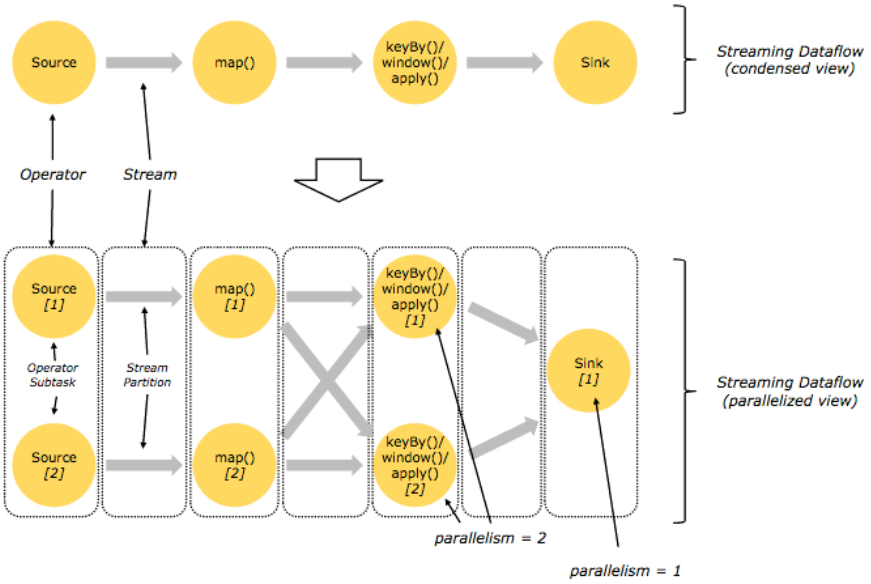


Figura 5.2: Paralelismo de los operadores sobre un flujo de datos.

5.2.1. Programa y flujo de datos

Los programas en Flink son inherentemente paralelos y distribuidos. Durante su ejecución, un stream tiene uno o más particiones, y cada operador tiene una o más subtareas. Las subtareas son independientes unas de las otras, y se ejecutan en diferentes hilos e incluso en diferentes máquinas o contenedores. El número de subtareas indica el paralelismo de un operador en particular. El paralelismo de un stream es siempre el de su operador productor. Como podemos ver en la figura 5.2 diferentes operadores de un mismo programa pueden tener diferentes niveles de paralelismo.

Cada stream puede transportar datos entre dos operadores siguiendo un patrón uno-a-uno o un patrón de redistribución:

- Uno-a-uno: (por ejemplo, entre el *Source* y el operador *map()* de la figura anterior) preserva el particionado y el orden de los elementos. Esto quiere decir que la subtarea '1' del operador *map()* verá los mismos elementos en el mismo orden como si fueran producidos por la subtarea '1' del operador *Source*.
- Redistribución: (como entre el operador *map()* y el operador *keyBy/window* de la figura) cada subtarea envía los datos a diferentes subtareas, dependiendo de la transformación seleccionada. En este tipo de intercambios el orden entre los elementos solo es preservado dentro de cada par de subtareas emisoras y receptoras.

En la mayoría de las operaciones que vamos a llevar a cabo sobre los datos de transacciones vamos a utiliza el patrón uno-a-uno, utilizando la redistribución en operadores que agrupen transacciones por determinados campos como el número de cuenta del cliente.

Para la ejecución del programa sobre un clúster de Apache Flink con diferentes nodos vamos a crear un clúster en Docker con contenedores que van a asumir diferentes tipos de roles (JobManager y TaskManagers), en la sección A.1 del apéndice podemos ver la definición del clúster y detalles técnicos de la arquitectura de un clúster de Apache Flink.

5.3. OLAP vs. OLTP

Podemos dividir los sistemas de almacenamiento de datos en transaccionales (OLTP) o analíticos (OLAP), son dos de los sistemas más comunes para el manejo de datos.

- OLTP (On-line Transaction Processing): Está involucrado en las operaciones de un sistema. Se caracteriza por un gran número de transacciones on-line de pequeña magnitud (INSERT, UPDATE, DELETE). El objetivo principal de OLTP es ejecutar transacciones de forma eficiente, manteniendo la integridad de los datos en entornos de multi acceso. En una base de datos OLTP los datos, esquema y relaciones están definidos siguiendo el modelo relacional (normalmente 3NF[1]). En este modelo las consultas y modificaciones involucran registros individuales como por ejemplo actualizar el email de un empleado en la base de datos de una empresa.
- OLAP (On-line Analytical Processing): Es una clase de sistema que provee soluciones a consultas multi-dimensionales. Se caracteriza por un número pequeño de transacciones complejas y su fuente de datos suele ser información proveniente de un sistema OLTP. Las bases de datos OLAP están altamente desnormalizadas, lo que provoca almacenamiento redundante pero que mejora el rendimiento analítico. En estos sistemas el tiempo de respuesta es una medida efectiva. En estas bases de datos suele haber datos históricos almacenados en un esquema multi-dimensional (típicamente esquema en estrella).

La base de datos de transacciones que manejamos en nuestro sistema sigue el modelo OLAP, centrada en el análisis de los datos, agregados, histórico y sobre todo velocidad a la hora de realizar consultas. El esquema está altamente desnormalizado, con toda la información necesaria en la propia transacción por lo que un motor de búsqueda como Elasticsearch es más apropiado al modelo de datos siendo un sistema relacional menos indicado, ya que añade sobrecarga innecesaria.

5.4. Elasticsearch y almacenamiento en clúster

Elasticsearch es una plataforma de búsqueda cercana al tiempo real, esto quiere decir que existe una pequeña latencia (normalmente un segundo) desde que se indexa un documento hasta que está disponible para ser consultado. Esta característica hace que

[1]Tercera forma normal (3NF) es una forma normal usada en la normalización de bases de datos.

sea una plataforma idónea para el propósito del proyecto. Elasticsearch está diseñado para ejecutarse en clúster. Un clúster consiste en un conjunto de nodos (servidores) que almacenan todos los datos y proveen indexación y capacidad de búsqueda a través de todos ellos. Un clúster se identifica por un nombre único. Cada nodo se asigna a un clúster dependiendo del nombre de clúster que tenga asignado en su configuración, por eso es importante que este nombre sea único.

Nuestro sistema se va a componer de un clúster de tres nodos que actuarán con todos los roles asignables. Esto quiere decir que el propio sistema auto-asignará un nodo maestro y manejará las situaciones de error para reasignar roles en caso de que fuera necesario. Cada nodo que forma parte del clúster almacena datos y participa en las tareas de indexado y búsqueda. Esto hace que sea más eficiente al distribuir la carga y el procesamiento de datos.

Los datos que vamos a almacenar van a ser relativos a las transacciones que sean procesadas por Apache Flink, es decir, todos los datos a almacenar van a tener más o menos la misma estructura. En Elasticsearch la información se almacena en índices (colecciones de documentos con características similares). Un índice se identifica por un nombre y este será utilizado en las operaciones de indexación, búsqueda, actualización y borrado contra documentos que almacene (un documento corresponde con una transacción, es el homólogo a una fila en una tabla de una base de datos relacional). Dentro de un clúster podemos definir todos los índices que sea necesario, en nuestro caso, crearemos un único índice para almacenar todas las transacciones.

Dentro de cada índice podemos definir tipos, los cuales son usados como una categoría o partición lógica de los datos, con lo que podemos almacenar distintos tipos de documentos dentro de un mismo índice. Esta característica es especialmente útil en nuestro caso, ya que todos los datos que almacenamos son transacciones pero cada una tiene sus propias características y campos (e-commerce, POS y bancarias), por lo que este modelo nos proporciona flexibilidad a la hora de almacenar transacciones de todo tipo incluso de nuevos tipos que puedan aparecer de forma transparente. Obsérvese que en un modelo relacionar tendríamos que modelizar cada tipo para crear sus tablas correspondientes, relaciones, etc.

Elasticsearch está diseñado para almacenar volúmenes inmensos de información por lo que un índice puede almacenar conjuntos de datos que superen los límites de almacenamiento de un nodo. Por ello, cada índice se puede subdividir en particiones las cuales sean físicamente almacenadas en nodos distintos. Esta característica es importante por dos razones: permite escalado horizontal y permite distribuir y paralelizar operaciones a través de las particiones (potencialmente distribuidas en múltiples nodos). Como mecanismo de recuperación ante posibles fallos en algún nodo del clúster, Elasticsearch nos permite crear copias de las particiones de nuestro índice, llamadas particiones de réplica. Esta característica nos provee alta disponibilidad en caso de fallo en alguna partición o nodo y nos ayuda a escalar las consultas ya que las búsquedas pueden ser ejecutadas en todas las réplicas en paralelo. Por defecto, Elasticsearch crea cinco particiones y una réplica por cada índice.

Con el objetivo de probar el funcionamiento de un clúster de Elasticsearch y utilizarlo

durante la fase de desarrollo vamos a crear un clúster en Docker con dos nodos. Este desligue requiere cierta configuración extra relativa a la memoria asignada a cada contenedor e inicialización de variables de entorno para formar el clúster y configuración de seguridad. Podemos ver los detalles de la creación del clúster en la sección A.2.2 del apéndice.

5.5. Microsoft Azure como plataforma

El objetivo de este proyecto es construir un sistema de procesamiento y consulta de datos de alto rendimiento, por lo que necesitamos un entorno adecuado. Instalar el sistema en un entorno local u ordenador personal no nos daría un rendimiento ni siquiera cercano al que podemos conseguir en un entorno real o de producción, por eso vamos a utilizar un proveedor de cloud en el que vamos a crear máquinas sin escatimar en potencia para sacar el máximo rendimiento del software que vamos a desplegar.

El sistema de mensajería de RabbitMQ, Apache Flink y Elasticsearch se ejecutarán en servidores Linux. En cuanto al servidor de bases de datos SQL Server va a estar alojado en un Windows Server 2016 Datacenter. Este último lo utilizaremos para evaluar el rendimiento de ambas plataformas y compararlo con el clúster de Elasticsearch.

A pesar de que existen alternativas en cuanto a plataformas cloud, como Amazon Web Services (AWS) o Google Cloud, hemos elegido Microsoft Azure para sustentar nuestro sistema. Todas ellas permiten la creación de máquinas virtuales con diferentes características y redes virtuales donde agrupar conjuntos de máquinas. Además, todas ellas soportan las tecnologías que vamos a utilizar. El motivo de haber elegido Microsoft Azure es principalmente económico debido a que la compañía es partner de Microsoft y disponemos de crédito todos los meses para utilizar la plataforma y todos sus recursos.

6
Desarrollo

El desarrollo de este sistema tiene como objetivo conseguir un flujo de procesamiento en streaming de las transacciones de comercio electrónico que llegan del sistema de pagos de la compañía, el almacenamiento de éstas una vez tratadas y su consulta de forma eficiente. El proyecto se va a centrar en conseguir alto rendimiento a la hora de procesar y consultar los datos, teniendo en cuenta que se deben manejar grandes volúmenes de información.

Para el procesamiento de los datos se ha elegido Apache Flink como sistema de procesamiento en streaming. Su carácter distribuido nos va a ofrecer un rendimiento apropiado y encaja con el flujo entrante de datos que nos llega de forma continua a través de una cola de mensajes.

En cuanto al almacenamiento y consulta de los datos necesitamos un sistema que nos permita manejar y visualizar los datos de forma eficiente, por ello se ha elegido Elasticsearch como motor de búsqueda. Elasticsearch nos va a permitir realizar consultas muy rápidas y flexibilidad a la hora de almacenar datos. Como actualmente se dispone de SQL Server como sistema gestor de bases de datos para el almacenamiento de las transacciones se va a comparar el rendimiento que ambos sistemas ofrecen y su capacidad para responder a consultas de varios tipos con diferentes volúmenes de datos, desde miles a millones de registros.

6.1. Análisis y descripción de los datos

Antes de empezar a diseñar el sistema debemos analizar el tipo de datos con los que vamos a trabajar, para ello vamos a definir y especificar cada uno de los campos de los

que se compone una transacción.

Existen tres tipos de transacciones: e-commerce, point of sale (POS) y transacciones bancarias. Todas ellas comparten determinados campos que son comunes a toda transacción pero cada tipo adicionalmente tiene información específica que no comparte con las demás. A continuación se van a describir con detalle cada uno de los campos que componen cada uno de estos tipos de transacciones.

