
*Facultad de Ciencias*

# Caracterización de tecnologías de procesamiento de datos en streaming sobre una arquitectura orientada al dato

## (Characterization of stream processing technologies on a data-oriented architecture)

**Trabajo de Fin de Máster**

**para acceder al**

**MÁSTER EN INGENIERÍA INFORMÁTICA**

**Autor: Miguel Algorri Álvarez**

**Director\es: Marta Elena Zorrilla Pantaleón**

# Tabla de contenido

# Agradecimientos

Primero quiero agradecer a toda mi familia y amigos por todo el apoyo que me han dado siempre.

Por otro lado, agradezco a todo el equipo de CIC por facilitarme un caso de uso real y datos para trabajar con él para implementar el *benchmark* utilizado en este trabajo.

También quiero agradecer al grupo ISTR de la Universidad de Cantabria por haber confiado en mis habilidades para trabajar con ellos y hacerme sentir como uno más del grupo.

Y, por último, a todos los profesores que me han dado clase a lo largo del máster ya que gracias a ellos he crecido personal y profesionalmente. Y, en especial, gracias a los profesores Marta Zorrilla y José María Drake por haberme ayudado y guiado a lo largo de mis estudios universitarios.

## Resumen

La arquitectura orientada al dato surge como marco arquitectónico para ingerir, procesar y generar valor de las ingentes cantidades datos de distinta tipología que llegan al sistema a gran velocidad. La reciente definición de este marco y de tecnologías para el desarrollo de aplicaciones (microservicios) sobre él hacen necesario un estudio que caracterice a las mismas para los casos de uso más habituales. Este trabajo explora las diferentes tecnologías existentes y compara las prestaciones y el rendimiento de un subconjunto de ellas mediante el uso de *benchmarks*, utilizando tanto datos simulados como procedentes del mundo real. En particular analiza tres procesadores de *streaming*, Spark, Storm y S-Store; un gestor de colas, Kafka y, cuatro sistemas de almacenamiento: Redis, Cassandra, MemSQL y S-Store. Estas pruebas de rendimiento han sido ejecutadas en tres entornos diferentes: un servidor *on-premise*, un clúster de supercomputación y en la *cloud*. Del análisis de los resultados, se ofrece la caracterización que permitirá establecer criterios de diseño para la implementación de flujos de datos que respondan a los siguientes casos de uso: el procesamiento de lecturas horarias de contadores eléctricos, el análisis de *clicks* en anuncios y el cálculo de una tabla clasificatoria en una votación en tiempo real.



**Palabras clave:** *flujos de datos continuos, procesado en tiempo real, bases de datos en memoria, bus de datos, big data*

## Abstract

The data-oriented architecture was created as an architectural framework to ingest, process and generate value from huge amounts of different types of data that are inserted to the system at high throughputs. The recent appearance of this framework and technologies to develop applications (microservices) based on it, makes a study that characterizes them for the most common use cases necessary. This work explores these technologies and compares the features and performance of a subset by means of the execution of several benchmarks. In particular, three streaming processors are analyzed: Spark, Storm and S-Store; a message broker: Kafka and four storage engines: Redis, Cassandra, MemSQL and S-Store. These have been tested in three different environments: an on-premise server, a supercomputing cluster and in the cloud. The characterization of these tools is offered from the analysis of the results achieved. These will help us to establish different design criteria to implement data pipelines that solve the following use cases: processing hourly readings from electric counters, analyzing clicks on ads and calculating a chart leaderboard based on real time voting.



**Keywords:** *data streaming, real time processing, in-memory databases, data pipelines, big data*

# Introducción y contexto

Debido a la transformación digital que se está produciendo en todos los sectores, la cantidad de datos e información digital está creciendo de una manera desmesurada como ya señalaba el informe realizado por ScienceDaily [1], en 2013 en el que se estimaba que el 90% de los datos que existían en todo el mundo habían sido generados en los últimos dos año. Esto se debe a que prácticamente cualquier dispositivo genera información. Esto engloba desde viviendas y vehículos hasta máquinas industriales pasando por dispositivos más personales como los relojes.

La disponibilidad de esta cantidad ingente de datos abre una ventana para crear más valor y oportunidades de negocio mediante el