6.1.1. Definición de los campos

Campos comunes a todas las transacciones

Tabla 6.1: Definición de campos comunes a todas las transacciones

Nombre	**Tipo**	**Descripción**
Id	`int64`	Identificador único
DateTime	`string`	Fecha de la transacción
Type	`string`	Tipo de transacción
Amount	`decimal`	Cantidad de dinero
CurrencyCode	`string`	Código de la moneda
PaymentMethod	`string`	Método de pago
PaymentBrand	`string`	Marca del método de pago
Source	`string`	Origen de la transacción
ResponseCode	`string`	Código de respuesta del procesador
Status	`string`	Estado de la transacción
BankCode	`string`	Código de respuesta del banco
SettlementStatus	`string`	Estado de la consolidación de la transacción
CustomerAccount	`string`	Identificador de la cuenta del merchant
CustomerName	`string`	Nombre del merchant
AccountHolder	`string`	Nombre del titular de la tarjeta o de la cuenta
Region	`int`	Región de comercio (0: Ninguno, 1: Interregional, 2: Internacional)
Description	`string`	Descripción

Campos específicos de transacciones e-commerce

Tabla 6.2: Definición de campos de transacciones e-commerce

Nombre	**Tipo**	**Descripción**
ThreeDSecure	`int`	3D Secure habilitado
TrackingCode	`string`	Código de rastreo definido por el merchant
IP	`string`	Dirección IP
Descriptor	`string`	Descriptor con información adicional
RiskScore	`int`	Nivel de riesgo asociado a la transacción
Expiry	`string`	Fecha de expiración de la tarjeta
Email	`string`	Email del titular de la tarjeta o de la cuenta

Campos específicos de transacciones POS

Tabla 6.3: Definición de campos de transacciones POS

Nombre	**Tipo**	**Descripción**
Terminal	`int`	Identificador del terminal de punto de venta
LocationId	`int`	Identificador de la localización del punto de venta
LocationDescription	`string`	Descriptor de la localización del punto de venta
EFT	`string`	Transferencia electrónica de fondos
AcquirerName	`string`	Nombre del adquiriente que resuelve la transacción

Campos específicos de transacciones bancarias

Tabla 6.4: Definición de campos de transacciones bancarias

Nombre	**Tipo**	**Descripción**
BIN	`int`	Número de Identificación del Banco
BIC	`string`	Código identificador del banco
BankName	`string`	Nombre del banco

6.2. Procesamiento en streaming con Apache Flink

Una vez que tenemos todo preparado para empezar a procesar los datos que nos llegan desde la cola de mensajes de RabbitMQ, podemos empezar a desarrollar un sistema de procesamiento en streaming que va a ir tratando todas las transacciones y aplicando diferentes operaciones sobre los datos. Los bloques básicos de los que se compone un programa en Flink son streams y transformaciones. Conceptualmente un *stream* es un flujo de registros de datos (en nuestro caso transacciones) potencialmente infinito, y una *transformación* es una operación sobre uno o más streams de entrada, y que produce como resultado uno o más streams de salida.

Cada stream comienza con una o más fuentes y termina en uno o más sumideros. En nuestro caso la fuente de datos va a ser la cola de mensajes de RabbitMQ, sobre cada transacción se van a aplicar diferentes transformaciones y finalmente se van a almacenar en Elasticsearch mediante el conector que hace de sumidero.

6.2.1. Conexión con RabbitMQ

Apache Flink ofrece una implementación de un conector para RabbitMQ, este conector provee acceso a streams de datos desde RabbitMQ, para utilizar este conector añadimos la dependencia de Maven a nuestro proyecto. En nuestro caso la versión compatible con la versión 1.4.2 de Flink que estamos ejecutando:

Código 6.1: Dependecia de Maven para el conector de RabbitMQ

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-rabbitmq_2.11</artifactId>
    <version>1.4.2</version>
</dependency>
```

En Apache Flink los conectores no son parte de la distribución original y deben ser añadidos como dependencias adicionales.

Una vez que tenemos disponible el conector debemos configurar la cola de mensajes como fuente de datos, la clase `RMQSource` del conector nos provee la implementación para consumir mensajes desde un cola de RabbitMQ. Esta clase nos provee tres niveles de garantía a la hora de consumir mensajes, para conseguir garantía de *exactamente-un-mensaje* (todos los menajes van a llegar al sistema y no van a producirse duplicados), configuramos el uso de ids de correlación durante la inicialización de la clase `RMQSource` y deshabilitamos el paralelismo.

El siguiente código muestra como conectamos el sistema para leer de la cola de mensajes de transacciones, en este caso desde una cola de mensajes local llamada 'transactions', con uso de ids de correlación y especificando el esquema de deserialización `TransactionSchema` para producir objetos de tipo `Transaction`, los cuales vamos

a manejar en el programa.

Código 6.2: Conector de RabbitMQ

```
env.enableCheckpointing(...);

TypeInformation<Transaction> typeInfo = TypeInformation.of(Transaction.
   ↪ class);

final RMQConnectionConfig connectionConfig = new RMQConnectionConfig.
   ↪ Builder()
        .setHost("127.0.0.1")
        .setPort(5000)
        .setVirtualHost("/")
        .setUserName(...)
        .setPassword(...)
        .build();

final DataStreamSource<Transaction> transactionStream = env
  .addSource(new RMQSource<Transaction>(
      connectionConfig,          // config for the RabbitMQ connection
      "transactions",            // name of the RabbitMQ queue to consume
      true,                      // use correlation ids
      new TransactionSchema()),  // deserialization schema
    typeInfo)
  .setParallelism(1);
```

Además del conector para lectura de una cola de RabbitMQ existe un conector para alimentar una cola de mensajes como sumidero.

6.2.2. Procesamiento de transacciones

Para el procesamiento de las transacciones vamos a trabajar con streams de datos. Sobre estos streams vamos a aplicar ciertas transformaciones que van a tratar las transacciones para limpiar los datos, filtrar transacciones mal formadas y enviar notificaciones a los usuarios.

El sistema ETL de Payvision que actualmente procesa las transacciones implementa gran cantidad de lógica de negocio, dependiendo del tipo de fuente de datos, del sistema de seguridad y del tipo de transacciones que lleguen. El objetivo de este proyecto no es replicar toda esa lógica, ya que se saldría fuera del alcance del proyecto en tiempo y presupuesto. Simplemente se trata de mostrar cómo tratar con los datos en un sistema de streaming y que ventajas y posibilidades ofrece, replicando la funcionalidad básica y mostrando su implementación en Apache Flink.

Una de las transformaciones más habituales en este proceso es la de aplicar una función que realice cierta lógica sobre los campos de una transacción. En Flink la transformación `Map` coge un elemento y produce otro de salida. En la sección de código 6.3 vemos cómo transformamos el código de la moneda en la que se ha realizado la transacción y lo

reemplazamos por su nombre completo. Automáticamente Flink va a aplicar esta función a cada una de las transacciones del stream.

Código 6.3: Transformación Map

```
transactionStream.map(new MapFunction<Transaction, Transaction>() {
  @Override
  public Transaction map(Transaction transaction) {

        switch (transaction.getCurrencyCode()) {
           case "EUR": transaction.setCurrencyCode("Euro");
               break;
           case "USD": transaction.setCurrencyCode("United␣States␣Dollar");
               break;
           case "CAD": transaction.setCurrencyCode("Canadian␣Dollar");
               break;
           case "GBP": transaction.setCurrencyCode("Pound␣sterling");
               break;
           case "INR": transaction.setCurrencyCode("Indian␣Rupee");
               break;
           case "JPY": transaction.setCurrencyCode("Japanese␣Yen");
               break;
           default: transaction.setCurrencyCode("Unknown");
               break;
        }

    return transaction;
  }
});
```

Además de aplicar funciones de transformación sobre los datos, debemos filtrar las transacciones que no sean válidas o estén mal formadas. Una de las validaciones que se deben llevar a cabo es la comprobación de que el identificador de la cuenta del cliente es correcta y existe en nuestro sistema. Para ello, vamos a aplicar la transformación `Filter`, la cual evalúa una función booleana para cada uno de los elementos y retiene solo los elementos en los cual la función haya devuelto un resultado positivo.

En la sección de código 6.4 vemos cómo las transacciones cuya cuenta de cliente no sea válida son automáticamente descartadas del flujo.

Código 6.4: Transformación Filter

```
transactionStream.filter(new FilterFunction<Transaction>() {
  @Override
  public boolean filter(Transaction transaction) {
    int customerAccount = transaction.getCustomerAccount();
    boolean isCustomerAccountValid = accountManager.check(customerAccount);

    return isCustomerAccountValid;
  }
});
```

El manejo de datos en tiempo real y el uso de una herramienta de procesamiento en streaming como Apache Flink nos ofrece muchas más posibilidades que un procesamiento en batch tradicional. Además de las transformaciones habituales sobre los datos como Map o Filter que hemos visto en código anterior es posible implementar funcionalidad mucho más compleja de forma sencilla y que no sería posible en un proceso por lotes.

Para ofrecer al usuario más información sobre las transacciones que se están llevando a cabo en su negocio vamos a informarle en tiempo real de comportamientos sospechosos o no habituales que se estén produciendo. En concreto, queremos enviarle una notificación cuando se produzcan veinte o más intentos de pago con tarjeta de crédito y que estén siendo declinados durante los pasados diez minutos.

Código 6.5: Manejo de ventanas de tiempo

```
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

transactionStream
    .assignTimestampsAndWatermarks(new AscendingTimestampExtractor<
       ↪ Transaction>() {
        @Override
        public long extractAscendingTimestamp(Transaction transaction) {
            return transaction.getDateTime();
        }
    })
    .filter(new FilterFunction<Transaction>() {
        @Override
        public boolean filter(Transaction transaction) {
            boolean isDeclined =
                transaction.getPaymentMethod().equals("Creditcard") &&
                transaction.getStatus().equals("Declined");
            return isDeclined;
        }
    })
    .keyBy("CustomerAccount")
    .window(SlidingProcessingTimeWindows.of(
        Time.minutes(10),
        Time.minutes(1)
    ))
    .apply(new WindowFunction<Transaction,Transaction,Tuple,TimeWindow>() {
        public void apply(Tuple tuple,
                          TimeWindow window,
                          Iterable<Transaction> values,
                          Collector<Transaction> out) {

            int valuesSize = ((Collection<Transaction>)values).size();

            if (valuesSize >= 20) {
                notificationSystem.sendRiskAlert(valuesSize);
            }
        }
    });
```

En la sección de código 6.5 vemos cómo conseguir este comportamiento en apenas

treinta líneas de código. Para conseguir un tratamiento de los datos en tiempo real y permitir retrasos en la llegada de las transacciones al sistema vamos a trabajar con el tiempo asignado al evento. Para ello, especificamos a Flink que el proceso va a trabajar con el tiempo del evento y extraemos este de cada uno de los elementos. El filtro inicial selecciona aquellas transacciones que sean del tipo *Creditcard* y que hayan sido declinadas, posteriormente vamos a agrupar las transacciones por cuenta de cliente, una vez que tenemos las transacciones agrupadas por usuario vamos a definir una ventana deslizante de diez minutos de longitud y que avance cada minuto. Con esto encapsulamos las transacciones ocurridas durante los diez minutos anteriores. En el caso de que superen o igualen las veinte transacciones declinadas se envía una alerta al usuario.

Este flujo de procesamiento se produce paralelamente al proceso principal no afectando ni añadiendo retrasos de tiempo a la hora de ingestar datos. El tratamiento de los datos en streaming resulta natural y fácil de entender e implementar gracias al modelo de programación de Apache Flink, con lo que las posibilidades de actuación sobre las transacciones son mucho mayores. Esto abre la puerta al análisis de los datos y obtención de estadísticas en tiempo real, lo cual nos puede dar mucha información que antes ignorábamos.

En el desarrollo de este programa tan solo se han implementado un procesamiento básico de los datos. La implementación de toda la lógica de negocio que hay detrás del sistema real puede ser migrado a esta plataforma una vez estudiada su viabilidad y descubiertas todas las posibilidades que ofrece.

6.2.3. Almacenamiento en Elasticsearch

Después de haber tratado los datos que nos han llegado desde la cola de mensajes debemos proceder a almacenarlos en Elasticsearch. Apache Flink dispone de un conector para configurar un clúster de Elasticsearch como sumidero o *Sink*, para incluirlo en el proyecto debemos añadir la dependencia de Maven asociada a la versión de Flink y de Elasticsearch que estamos corriendo.

Código 6.6: Dependecia de Maven para el conector de Elasticsearch

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-elasticsearch5_2.11</artifactId>
    <version>1.4.2</version>
</dependency>
```

El conector de Elasticsearch utiliza un cliente `TransportClient` para comunicarse con el clúster de Elasticsearch. Para empezar a cargar datos en el clúster debemos crear la conexión y editar los parámetros de configuración, en especial el nombre del clúster al que nos vamos a conectar. Para almacenar la configuración se debe crear un diccionario donde cada elemento clave-valor es un parámetro de configuración.

En la sección de código 6.7 vemos cómo se ha implementado la conexión con el

clúster de Elasticsearch. En la configuración de la conexión se ha especificado el nombre del clúster y el volumen de transacciones que se van a insertar en cada petición. A la hora de hacer peticiones contra el servidor, es posible configurar el número de elementos que van a formar parte de la operación editando la variable `ElasticsearchSink.CONFIG_KEY_BULK_FLUSH_MAX_ACTIONS`. Esta variable indica el número de elementos que van a almacenar en el buffer hasta que se haga la petición, en nuestro caso, de inserción. Obsérvese que las operaciones en lote nos va a ofrecer un rendimiento mucho mayor que hacerlas una a una. Además, téngase en cuenta que el protocolo de comunicación es HTTP por lo que el proceso de abrir y cerrar una conexión TCP puede ralentizar el sistema si se maneja una cantidad de datos significativo. En nuestro caso, realizaremos inserciones en lote por cada diez mil elementos. Este valor debe ser configurado dependiendo del volumen de datos por segundo que se maneje, incrementándose o disminuyéndose para obtener un rendimiento adecuado.

Además de la inserción, la función `ElasticsearchSinkFunction` puede ser usada para realizar múltiples tipos de peticiones como borrado (`DeleteRequest`) o actualización (`UpdateRequest`).

Código 6.7: Conector de Elasticsearch

```
Map<String, String> config = new HashMap<>();
config.put("cluster.name", "elasticsearch-cluster");
config.put(ElasticsearchSink.CONFIG_KEY_BULK_FLUSH_MAX_ACTIONS, "10000");

List<InetSocketAddress> transportAddresses = new ArrayList<>();
transportAddresses.add(new InetSocketAddress(InetAddress.getByName("
   ↪ 127.0.0.1"), 9300));

transactionStream.addSink(new ElasticsearchSink<>(config,
   ↪ transportAddresses, (ElasticsearchSinkFunction<Transaction>) (
   ↪ transaction, ctx, indexer) -> {

        Map<String, String> json = new HashMap<>();

        json.put("dateTime", transaction.getDateTime());
        json.put("type", transaction.getType());
        json.put("amount", transaction.getAmount().toString());
        ...

        indexRequest = Requests.indexRequest()
                .index("transactions")
                .type("ecommerce")
                .source(json);

        indexer.add(indexRequest);
}));
```

Mediante el uso de *checkpointing* en Flink es posible conseguir tolerancia a fallos utilizando este conector, garantiza entrega *at-least-once* de cada acción contra el clúster de Elasticsearch. Internamente se asegura de que cada petición, antes de que el punto de guardado se lance, ha sido satisfactoriamente recibida por el servidor de Elasticsearch

antes de que más registros sean enviados al sumidero.

Una de las limitaciones observadas en este conector es la imposibilidad de conectar con un clúster de Elasticsearch que requiera autenticación. La única alternativa hasta ahora es la implementación propia de un sumidero que utilice la API de Java y el cliente que ofrece Elasticsearch para la comunicación y almacenamiento de los datos.

6.3. Almacenamiento y motor de búsqueda

Los datos que vamos a almacenar y sobre los que vamos a realizar consultas son los datos de transacciones que han sido procesadas por Apache Flink. El objetivo del sistema de almacenamiento es proveernos alto rendimiento a la hora de hacer búsquedas y realizar consultas sobre los datos, por lo que vamos a comparar dos sistemas con paradigmas diferentes a la hora de modelar los datos y vamos a intentar obtener el máximo rendimiento de cada uno de ellos.

El sistema actual que está implantado en la organización consiste en un servidor SQL Server donde se almacenan todos los datos de las transacciones. Vamos a replicar este sistema en una máquina con unas características específicas y vamos a recrear el modelo de datos para posteriormente evaluar el rendimiento y compararlo con Elasticsearch. El sistema propuesto va a consistir en un clúster de Elasticsearch debido al tipo de datos que manejamos y al objetivo de las consultas que se realizan habitualmente en el sistema de reporting, teniendo como objetivo agilizar la consulta de grandes cantidades de datos.

Durante el desarrollo del proyecto se van a utilizar infraestructuras montadas sobre contenedores Docker. Podemos ver cómo crear un servidor SQL Server en la sección A.3.2 y un clúster de Elasticsearch en la sección A.2.2 del apéndice.

6.3.1. Modelo de datos y definición de tipos

Modelo de datos relacional

El modelo de datos relacional que tradicionalmente se ha utilizado en bases de datos como SQL Server consiste en relacionar las filas de las tablas hijas con las de la tabla padre mediante cuando existe una jerarquía de herencia. En nuestro modelo de datos tenemos una tabla padre GenericTransaction y tres hijas que cuelgan de esta: EcommerceTransaction, POSTransaction y BankTransaction.

Si construyéramos un modelo de relaciones al uso tendríamos un modelo como el que vemos en la figura 6.1. Sin embargo, por motivos de rendimiento, las claves de estas tablas no van a estar relacionadas. Esto es debido al tipo de consultas que se realizan sobre estas tablas. Cuando consultamos las transacciones para determinado usuario el filtro principal y que más coste supone es el de seleccionar las transacciones en un rango de fechas seleccionado. Por este motivo los indices de estas tablas juegan un papel fundamental en

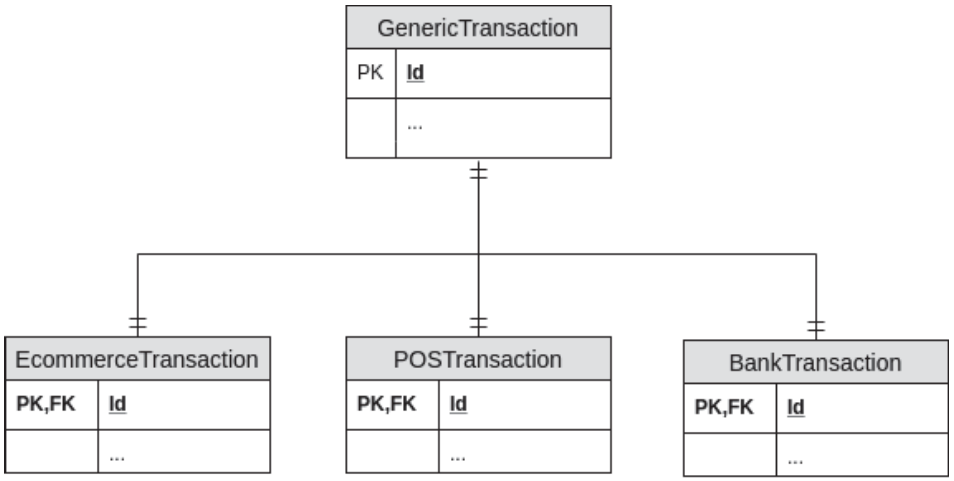


Figura 6.1: Modelo Entidad-Relación tradicional

la organización de los datos de estas tablas.

El índice más importante se encuentra en GenericTransaction y consiste en un índice de tipo clúster (los datos de la tabla se van a almacenar físicamente ordenados en disco) sobre las columnas DateTime y CustomerAccount. Esto nos va a proporcionar un rendimiento mucho mejor para las consultas que involucren estos campos. Además, tenemos índices de tipo no clúster para otros campos como el Id o el Amount involucrados en otras consultas. En las tablas hijas EcommerceTransaction, POSTransaction y BankTransaction el índice clúster esta definido sobre el campo Id para hacer los JOINs más eficientes.

Debido al tipo de consultas que vamos a realizar hemos diseñado un modelo que nos permita mejorar el rendimiento a costa de tener un modelo de tablas y relaciones 'no normalizado' y sin relación directa entre claves. Podemos ver el código SQL para la creación de las tablas y sus índices en la sección B.2.1 del apéndice.

Definición del índice en Elasticsearch

En Elasticsearch los datos se almacenan en índices. Dentro de un mismo índice podemos tener distintos tipos de datos, como una manera de agrupar documentos similares. En nuestro caso, vamos a crear un índice para almacenar las transacciones tanto e-commerce, como POS o bancarias. Gracias a la flexibilidad de estos índices podríamos añadir nuevos tipos de transacciones sin problema al no manejar esquemas de datos fijos.

A pesar de que no es necesario definir la estructura de un documento (debido a que el motor de Lucene infiere los tipos encontrados para indexar la información) es conveniente definir los campos y tipos de datos de los que cada documento se va a componer. Esto hace que el motor de indexación pueda comprender mejor los datos que le van a llegar en cada campo y por lo tanto indexar mejor la información.

La mayoría de los datos que vamos a incluir en cada documento (transacción) son de tipo texto. Elasticsearch traba bien con campos de tipo texto pero debemos diferenciar

entre campos de texto con varias palabras y campos de texto que consistan en tokens o palabras clave. Este tratamiento va a influir en el tipo de búsquedas que vamos a realizar sobre los datos y por tanto en su eficacia y precisión. Para los campos de tipo id o códigos numéricos utilizaremos el tipo `integer`. Uno de los campos más importantes a la hora de definir los tipos es el campo fecha, al tratarse de uno de los campos más utilizados en las búsquedas debe estar bien modelado. Para este tipo de campos Elasticsearch dispone del tipo de datos `date`, el cual interpreta el contenido y formato del valor que reciba en ese campo y lo transforma a una representación de fecha estándar dentro del sistema. Es importante definir bien este tipo de campos porque por defecto el motor de Lucene lo inferiría como texto al no poder asumir que es la representación de un instante de tiempo.

En la sección B.1.1 del apéndice podemos ver la operación de creación del índice de transacciones y sus *mappings* o definición de tipos. Para cada tipo de transacción se debe definir el nombre de los campos de los que se va a componer y su tipo de datos. Cuanto más precisa sea esta definición, mejores resultados obtendremos tanto a la hora de inserción de documentos como a la hora de realizar consultas.

Aunque no este definido en el código creación del índice, por defecto se crean cinco *shards* o particiones primarias con una réplica, lo que significa que nuestro índice tendrá cinco particiones primarias y otras cinco particiones de réplica al tener más de un nodo.

6.3.2. Creación de la infraestructura e inserción de datos

Una vez que tenemos el desarrollo listo vamos a montar un entorno real donde los datos de las transacciones que vayamos procesando van a ser almacenadas. Para ello hemos utilizado la plataforma de cloud Microsoft Azure sobre la que crearemos los servidores de SQL y Elasticsearch. Estos servidores van a consistir en máquinas de alto rendimiento con optimización de memoria, lo que las hace indicadas para este propósito.

Ambos sistemas han sido creados con características que los permitan ofrecer su mejor rendimiento:

- SQL Server: consiste en un servidor Windows Server 2016 Datacenter con SQL Server 2016 instalado en su versión Enterprise, además de SQL Server Management Studio 17.7 desde el que analizaremos las consultas y planes de ejecución. Podemos ver los detalles relativos a la configuración en la sección A.3.1 del apéndice.

- Elasticsearch: se compone de un clúster de tres máquinas con Ubuntu Server 16.04 y Elasticsearch 5.5.3, además del paquete de monitorización de X-Pack. En el nodo principal está instalado Kibana [18] desde el que ejecutaremos las pruebas y monitorizaremos el clúster. Podemos ver los detalles relativos a la configuración en la sección A.2.1 del apéndice.

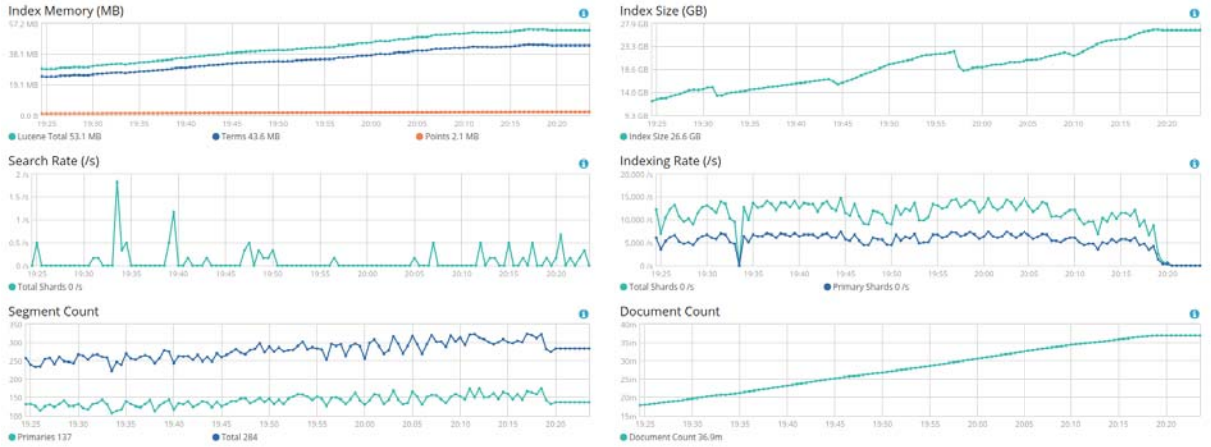


Figura 6.2: Monitorización de índices con Kibana

Rendimiento y ratio de indexación de documentos

La inserción de transacciones en SQL Server se ve sujeta al modelo del índice de tipo clúster que exista en cada tabla, especialmente en la de GenericTransaction. Un índice clúster almacena físicamente las filas de la tabla en orden según las columnas por las que se ha definido. Esto hace que el rendimiento a la hora de realizar consultas se vea incrementado notablemente cuando se filtra por los correspondientes campos. No sólo se indexa por el índice clúster sino que los demás índices que existan también afectan al rendimiento a la hora de insertar datos en cada tabla. Esto hace que en nuestro entorno de pruebas no superemos las 1500 inserciones por segundo, lo cual es un volumen muy bajo.

El objetivo del proyecto es conseguir tiempo real en la visualización de los datos lo que supone que necesitemos alto rendimiento también a la hora de insertar datos en el sistema sin importar que el volumen sea alto. Elasticsearch puede conseguir un ratio de indexación bastante alto, aun más si disponemos de un clúster. Para evaluar el rendimiento de Elasticsearch a la hora de indexar documentos vamos a monitorizar su estado mientras llevamos a cabo inserciones masivas de datos desde Apache Flink. Gracias al paquete de motorización incluido en X-Pack podemos monitorizar todos los nodos y visualizar en tiempo real el sistema.

Podemos observar a través del panel de monitorización de Kibana (figura 6.2) que estamos ingestando alrededor de dos millones de transacciones cada cinco minutos, por lo que el ratio de indexación para los *shard* primarios tanto como para los secundarios supera con creces las necesidades requeridas respecto al volumen de datos que vamos a manejar. El ratio medio de indexación es de 7500 documentos por segundo en los *shard* primarios durante la fase de carga masiva de transacciones.

Durante el proceso de inserción de miles de transacciones por segundo observamos que la memoria de *heap* de la JVM para cada TaskManager pre-asignada por defecto es muy baja (1024 MB), lo que provoca que el nodo falle debido a la saturación de memoria. Para solucionar este problema la incrementamos hasta 5120 MB, con lo que conseguimos

un rendimiento adecuado. Hay que tener en cuenta que si añadimos demasiada memoria el proceso de inicialización del TaskManager se vuelve más lento debido a que la JVM tardará más en inicializar y reservar memoria. Para modificar la memoria de *heap* asignada a los TaskManagers modificamos la propiedad `taskmanager.heap.mb` en el fichero de configuración de Apache Flink (`conf/flink-conf.yaml`).

6.3.3. Búsqueda y consulta de datos

Elasticsearch dispone de una API REST de búsqueda que nos va a permitir ejecutar consultas sobre los datos almacenados. Las consultas se escriben en su propio lenguaje, Query DSL, y se pueden formar utilizando parámetros *query string* o dentro del *body* de la petición HTTP. Las consultas más complejas suelen incluirse en el *body* y tienen estructura JSON.

Una consulta puede ser ejecutada sobre múltiples tipos de un índice, y sobre múltiples índices. Cada campo de un documento es indexado y puede ser consultado. Cuando ejecutamos una petición de búsqueda, esta será difundida a todas las particiones del índice o índices devolviendo un resultado a una velocidad exorbitante. Esto es algo que no podríamos conseguir en una base de datos tradicional.

En el siguiente código podemos ver el código de consulta de todas las transacciones con customerAccount igual a 12586:

Código 6.8: Ejemplo de búsqueda en Elasticsearch

```
curl -XGET "http://elasticsearch:9200/transactions/_search" -d'
{
  "query": {
    "term": {
      "customerAccount": "12586"
    }
  }
}'
```

Si quisiéramos realizar la búsqueda pero solo en las que son de tipo e-commerce, simplemente añadiríamos el tipo en la URL:

Código 6.9: Ejemplo de búsqueda por tipo en Elasticsearch

```
curl -XGET "http://elasticsearch:9200/transactions/ecommerce/_search"
```

Debemos tener en cuenta que todas las consultas en Elasticsearch realizan algún tipo de calculo de relevancia a la hora de seleccionar los resultados. Sin embargo no todas las consultas tienen una fase de análisis. Además de las consultas especializadas como las consultas `bool` o `function_score`, que no operan sobre campos de texto, las consultas textuales se pueden dividir en dos familias:

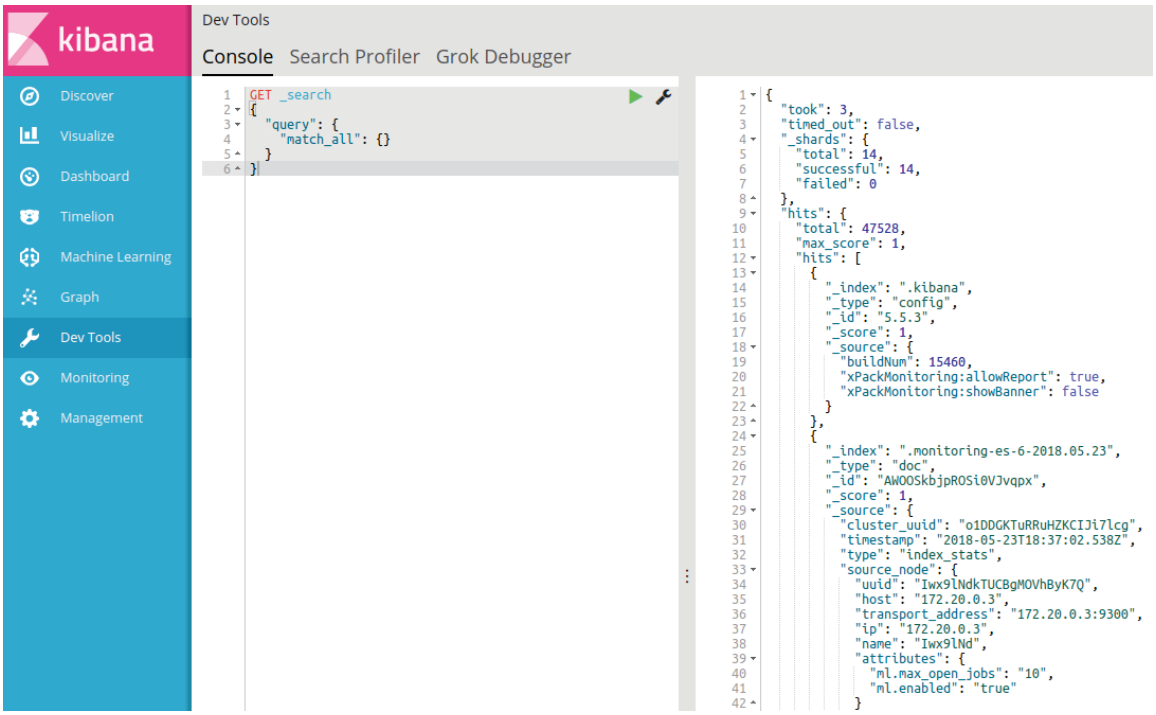


Figura 6.3: Consola de búsquedas en Kibana

- Consultas basadas en términos: Son consultas de bajo nivel que no requieren de una fase de análisis ya que operan sobre un único término. En una consulta de este tipo se busca por un *término exacto* en el índice invertido y se calcula su relevancia para cada documento que contiene ese término.

- Consultas de texto completo: Si se quiere realizar consultas sobre campos de texto completos, primero se pasará el string de búsqueda a través del analizador apropiado y se producirán una lista de términos para ser buscados. Una vez que estos términos han sido obtenidos se ejecutará una consulta para cada uno de ellos combinando los resultados para conseguir una puntuación de relevancia final.

A través de Kibana podemos realizar consultas fácilmente desde las herramientas para desarrolladores. Para acceder a Kibana abrimos un navegador web y accedemos a la dirección donde esté levantado, por defecto en el puerto 5601. En la figura 6.3 podemos ver una captura de consola a través de la cual podemos ejecutar consultas.

7
Resultados

7.1. Análisis de rendimiento de consulta

El rendimiento que nos de la base de datos es muy importante a la hora de conseguir un sistema rápido y eficiente, pero sobre todo fundamental para un sistema en el que los datos deben estar disponibles en el menor tiempo posible para su visualización. Cuando el volumen de datos es suficientemente grande pueden existir problemas de rendimiento a la hora de realizar consultas sobre los datos. Por ello vamos a medir dos sistemas de almacenamiento de datos: en primer lugar, SQL Server, el cual es la herramienta que actualmente esta siendo utilizada como base de datos y en segundo lugar, Elasticsearch, la cual se propone como alternativa de almacenamiento rápida, distribuida y escalable.

Para realizar esta comparación de rendimiento en términos de consulta de datos vamos a ejecutar distintos tipos de consultas habituales en nuestro sistema de reporting y evaluar el tiempo de respuesta. Como el volumen de los datos que almacenamos es importante, y trascendente en el tiempo de respuesta del servidor, se van a ejecutar las consultas sobre distintos volúmenes incrementando el número de registros en cada iteración. Ambos sistemas han sido construidos con las características hardware y software más apropiadas para cada tecnología, por lo que el servidor de SQL Server consistirá de un servidor con Windows Server 2016 Datacenter y Elasticsearch sobre tres servidores Linux (Ubuntu Server 16.04) en clúster. Todas las máquinas son idénticas en términos de hardware, por lo que la velocidad de la memoria o el volumen de entrada/salida en disco son los mismos. En la secciones A.2.1 y A.3.1 del apéndice podemos ver todos los detalles de infraestructura y configuración de ambos sistemas.

Para proveer de datos a ambos sistemas se ha implementado un generador de transacciones simuladas, las cuales se irán insertando en las dos plataformas. Para la

realización de las pruebas se utilizarán volúmenes que irán desde un millón hasta cien millones de registros. En el caso de SQL Server la creación de índices es algo fundamental para que las consultas tengan un rendimiento apropiado. Para ello analizamos las consultas que vamos a ejecutar y creamos los índices adecuados. Por ejemplo, casi todas las consultas tienen un filtro por fecha por lo que creamos un índice de tipo clúster sobre el campo fecha, lo que hará que los datos se almacenen físicamente en disco ordenados por fecha y que las consultas sean mucho más eficientes. Esta creación de índices la haremos en el momento de la creación de las tablas para que estén presentes desde el primer momento. En la sección B.2.1 del apéndice podemos ver el código SQL para la creación de las tablas de transacciones y sus índices. Elasticsearch automáticamente indexa todos los campos de cada documento (internamente llevado a cabo por el motor de Lucene) y crea los índices correspondientes dependiendo del tipo de datos que contengan, por lo que debemos definir los tipos de transacciones que vamos a almacenar en el índice para asegurarnos de que interpreta los datos adecuadamente. En la sección B.1.1 del apéndice podemos ver el código Query DSL para la creación del índice de transacciones y sus tipos.

Todas las consultas sobre el servidor de SQL son en caliente, es decir, la primera vez se ejecuta una consulta su tiempo de respuesta es mucho mayor debido a que el motor de SQL Server tiene que generar el plan de ejecución, por lo que solo vamos a tener en cuenta las sucesivas consultas en análisis de rendimiento. Para reflejar esta diferencia, y su implicación en el tiempo de respuesta, las pruebas realizadas sobre un conjunto de cien millones de registros muestran que los tiempos de respuesta de una primera ejecución pueden llegar a ser 20 veces o más el tiempo que tardarían las sucesivas consultas del mismo tipo.

7.1.1. Búsqueda y operaciones de consulta

El rendimiento de las consultas ejecutadas sobre ambos sistemas se ha medido con una sola consulta simultánea, es decir, no existe penalización asociada a bloqueos de tablas o a sobrecarga del servidor por múltiples consultas concurrentes. Tampoco se producirán inserciones o modificaciones de los datos durante la ejecución de estas consultas.

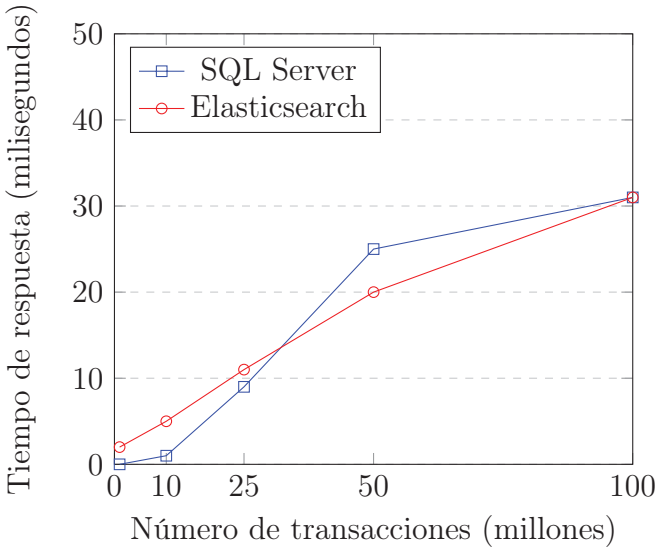
Ambos sistemas cuentan con motorización del tiempo que cada consulta tarda en ser ejecutada por el servidor, incluso podemos ver el plan de ejecución y detalles asociados a la ejecución. El tiempo de respuesta lo mediremos en milisegundos y solo se tendrá en cuenta el tiempo de procesamiento y computación en el servidor, sin tener en cuenta el tiempo de transmisión por la red ni tiempo de procesamiento por parte del cliente.

Consultas simples

Las consultas simples sobre una tabla en SQL Server son bastante eficientes con los índices adecuados. Además, gracias a los índices columnares, los filtros y agregados sobre las columnas indexadas se vuelven mucho más rápidos. Los tiempos medidos en el servidor SQL Server están en entre 1 y 40 milisegundos. El tipo de índice influye de forma dramática en el tiempo de respuesta. Para índices de tipo no clúster y campos secundarios en la

definición de éstos el tiempo de respuesta se ve incrementado notablemente.

En el caso de Elasticsearch, el rendimiento de consultas simples es similar al de SQL Server, por lo que no habría mucha diferencia entre ambos sistemas. En la siguiente gráfica podemos observar como el tiempo de respuesta es similar en este tipo de consultas:



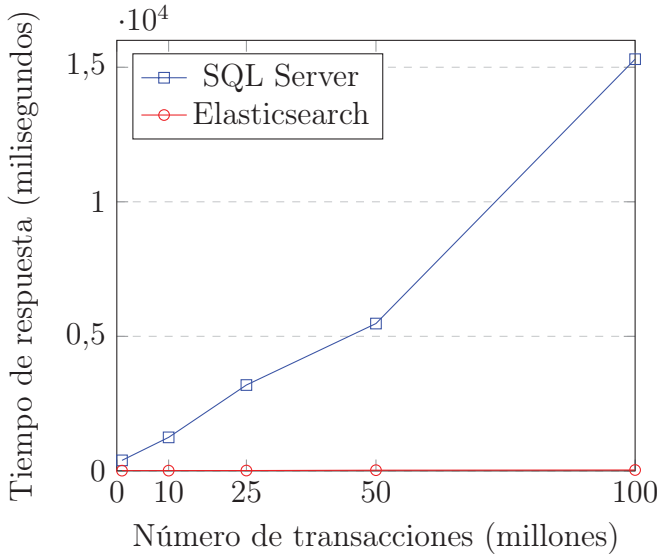
El rendimiento de SQL Server depende en gran medida de los índices creados en cada tabla. Por eso Elasticsearch nos ofrece mucha más flexibilidad, ya que automáticamente indexa todos los campos por lo que tenemos mucha más libertad a la hora de hacer consultas. Si quisiéramos replicar este comportamiento en SQL Server saturaríamos las tablas de índices, lo cual afectaría notablemente y añadiría mucha sobrecarga a cada operación al tratarse de un sistema transaccional.

Los índices columnares han sido introducidos en SQL Server 2016. Mediante éstos podemos generar índices por columnas almacenando la información de cada columna o columnas indexadas en ficheros independientes mucho más pequeños y por tanto más sencillos de recorrer. Estos índices aportan un rendimiento muy superior para determinadas consultas como agregados o consultas analíticas. Este tipo de consultas son altamente eficientes pero están limitadas a un número limitado de columnas.

Consultas condicionales

Las consultas sobre una única tabla son muy eficientes en SQL Server y más si se dispone de los índices adecuados. En modelos relacionales es habitual hacer uniones de tablas a través de las cuales obtener datos almacenados en diferentes tablas del modelo. Sin embargo, Elasticsearch se basa en un modelo desnormalizado donde todos los datos se encuentran agrupados dentro de un mismo índice. Esta característica provoca almacenamiento redundante pero mejora el rendimiento analítico.

Vamos a ejecutar varias consultas que involucren dos tablas en SQL Server para obtener los detalles de determinadas transacciones según una condición de filtrado. En Elasticsearch esta consulta discriminará por el tipo deseado dentro de un mismo índice. Las consultas realizadas tanto para Elasticsearch como para SQL Server las podemos encontrar en las secciones B.1.2 y B.2.2 del apéndice, consultas de la 2 a la 5 en el caso de Elasticsearch y de la 2 a la 6 para SQL Server. Los tiempos medios de respuesta los hemos recogido en la siguiente gráfica:



Como podemos observar las consultas ejecutadas sobre SQL Server a medida que va aumentando el número de registros van siendo cada vez más costosas, llegando a tardar quince segundos con cien millones de registros almacenados. Sin embargo, las búsquedas realizadas sobre Elasticsearch son casi inmediatas independientemente del número de registros, no superando los 40 milisegundos en ningún caso. Esta diferencia tan abismal es debida a varios factores. La unión de dos tablas tiene un coste muy elevado en SQL Server a pesar de que el optimizador de consultas intente acotar los datos y trabajar con conjuntos pequeños de datos intentando filtrar en fases tempranas de la ejecución. En Elasticsearch la situación es radicalmente diferente, ya que todos los datos se encuentran en el mismo índice y tan solo tiene que eliminar los tipos que no satisfagan la condición de la búsqueda con lo que reduce el rango de datos. Además, todas las consultas se ejecutan en el clúster de forma paralela con lo que ganamos en rendimiento horizontal.

Debemos recalcar que estos tiempos de respuesta han sido medidos con el plan de ejecución de SQL Server ya creado, por lo que el tiempo de respuesta es mucho menor. Las pruebas realizadas sobre el servidor sin inicializar llegan a demorarse hasta diez minutos para cien millones de registros.

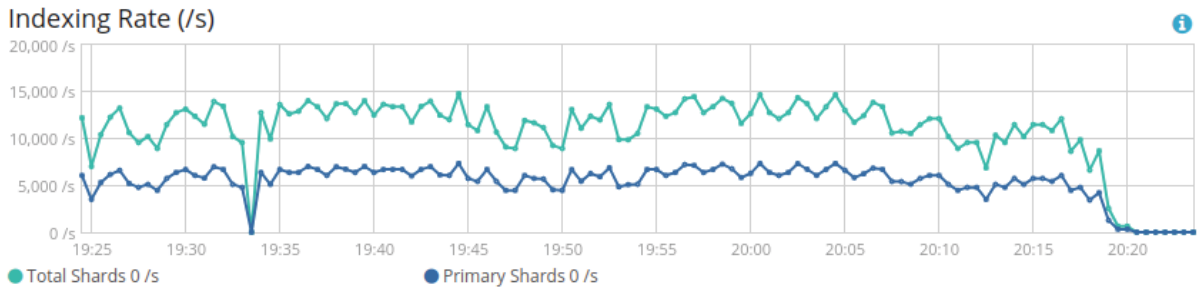


Figura 7.1: Ratio de indexación en el clúster de Elasticsearch.

7.2. Visualización de datos en tiempo real

Como pudimos observar durante la fase de desarrollo del proyecto, el ratio de indexación de documentos en el clúster de Elasticsearch es muy elevado llegando a indexar alrededor de 7000 documentos por segundo en la partición primaria y hasta 15000 en el conjunto de todas las particiones como muestra la gráfica de la figura 7.1 extraída de la herramienta de motorización de Kibana, con lo que conseguimos insertar una media de 2 millones de transacciones cada cinco minutos. Con estos resultados podríamos llegar a insertar 24 millones de transacciones a la hora, lo que satisface con creces el requisito de ingesta de 5 millones de transacciones diarias.

La parte más lenta del proceso es la inserción de datos en el clúster de Elasticsearch. El tiempo de procesamiento de las transacciones con Apache Flink apenas afecta al rendimiento, pudiendo realizarse de forma paralela y distribuyendo la carga en el clúster. La inserción e indexación de transacciones por parte de Elasticsearch es más costosa y no puede igualar a la velocidad con la que Apache Flink le suministra los datos, esto es debido a que el motor de indexación de Lucene tiene que analizar e indexar todos los campos de cada documento. A pesar de esto, como hemos visto en la figura 7.1, Elasticsearch es increíblemente rápido a la hora de insertar nuevos datos y e incluso con cargas altas de datos es altamente eficiente.

El tiempo desde que una transacción se recoge de la cola de mensajes de RabbitMQ hasta que es insertada en Elasticsearch y esta lista para ser consultada es de una fracción de segundo. Aunque no ha sido posible medir este recorrido de forma precisa, la suma de los tiempos que cada sistema dedica a cada transacción es menor a un segundo. Esto hace posible que los datos estén disponibles para su consulta prácticamente en tiempo real, mejorando dramáticamente el rendimiento del requisito no funcional que define un minuto como tiempo máximo de espera.

8

Conclusiones

8.1. Objetivos y resultados

El objetivo de este proyecto era conseguir una plataforma de procesamiento y visualización de transacciones de comercio electrónico en tiempo real, es decir, los datos deben estar disponibles en el mínimo tiempo posible para su consulta, por lo que el principal aspecto sobre el que hemos trabajado ha sido la obtención de un rendimiento apropiado de procesamiento, ingestión y consulta de grandes volúmenes de información. Estos datos son mostrados a los usuarios de la plataforma a través de una aplicación web de reporting, por lo que los datos deben poderse buscar y visualizar de forma eficiente.

Para conseguir este sistema hemos analizado los procesos que se estaban llevando a cabo para ingestar transacciones y mostrarlas a través de la aplicación web y los hemos reemplazado por procesos y tecnologías que encajan mejor con las necesidades y requisitos de rendimiento. En primer lugar, hemos reemplazado el sistema de ETL que estaba ejecutando procesamiento por lotes de ficheros por un sistema de procesamiento en streaming, donde el tratamiento y almacenamiento de las transacciones se lleva a cabo de forma inmediata a medida que estas llegan al sistema. En segundo lugar, hemos reemplazado la tecnología de almacenamiento relacional por un motor de búsqueda mucho más rápido y eficiente para el tipo de datos que estamos almacenado.

Mediante un sistema de procesamiento en streaming podemos tratar las transacciones que nos llegan del sistema de pagos de Payvision de forma inmediata. Esto lo conseguimos recibiendo los datos a través de un sistema de mensajería donde se van dejando las transacciones que han pasado por el procesador de pagos y que vamos leyendo continuamente. Con este sistema no necesitamos esperar a la generación de un fichero que sea tratado diariamente en lote, con lo que podemos ofrecer un tratamiento en tiempo

real de los datos que recibimos a través de la cola de mensajes y obtenemos la capacidad de ingestión de transacciones en nuestro sistema de forma inmediata para su consulta y la posibilidad de ofrecer al usuario nueva información producto de un análisis de las transacciones en tiempo real. Apache Flink ha resultado ser un sistema idóneo para este procesamiento en streaming ofreciéndonos alto rendimiento, escalabilidad, fiabilidad e integración con los otros sistemas que hemos utilizado.

Además del sistema de procesamiento en streaming, hemos conseguido un rendimiento de búsqueda y consulta de datos mucho mayor debido a la utilización de Elasticsearch. Este motor de búsqueda nos permite almacenar la información de forma flexible y desnormalizada, lo cual encaja de manera natural con la forma en la que los datos deben ser almacenados para un sistema OLAP como el nuestro. Los problemas de rendimiento a la hora de realizar consultas sobre los datos han sido solventados para volúmenes grandes de transacciones y durante las pruebas de rendimiento ha demostrado ser capaz de responder en pocos milisegundos a cualquier consulta haciéndolo indicado para alimentar a la interfaz web de forma eficiente.

El resultado de este trabajo es un sistema de procesamiento y reporting de transacciones de alto rendimiento, distribuido y eficiente, permitiendo la visualización de las transacciones procesadas prácticamente en tiempo real por el usuario a través de la aplicación web. El procesamiento de los datos en streaming con Apache Flink junto con el almacenamiento en Elasticsearch nos permiten la ingestión de más de 20 millones de transacciones a la hora y tiempos de respuesta menores a 40 milisegundos para cualquier consulta por lo que el sistema satisface ampliamente los requisitos de rendimiento para el volumen de transacciones esperado, siendo además escalable horizontalmente si fuera necesario.

8.2. Dificultades y problemas encontrados

Los principales problemas que nos hemos encontrado están relacionados con la integración entre las tres tecnologías utilizadas (RabbitMQ, Apache Flink y Elasticsearch) y el paso de datos entre ellos.

El caudal de transacciones que Apache Flink puede leer de RabbitMQ es muy elevado. Si el flujo de transacciones entrantes es mayor que el que el sistema puede procesar se puede producir un aumento progresivo de la memoria de *heap* de cada TaskManager y provocar la caída de este. Si esto sucede el sistema debe estar preparado para la recuperación ante fallos restaurando el estado de procesamiento desde un estado o *checkpoint* anterior. Al tratarse de un sistema de streaming, la depuración del código se hace compleja. Por suerte, Apache Flink ofrece herramientas para depurar localmente el código haciendo este proceso mucho más sencillo.

El impedimento más destacable que nos hemos encontrado ha sido la falta de soporte en cuanto a mecanismos de autenticación por parte del conector de Apache Flink para Elasticsearch. Si disponemos de un clúster de Elasticsearch con el paquete X-Pack de seguridad habilitado y se requiere autenticación para operar sobre el clúster no podremos

hacerlo con la versión actual para las versiones 2.x y 5.x de Elasticsearch. Si queremos conectar Apache Flink con un clúster que requiera autenticación debemos implementar nuestro propio sumidero o *sink* utilizando la API de Java que ofrece Elasticsearch. Esto no es trivial si queremos replicar el comportamiento del conector ofrecido por Apache Flink en términos de tolerancia a fallos y manejo de *checkpoints*.

9

Líneas Futuras

El desarrollo de este proyecto ha abierto la puerta a un nuevo paradigma de procesamiento de datos, este procesamiento en streaming nos ofrece un espectro muy amplio de posibilidades en términos de tratamiento de datos en tiempo real. Apache Flink nos permite trabajar con estado entre eventos, ventanas de tiempo, flujos de trabajo paralelos, etc. todo esto nos permite diseñar soluciones de procesamiento de datos en tiempo real muy potentes pero también añade otros problemas asociados a este modelo de programación. El uso de ventanas de tiempo nos permite trabajar fácilmente con eventos que lleguen en tiempo real al sistema y realizar operaciones o análisis de datos basados en marcas de tiempo, esto añade cierta complejidad si queremos manejar eventos que puedan llegar con retraso al sistema. Apache Flink nos ofrece herramientas para trabajar cómodamente con tiempos asociados a los eventos manteniendo un reloj interno basado en las marcas de tiempo que va analizando, esto hace posible trabajar con eventos que lleguen al sistema con cierto retraso adaptándose al tiempo que estos tengan asociado. Puede que no parezca una característica muy destacable, sin embargo, es el punto de partida para abordar dos problemas que no hemos abordado dentro del proyecto: reprocesamiento de transacciones y manejo de *checkpoints*.

El reprocesamiento de transacciones conlleva cambios que afectan transversalmente a todas las piezas del sistema, por una parte, la cola de mensajes debería mantener los eventos en el tiempo para poder ser reconsumidos, Apache Flink debe ser capaz de tratar los eventos según su marca de tiempo y de mantener puntos de partida de los que arrancar si no se quiere reprocesar todo el histórico de transacciones y por último debe ser capaz de actualizar los registros almacenados en Elasticsearch sobrescribiendo cada transacción evitando duplicados. Un aumento drástico en el volumen de transacciones junto con la necesidad de reprocesamiento podría convertir a Apache Kafka en una alternativa idónea para el sistema de mensajes, soportando cargas de datos masivas, alta disponibilidad y persistencia de eventos.

A pesar de que el rendimiento del sistema es suficiente para el volumen de datos con el que trabajamos y los datos son manejados de forma eficiente por Elasticsearch, sería necesario profundizar en las mejoras de rendimiento y seguridad que se podrían aplicar en el clúster. El uso de índices por merchant agilizaría las consultas para volúmenes masivos de transacciones, además, aumentaría la seguridad y la disponibilidad de los datos de forma inherente. El acceso a los datos se produciría por índice eliminando la posibilidad de acceder a transacciones ajenas y los índices evolucionarían independientemente con sus propias particiones y réplicas, minimizando el daño que provocaría la corrupción de los datos en un índice o nodo.

Por último, la seguridad del clúster de Elasticsearch es importante, debido a las limitaciones que existen en el conector de Apache Flink para conectarse de forma autenticada al clúster debemos deshabilitar el paquete de seguridad de X-Pack exponiendo los datos y detalles del clúster. Esto podemos solventarlo restringiendo el acceso por red o por IP, pero no sería la solución apropiada. Debido a que el conector de Apache Flink tiene limitaciones no solo en cuanto a conexiones autenticadas, sino que no dispone de soporte para versiones superiores a 5.x de Elasticsearch, la implementación de un conector propio utilizando la API de Java que ofrece Elasticsearch sería una alternativa a tener en cuenta en futuras versiones.

Glosario

Card holder Persona asociada a una tarjeta de crédito..

E-commerce El comercio electrónico, también conocido como e-commerce (electronic commerce en inglés) o bien negocios por Internet o negocios online, consiste en la compra y venta de productos o de servicios a través de medios electrónicos, tales como Internet y otras redes informáticas..

ETL Extract, Transform and Load («extraer, transformar y cargar», frecuentemente abreviado ETL) es el proceso que permite a las organizaciones mover datos desde múltiples fuentes, reformatearlos y limpiarlos, y cargarlos en otra base de datos, data mart, o data warehouse para analizar, o en otro sistema operacional para apoyar un proceso de negocio..

Merchant Persona o compañía que vende productos o servicios. En el ámbito del comercio electrónico estas acciones se realizan de forma electrónica a través de internet y los fondos asociados a la transacción van a parar al banco de la merchant desde el banco del card holder..

Pagos alternativos Pagos alternativos a la tarjeta de crédito o débito tradicional, en este tipo de pagos es un intermediario el que gestiona el pago con el comerciante, de esta forma el cliente no se ve obligado a proveer los datos de su tarjeta de crédito. El ejemplo más conocido es PayPal..

POS Point of sale (POS) o point of purchase (POP) es el lugar donde se completa una transacción económica, el merchant calcula el dinero que debe pagar el cliente, prepara una factura y se procede a realizar el pago. A través del punto de venta el cliente realiza el pago al merchant y se produce el pago electrónico..

Reseller Persona o compañía dedicada a ofrecer bienes o servicios a través de otras compañías, esto le permite ofrecer una variedad mayor de servicios. Puede agrupar varias merchants..

Bibliografía

[1] *RabbitMQ*. Pivotal. 2018. URL: https://www.rabbitmq.com/ (visitado 10-06-2018).

[2] *Apache Kafka. A distributed streaming platform*. The Apache Software Foundation. 2018. URL: https://kafka.apache.org/ (visitado 10-06-2018).

[3] Pieter Humphrey. *Understanding When to use RabbitMQ or Apache Kafka*. Pivotal. 2017. URL: https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka (visitado 04-06-2018).

[4] Srinath Perera. *https://www.quora.com/What-is-stream-processing-in-big-data-and-what-does-it-do*. Quora. 2018. URL: https://www.datamation.com/big-data/big-data-analytics.html (visitado 18-04-2018).

[5] Cynthia Harvey. *Big Data Analytics*. Datamation. 2017. URL: https://www.datamation.com/big-data/big-data-analytics.html (visitado 26-04-2018).

[6] *Apache Hadoop*. The Apache Software Foundation. 2018. URL: http://hadoop.apache.org/ (visitado 10-06-2018).

[7] *Apache Storm. A free and open source distributed realtime computation system*. The Apache Software Foundation. 2018. URL: http://storm.apache.org/ (visitado 10-06-2018).

[8] *Apache Samza. A distributed stream processing framework*. The Apache Software Foundation. 2018. URL: http://samza.apache.org/ (visitado 10-06-2018).

[9] *Apache Spark. A unified analytics engine for large-scale data processing*. The Apache Software Foundation. 2018. URL: https://spark.apache.org/ (visitado 10-06-2018).

[10] *Apache Flink. An open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications*. The Apache Software Foundation. 2018. URL: https://flink.apache.org/ (visitado 10-06-2018).

[11] *Introduction to Apache Flink*. The Apache Software Foundation. 2017. URL: https://flink.apache.org/introduction.html (visitado 17-04-2018).

[12] Justin Ellingwood. *Hadoop, Storm, Samza, Spark, and Flink: Big Data Frameworks Compared*. DigitalOcean Inc. 2016. URL: https://www.digitalocean.com/community/tutorials/hadoop-storm-samza-spark-and-flink-big-data-frameworks-compared (visitado 22-04-2018).

[13] Ashish Dalvi. *NoSQL vs. RDBMS 3 differences in power and performance*. CTI. 2015. URL: https://www.cptech.com/blog/nosql-vs-rdbms (visitado 02-05-2018).

[14] *Elasticsearch*. Elastic. 2018. URL: https://www.elastic.co/products/elasticsearch (visitado 10-06-2018).

[15] *Apache Lucene*. The Apache Software Foundation. 2018. URL: https://lucene.apache.org/ (visitado 10-06-2018).

[16] *Solr is the popular, blazing-fast, open source enterprise search platform built on Apache Lucene*. The Apache Software Foundation. 2018. URL: http://lucene.apache.org/solr/ (visitado 10-06-2018).

[17] Alex Brasetvik. *Elasticsearch as a NoSQL Database*. Elastic. 2013. URL: https://www.elastic.co/blog/found-elasticsearch-as-nosql (visitado 14-05-2018).

[18] *Kibana*. Elastic. 2018. URL: https://www.elastic.co/products/kibana (visitado 10-06-2018).

Apéndices

A

Entorno e infraestructura

A.1. Apache Flink

Un clúster típico de Apache Flink está compuesto de un nodo maestro (JobManager) y uno o varios nodos esclavos o trabajadores (TaskManagers). Un usuario puede ejecutar una aplicación de Flink en el clúster subiendo un ejecutable en Java, Scala o Python a la plataforma.

- JobManagers: Gestionan la ejecución distribuida de tareas. Planifica las tareas, gestiona los checkpoints, maneja la recuperación ante fallos, etc. Siempre debe existir al menos un JobManager, en un entorno de alta disponibilidad deben existir múltiples JobManagers, uno de ellos adquiere el rol de líder mientras que los otros permanecen en segundo plano a la espera de ser requeridos.
- TaskManagers: Ejecutan las tareas (más específicamente, las subtareas) de un proceso de ejecución, además de almacenar e intercambiar flujos de datos. Debe existir al menos un TaskManager.

Los puertos por defecto que expone cada servicio son:

- Cliente Web: 8081
- JobManager (RPC): 6123
- TaskManagers (RPC): 6122
- TaskManagers (Datos): 6121

A.1.1. Configuración del entorno en Microsoft Azure

El servidor de Apache Flink se compone de una máquina virtual D8s_v3 con las siguientes características:

- Tipo de computación: Propósito general
- VCPUs: 8
- RAM: 32 GB
- Discos de datos: 16
- Max IOPS: 16000
- SSD: 64 GB
- SO: Ubuntu Server 16.04

En está máquina ejecutaremos un clúster de Apache Flink levantado en Docker con un JobManager y 4 TaskManagers.

A.1.2. Creación de un clúster en Docker

Creación del clúster con Docker Compose

Código A.1: Flink cluster docker-compose.yaml

```
version: "2.1"
services:

  jobmanager:
    image: "flink:1.4.2-scala_2.11-alpine"
    expose:
      - "6123"
    ports:
      - "8081:8081"
    command: jobmanager
    environment:
      - JOB_MANAGER_RPC_ADDRESS=jobmanager

  taskmanager:
    image: "flink:1.4.2-scala_2.11-alpine"
    expose:
      - "6121"
      - "6122"
    depends_on:
      - jobmanager
    command: taskmanager
    links:
      - "jobmanager:jobmanager"
```

```
    environment:
      - JOB_MANAGER_RPC_ADDRESS=jobmanager
```

Ejecutando el comando `docker-compose up` levantamos el clúster. Para escalar el clúster modificando el número de nodos esclavos podemos hacerlo con el comando `docker-compose scale taskmanager=<N>`

Ejecución de una aplicación

Para ejecutar una aplicación de Flink tenemos que subir el ejecutable al clúster e iniciar su ejecución con el comando `flink run`:

Código A.2: Añadir aplicación al clúster

```
JOBMANAGER=$(docker ps --filter name=jobmanager --format={{.ID}})
docker cp <jar> "$JOBMANAGER":/<jar_path>
docker exec -d "$JOBMANAGER" flink run /<jar_path>
```

A.2. Elasticsearch

Para nuestro clúster de Elasticsearch vamos a tener un total de 3 nodos cada uno con varios roles con los que distribuir la carga y la funcionalidad entre todos los nodos de forma más eficiente.

En Elasticsearch cada nodo se puede configurar para uno o varios propósitos, en un cluster de un nodo, este asumirá todos los roles mientras que si disponemos de varios nodos podemos configurar cada uno de estos para llevar a cavo una función específica:

- Nodo maestro: Un nodo con la propiedad `node.master` habilitada lo hace candidato para ser elegido como nodo maestro, el cual será el encargado del control del cluster.
- Nodo de datos: Un nodo con la propiedad `node.data` habilitada lo designa como nodo de datos, el cual se encarga de almacenar los datos y realizar operaciones relacionadas con estos como CRUD, búsquedas y agregados.
- Nodo de ingestión: Un nodo con la propiedad `node.ingest` habilitada lo designa como nodo de ingestión, el cual es encargado de aplicar las operaciones definidas en un flujo de ingestión y que tienen como objetivo transformar, enriquecer, etc. los datos antes de ser indexados por el sistema. Con procesos de ingesta pesados tiene sentido utilizar nodos de ingestión dedicados.
- Nodo de tribu: Un nodo de este tipo de configura mediante la propiedad `tribe.*`, este tipo de nodos se encargan de conectar múltiples clústeres y de realizar operaciones de búsqueda y otras operaciones entre todos los clústeres conectados.

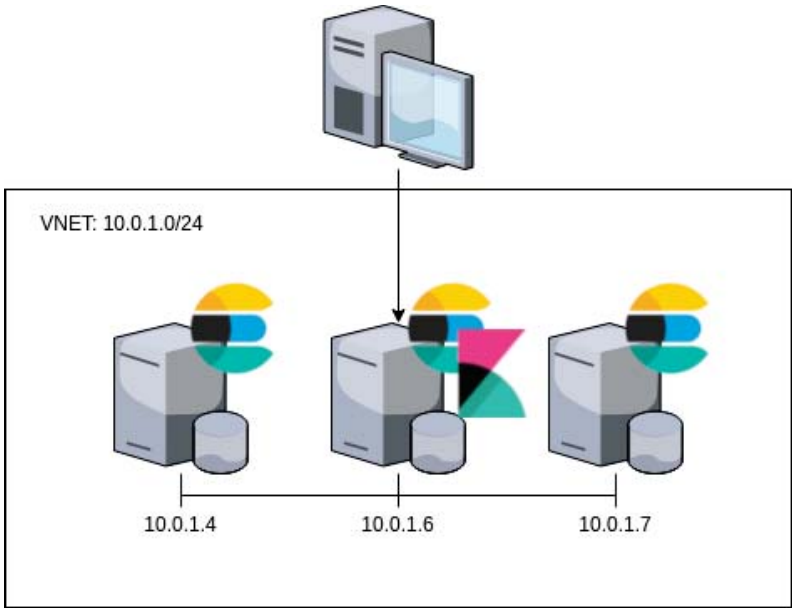


Figura A.1: Clúster de Elasticsearch

Existe un tipo adicional de nodo denominado nodo de coordinación, podemos definir un nodo de este tipo deshabilitando todas las propiedades anteriormente vistas. Este nodo se encarga de recibir las peticiones de búsqueda o indexado por lotes que puedan requerir acciones en varios nodos de datos. Este proceso se lleva a cabo en dos fases, la primera recibe la petición y realiza una interacción con los nodos de datos afectados, en la segunda fase recoge todos los resultados enviados desde los nodos de datos y los une para formar un único resultado.

A.2.1. Configuración del entorno en Microsoft Azure

El clúster de Elasticsearch se compone de tres máquina virtuales E4s_v3 con las siguientes características:

- Tipo de computación: Optimización de memoria
- VCPUs: 4
- RAM: 32 GB
- Discos de datos: 8
- Max IOPS: 8000
- SSD: 128 GB
- SO: Ubuntu Server 16.04

Podemos ver el diagrama de la infraestructura en la figura A.1.

Una vez que tenemos la infraestructura de máquinas disponible, instalamos Elasticsearch y editamos el archivo de configuración `/etc/elasticsearch/elasticsearch.yml` para configurar el clúster de nodos del que se va a componer la instalación. La versión de Elasticsearch con la que vamos a trabajar es la 5.5.3.

Código A.3: Configuración de Elasticsearch

```
cluster.name: elasticsearch-cluster
network.host: 0.0.0.0
discovery.zen.ping.unicast.hosts: ["10.0.1.4", "10.0.1.6", "10.0.1.7"]
xpack.monitoring.enabled: true
xpack.security.enabled: false
```

Además, en el nodo maestro instalamos Kibana para poder interactuar y monitorizar el clúster. Editamos el archivo de configuración `/etc/kibana/kibana.yml` para especificar la URL de Elasticsearch, la configuración de red y los paquetes de X-Pack.

Código A.4: Configuración de Kibana

```
server.port: 5601
server.host: "0.0.0.0"
elasticsearch.url: "http://localhost:9200"
xpack.monitoring.enabled: true
xpack.security.enabled: false
```

Una vez que ya tenemos todas las máquinas del clúster configuradas comprobamos el estado del clúster:

Código A.5: Estado del cluster de Elasticsearch

```
curl -XGET "http://localhost:9200/_cluster/health"

{
  "cluster_name": "elasticsearch-cluster",
  "status": "green",
  "timed_out": false,
  "number_of_nodes": 3,
  "number_of_data_nodes": 3,
  "active_primary_shards": 24,
  "active_shards": 48,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 0,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "number_of_in_flight_fetch": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 100
}
```

Adicionalmente, podemos ver el estado individual de los nodos del clúster:

Código A.6: Estado de los nodos del cluster de Elasticsearch

```
curl -XGET "http://localhost:9200/_cat/nodes?v"

ip       heap.percent ram.percent cpu  ...  node.role master name
10.0.1.6           11          13   2       mdi       *      MEhLM6Q
10.0.1.4           21          14   1       mdi       -      14bAp-U
10.0.1.7           12          14   2       mdi       -      XpfRFS4
```

A.2.2. Creación de un clúster en Docker

Elasticsearch esta también disponible en imágenes de Docker, cada imagen utiliza como base CentOS 7 e incluye X-Pack pre-instalado. Todas las imágenes disponibles se pueden encontrar en `www.docker.elastic.co`

Existen tres configuraciones básicas disponibles. La configuración *básica* incluye Elasticsearch y X-Pack Basic pre-instalado con licencia gratuita. La configuración *platinium* incluye además todas las características avanzadas de X-Pack con una licencia de prueba de treinta días. Por último, la configuración *oss* es la más simple, contiene solo Elasticsearch en su versión open-source.

Para descargarse la imagen de Elasticsearch simplemente ejecutamos el comando `docker pull`:

Código A.7: Imágenes de Docker

```
docker pull docker.elastic.co/elasticsearch/elasticsearch
docker pull docker.elastic.co/elasticsearch/elasticsearch-platinum
docker pull docker.elastic.co/elasticsearch/elasticsearch-oss
```

Creación del clúster con Docker Compose

Para nuestro clúster en Docker crearemos dos nodos que harán las funciones de nodos maestros y de datos, además de las de ingestion y coordinación. Adicionalmente levantaremos una imagen de Kibana para gestionar vía web el clúster y tener acceso a todas las características de X-Pack disponibles. Obsérvese que desactivamos X-Pack Security para todos los nodos del clúster, esto es debido a que no necesitamos mecanismos de autenticación.

Código A.8: Elasticsearch cluster docker-compose.yaml

```
version: '2.2'
services:

  elasticsearch1:
    image: docker.elastic.co/elasticsearch/elasticsearch:5.5.3
```

```
    container_name: elasticsearch1
    environment:
      - xpack.security.enabled=false
      - cluster.name=elasticsearch-cluster
      - bootstrap.memory_lock=true
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
    mem_limit: 4g
    mem_swappiness: 0
    cap_add:
      - IPC_LOCK
    volumes:
      - esdata1:/usr/share/elasticsearch/data
    ports:
      - 9200:9200
      - 9300:9300
    networks:
      esnet:
        aliases:
          - elasticsearch

  elasticsearch2:
    image: docker.elastic.co/elasticsearch/elasticsearch:5.5.3
    environment:
      - xpack.security.enabled=false
      - cluster.name=elasticsearch-cluster
      - bootstrap.memory_lock=true
      - discovery.zen.ping.unicast.hosts=elasticsearch1
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    ulimits:
      memlock:
        soft: -1
        hard: -1
      nofile:
        soft: 65536
        hard: 65536
    mem_limit: 4g
    mem_swappiness: 0
    cap_add:
      - IPC_LOCK
    volumes:
      - esdata2:/usr/share/elasticsearch/data
    networks:
      - esnet

  kibana:
    image: docker.elastic.co/kibana/kibana:5.5.3
    ports:
      - 5601:5601
```

```
    environment:
      - ELASTICSEARCH_URL=http://elasticsearch:9200
    networks:
      - esnet
    depends_on:
      - elasticsearch1

volumes:
  esdata1:
    driver: local
  esdata2:
    driver: local

networks:
  esnet:
    driver: bridge
```

Ejecutando el comando `docker-compose up` levantamos el clúster, además utilizamos volúmenes para los nodos de datos que se crearán también en el caso de que no existan previamente.

Configuración de nodos especializados

Si queremos añadir nodos con roles específicos a nuestro clúster podemos hacerlo en la configuración de entorno de nuestra definición de servicio:

Código A.9: Configuración de nodo de ingestión

```
elasticsearch-ingest-node:
  image: docker.elastic.co/elasticsearch/elasticsearch:5.5.3
  environment:
    - xpack.security.enabled=false
    - cluster.name=elasticsearch-cluster
    - bootstrap.memory_lock=true
    - node.master=false
    - node.data=false
    - node.ingest=true
    - search.remote.connect=false
    - discovery.zen.ping.unicast.hosts=elasticsearch-master-node
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
  ulimits:
    memlock:
      soft: -1
      hard: -1
    nofile:
      soft: 65536
      hard: 65536
  mem_limit: 4g
  mem_swappiness: 0
  cap_add:
    - IPC_LOCK
  networks:
```

```
    - esnet
```

En el caso de que queramos tener un nodo cliente o de coordinación en nuestro clúster debemos abrir los puertos 9200 y 9300 mediante los que interactuaremos a la hora de realizar cualquier petición.

Código A.10: Configuración de nodo de coordinación

```
elasticsearch-coordinating-only-node:
  image: docker.elastic.co/elasticsearch/elasticsearch:5.5.3
  environment:
    - xpack.security.enabled=false
    - cluster.name=elasticsearch-cluster
    - node.master=false
    - node.data=false
    - node.ingest=false
    - search.remote.connect=false
    - discovery.zen.ping.unicast.hosts=elasticsearch-master-node
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
  ports:
    - 9200:9200
    - 9300:9300
  networks:
    - esnet
```

Obsérvese que por defecto, cualquier nodo en el clúster puede actuar como un cliente para conectar con clústers remotos. La propiedad `search.remote.connect` puede ser definida como `false` para prevenir que ciertos nodos se conecten a clústers remotos. Las búsquedas inter-clúster deben ser enviadas a nodos con la propiedad de cliente inter-clúster habilitada.

Configuración avanzada

La configuración para la propiedad `vm.max_map_count` del kernel debe establecerse con al menos un valor de 262144 para su uso en producción.

Código A.11: Configuración del máximo número de áreas de MV para un proceso

```
$ grep vm.max_map_count /etc/sysctl.conf
vm.max_map_count=262144
```

A.3. SQL Server

El servidor de SQL Server va a consistir en un solo nodo, para el entorno real vamos a tener una instalación de SQL Server 2016 con todas sus características disponibles en su

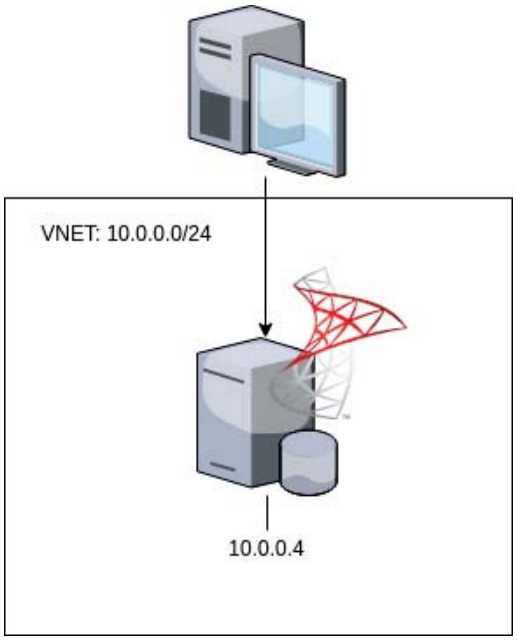


Figura A.2: Servidor de SQL Server

versión trial. Además se ejecutará sobre Windows Server 2016 para obtener un entorno Windows completo.

A.3.1. Configuración del entorno en Microsoft Azure

El servidor de SQL se compone de una máquina virtual E4s_v3 con las siguientes características:

- Tipo de computación: Optimización de memoria
- VCPUs: 4
- RAM: 32 GB
- Discos de datos: 8
- Max IOPS: 8000
- SSD: 128 GB
- SO: Windows Server 2016 Datacenter

Podemos ver el diagrama de la infraestructura en la figura A.2 y la información del sistema en la figura A.3

En la máquina se encuentra instalado SQL Server 2016 SP1 Enterprise Edition.

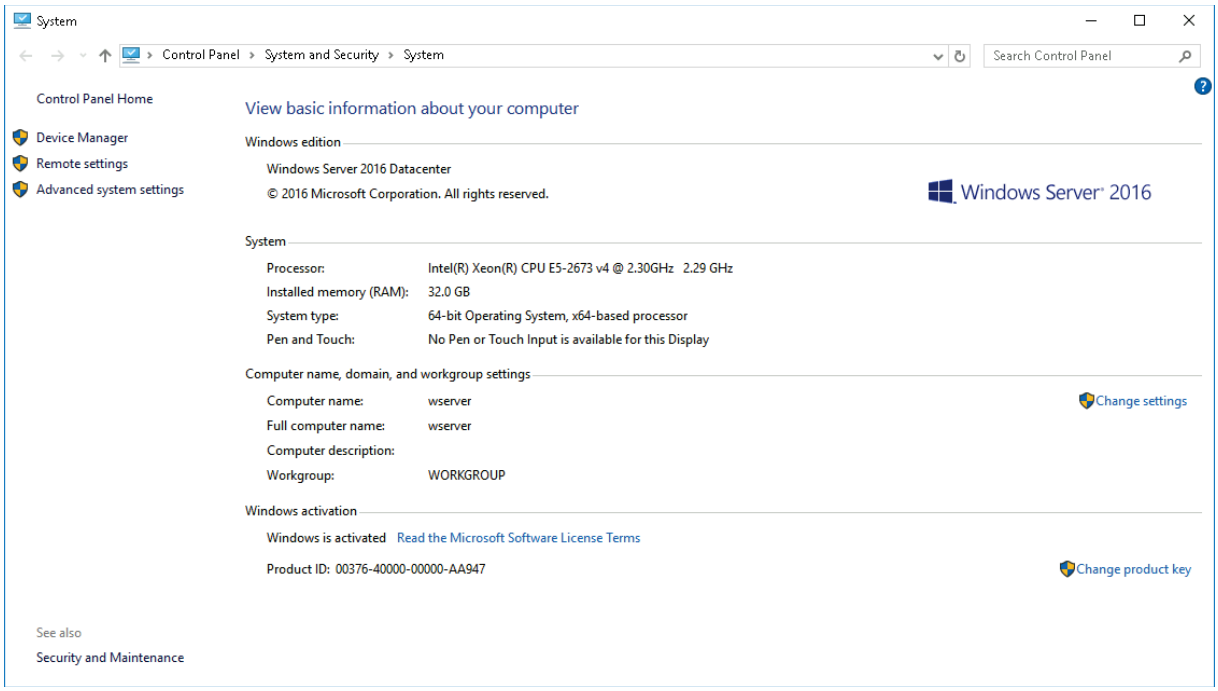


Figura A.3: Windows Server 2016 Datacenter: Información del sistema

A.3.2. Creación de un servidor en Docker

Desde la versión 2017 es posible ejecutar SQL Server en Linux. Microsoft ha hecho muchos esfuerzos para liberar una versión de SQL Server para Linux, lo que ha facilitado la creación de imágenes de Docker. La imagen oficial puede descargarse del hub de Microsoft, esta basada en Ubuntu 16.04 y contiene una instancia de SQL Server ejecutándose sobre él.

Código A.12: Imagen de Docker

```
docker pull microsoft/mssql-server-linux:2017-latest
```

Para iniciar el contenedor ejecutamos el siguiente comando donde mediante variables de entorno aceptamos el acuerdo End-User Licensing Agreement, especificamos una contraseña que cumpla los requisitos de SQL Server y exponemos el puerto de conexión con el servidor.

Código A.13: Ejecución del contenedor

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=yourStrong(!)Password'
  -p 1433:1433
  -d microsoft/mssql-server-linux:2017-latest
```

Existe una configuración adicional mediante la cual podemos especificar la versión de SQL Server que queremos ejecutar: MSSQL_PID equivale al ID de producto o edición de

SQL Server que el contenedor va a arrancar, si no se especifica esta variable de entorno utiliza la versión por defecto Developer Edition. Las opciones disponibles son: Developer, Express, Standard, Enterprise o EnterpriseCore.

Una vez que el contenedor esta arrancado interactuaremos con el mediante la herramienta de línea de comando `sqlcmd` incluida dentro del contenedor.

Código A.14: Conexión con SQL Server

```
docker exec -it <container_id|container_name> /opt/mssql-tools/bin/sqlcmd
  -S localhost -U sa -P <your_password>
```

Desde la línea de comandos de `sqlcmd` podemos ejecutar comandos Transact-SQL, en nuestro caso para crear una nueva base de datos:

Código A.15: Ejecución de comandos

```
1> CREATE DATABASE TestDB
2> GO
```

Si queremos ver la instancia desde una herramienta gráfica podemos instalar SQL Operations Studio desde la que podemos ver e interactuar con nuestra instancia de SQL Server.

B

Código Query DSL y SQL

B.1. Elasticsearch

B.1.1. Creación del índice y mappings

Código B.1: Creación del índice de transacciones y sus tipos de datos

```
curl -XPUT "http://elasticsearch:9200/transactions" -d'
{
  "mappings": {
    "ecommerce": {
      "properties": {
        "dateTime":            { "type": "date" },
        "type":                { "type": "keyword" },
        "amount":              { "type": "double" },
        "currencyCode":        { "type": "keyword" },
        "paymentMethod":       { "type": "keyword" },
        "paymentBrand":        { "type": "keyword" },
        "source":              { "type": "keyword" },
        "responseCode":        { "type": "keyword" },
        "status":              { "type": "keyword" },
        "bankCode":            { "type": "keyword" },
        "settlementStatus":    { "type": "keyword" },
        "customerAccount":     { "type": "keyword" },
        "customerName":        { "type": "keyword" },
        "accountHolder":       { "type": "keyword" },
        "region":              { "type": "integer" },
        "description":         { "type": "text" },
        "threeDSecure":        { "type": "integer" },
        "trackingCode":        { "type": "date"  },
```

```
      "ip":                   { "type": "ip" },
      "descriptor":           { "type": "keyword" },
      "riskScore":            { "type": "integer" },
      "expiry":               { "type": "keyword" },
      "email":                { "type": "keyword" }
    }
  },
  "pos": {
    "properties": {
      "dateTime":             { "type": "date" },
      "type":                 { "type": "keyword" },
      "amount":               { "type": "double" },
      "currencyCode":         { "type": "keyword" },
      "paymentMethod":        { "type": "keyword" },
      "paymentBrand":         { "type": "keyword" },
      "source":               { "type": "keyword" },
      "responseCode":         { "type": "keyword" },
      "status":               { "type": "keyword" },
      "bankCode":             { "type": "keyword" },
      "settlementStatus":     { "type": "keyword" },
      "customerAccount":      { "type": "keyword" },
      "customerName":         { "type": "keyword" },
      "accountHolder":        { "type": "keyword" },
      "region":               { "type": "integer" },
      "description":          { "type": "text" },
      "terminal":             { "type": "integer" },
      "locationId":           { "type": "integer" },
      "locationDescription":  { "type": "text" },
      "eft":                  { "type": "keyword" },
      "acquirerName":         { "type": "keyword" }
    }
  },
  "bank": {
    "properties": {
      "dateTime":             { "type": "date" },
      "type":                 { "type": "keyword" },
      "amount":               { "type": "double" },
      "currencyCode":         { "type": "keyword" },
      "paymentMethod":        { "type": "keyword" },
      "paymentBrand":         { "type": "keyword" },
      "source":               { "type": "keyword" },
      "responseCode":         { "type": "keyword" },
      "status":               { "type": "keyword" },
      "bankCode":             { "type": "keyword" },
      "settlementStatus":     { "type": "keyword" },
      "customerAccount":      { "type": "keyword" },
      "customerName":         { "type": "keyword" },
      "accountHolder":        { "type": "keyword" },
      "region":               { "type": "integer" },
      "description":          { "type": "text" },
      "bin":                  { "type": "integer" },
      "bic":                  { "type": "keyword" },
      "bankName":             { "type": "keyword" }
    }
  }
```

```
  }
}'
```

B.1.2. Búsquedas

Código B.2: Búsqueda 1

```
curl -XGET "http://elasticsearch:9200/transactions/_search?size=1000" -d'
{
  "query": {
    "term": {
      "customerAccount": "12586"
    }
  }
}'
```

Código B.3: Búsqueda 2

```
curl -XGET "http://elasticsearch:9200/transactions/pos/_search" -d'
{
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "dateTime": {
              "gte": "now-1000d/d",
              "lt": "now"
            }
          }
        },
        {
          "term": {
            "customerAccount": "12586"
          }
        },
        {
          "terms": {
            "locationId": [
              "100",
              "107",
              "116"
            ]
          }
        }
      ]
    }
  }
}'
```

Código B.4: Búsqueda 3

```
curl -XGET "http://elasticsearch:9200/transactions/pos/_search" -d'
{
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "dateTime": {
              "gte": "now-1000d/d",
              "lt": "now"
            }
          }
        },
        {
          "term": {
            "customerAccount": "12586"
          }
        },
        {
          "term": {
            "locationDescription": "pizza"
          }
        }
      ]
    }
  }
}'
```

Código B.5: Búsqueda 4

```
curl -XGET "http://elasticsearch:9200/transactions/pos/_search?size=10000"
    ↪ -d'
{
  "aggs": {
    "total_amount": {
      "sum": {
        "field": "amount"
      }
    }
  },
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "dateTime": {
              "gte": "now-1000d/d",
              "lt": "now"
            }
          }
        },
        {
```

```
          "term": {
            "customerAccount": "12586"
          }
        },
        {
          "terms": {
            "locationId": [
              "100",
              "107",
              "116"
            ]
          }
        }
      ]
    }
  }
}'
```

Código B.6: Búsqueda 5

```
curl -XGET "http://elasticsearch:9200/transactions/pos/_search?size=10000"
    ↪ -d'
{
  "sort": {
    "amount": {
      "order": "desc"
    }
  },
  "query": {
    "bool": {
      "must": [
        {
          "range": {
            "dateTime": {
              "gte": "now-1000d/d",
              "lt": "now"
            }
          }
        },
        {
          "term": {
            "customerAccount": "12586"
          }
        },
        {
          "terms": {
            "locationId": [
              "100",
              "107",
              "116"
            ]
          }
        }
      ]
```

```
    }
  }
}'
```

B.2. SQL

B.2.1. Creación de tablas e índices

Código B.7: Tabla GenericTransaction

```
CREATE TABLE [dbo].[GenericTransaction](
  [Id] [bigint] NOT NULL,
  [DateTime] [datetime2](7) NULL,
  [Type] [nvarchar](250) NULL,
  [Amount] [decimal](18, 2) NULL,
  [CurrencyCode] [nvarchar](250) NULL,
  [PaymentMethod] [nvarchar](250) NULL,
  [PaymentBrand] [nvarchar](250) NULL,
  [Source] [nvarchar](250) NULL,
  [ResponseCode] [nvarchar](250) NULL,
  [Status] [nvarchar](250) NULL,
  [BankCode] [nvarchar](250) NULL,
  [SettlementStatus] [nvarchar](250) NULL,
  [CustomerAccount] [nvarchar](250) NULL,
  [CustomerName] [nvarchar](250) NULL,
  [AccountHolder] [nvarchar](250) NULL,
  [Region] [int] NULL,
  [Description] [nvarchar](1024) NULL
) ON [PRIMARY]
GO

CREATE CLUSTERED INDEX [IX_GenericTransaction_DateTime]
  ON [dbo].[GenericTransaction]
  (
    [DateTime] ASC,
    [CustomerAccount] ASC
  )
GO

CREATE NONCLUSTERED COLUMNSTORE INDEX [IX_GenericTransaction_Amount]
  ON [dbo].[GenericTransaction]
  (
    [Amount],
    [Id],
    [CustomerAccount]
  )
GO

CREATE NONCLUSTERED INDEX [IX_GenericTransaction_Amount_Join]
  ON [dbo].[GenericTransaction] ([Id],[DateTime])
```

```
INCLUDE ([Amount])
GO
```

Código B.8: Tabla EcommerceTransaction

```
CREATE TABLE [dbo].[EcommerceTransaction](
  [Id] [bigint] NOT NULL,
  [ThreeDSecure] [int] NULL,
  [TrackingCode] [nvarchar](250) NULL,
  [IP] [nvarchar](250) NULL,
  [Descriptor] [nvarchar](250) NULL,
  [RiskScore] [int] NULL,
  [Expiry] [nvarchar](250) NULL,
  [Email] [nvarchar](250) NULL
) ON [PRIMARY]
GO

CREATE CLUSTERED INDEX [IX_EcommerceTransaction_Id]
  ON [dbo].[EcommerceTransaction] ([Id] ASC)
GO

CREATE NONCLUSTERED INDEX [IX_EcommerceTransaction_TrackingCode]
  ON [dbo].[EcommerceTransaction] ([TrackingCode] ASC);
GO
```

Código B.9: Tabla POSTransaction

```
CREATE TABLE [dbo].[POSTransaction](
  [Id] [bigint] NOT NULL,
  [Terminal] [int] NULL,
  [LocationId] [int] NULL,
  [LocationDescription] [nvarchar](250) NULL,
  [EFT] [nvarchar](250) NULL,
  [AcquirerName] [nvarchar](250) NULL
) ON [PRIMARY]
GO

CREATE CLUSTERED INDEX [IX_POSTransaction_Id]
  ON [dbo].[POSTransaction] ([Id] ASC)
GO

CREATE NONCLUSTERED INDEX [IX_POSTransaction_LocationId]
  ON [dbo].[POSTransaction] ([LocationId] ASC)
GO
```

Código B.10: Tabla BankTransaction

```
CREATE TABLE [dbo].[BankTransaction](
  [Id] [bigint] NOT NULL,
  [BIN] [int] NULL,
  [BIC] [nvarchar](250) NULL,
```

```
  [BankName] [nvarchar](250) NULL
) ON [PRIMARY]
GO

CREATE CLUSTERED INDEX [IX_BankTransactionn_Id]
  ON [dbo].[BankTransaction] ([Id] ASC)
GO
```

B.2.2. Consultas

Código B.11: Consulta 1

```
SELECT TOP (1000) [Amount],
    [Id],
    [CustomerAccount]
FROM GenericTransaction
WHERE Amount > 600
```

Código B.12: Consulta 2

```
SELECT TOP(1000) [Id]
      ,[DateTime]
      ,[Type]
      ,[Amount]
      ,[CurrencyCode]
      ,[PaymentMethod]
      ,[PaymentBrand]
      ,[Source]
      ,[ResponseCode]
      ,[Status]
      ,[BankCode]
      ,[SettlementStatus]
      ,[CustomerName]
      ,[AccountHolder]
      ,[Region]
      ,[Description]
FROM GenericTransaction
WHERE CustomerAccount = 11586
```

Código B.13: Consulta 3

```
SELECT TOP(10000) GT.[Id]
      ,[DateTime]
      ,[Type]
      ,[Amount]
      ,[CurrencyCode]
      ,[PaymentMethod]
      ,[PaymentBrand]
      ,[Source]
```

```
      ,[ResponseCode]
      ,[Status]
      ,[BankCode]
      ,[SettlementStatus]
      ,[CustomerAccount]
      ,[CustomerName]
      ,[AccountHolder]
      ,[Region]
      ,[Description]
      ,[Terminal]
      ,[LocationId]
      ,[LocationDescription]
      ,[EFT]
      ,[AcquirerName]
FROM [dbo].[GenericTransaction] AS GT
INNER JOIN [dbo].[POSTransaction] AS PT
  ON GT.Id = PT.Id
WHERE GT.CustomerAccount = 12586
  AND LocationId IN (100, 107, 116)
  AND [DateTime] BETWEEN DATEADD(DAY,-1000, SYSDATETIME()) AND SYSDATETIME
     ↪ ()
```

Código B.14: Consulta 4

```
SELECT TOP(10000) GT.[Id]
      ,[DateTime]
      ,[Type]
      ,[Amount]
      ,[CurrencyCode]
      ,[PaymentMethod]
      ,[PaymentBrand]
      ,[Source]
      ,[ResponseCode]
      ,[Status]
      ,[BankCode]
      ,[SettlementStatus]
      ,[CustomerAccount]
      ,[CustomerName]
      ,[AccountHolder]
      ,[Region]
      ,[Description]
      ,[Terminal]
      ,[LocationId]
      ,[LocationDescription]
      ,[EFT]
      ,[AcquirerName]
FROM [dbo].[GenericTransaction] AS GT
INNER JOIN [dbo].[POSTransaction] AS PT
  ON GT.Id = PT.Id
WHERE GT.CustomerAccount = 12586
  AND LocationDescription LIKE '%␣Pizza␣%'
  AND [DateTime] BETWEEN DATEADD(DAY,-1000, SYSDATETIME()) AND SYSDATETIME
     ↪ ()
```

Código B.15: Consulta 5

```
SELECT SUM([Amount])
FROM [dbo].[GenericTransaction] AS GT
INNER JOIN [dbo].[POSTransaction] AS PT
  ON GT.Id = PT.Id
WHERE GT.CustomerAccount = 12586
  AND LocationId IN (100, 107, 116)
  AND [DateTime] BETWEEN DATEADD(DAY,-1000, SYSDATETIME()) AND SYSDATETIME
     ↪ ()
```

Código B.16: Consulta 6

```
SELECT TOP(10000) GT.[Id]
      ,[DateTime]
      ,[Type]
      ,[Amount]
      ,[CurrencyCode]
      ,[PaymentMethod]
      ,[PaymentBrand]
      ,[Source]
      ,[ResponseCode]
      ,[Status]
      ,[BankCode]
      ,[SettlementStatus]
      ,[CustomerAccount]
      ,[CustomerName]
      ,[AccountHolder]
      ,[Region]
      ,[Description]
      ,[Terminal]
      ,[LocationId]
      ,[LocationDescription]
      ,[EFT]
      ,[AcquirerName]
FROM [dbo].[GenericTransaction] AS GT
INNER JOIN [dbo].[POSTransaction] AS PT
  ON GT.Id = PT.Id
WHERE GT.CustomerAccount = 12586
  AND LocationId IN (100, 107, 116)
  AND [DateTime] BETWEEN DATEADD(DAY,-1000, SYSDATETIME()) AND SYSDATETIME
     ↪ ()
ORDER BY [Amount]
```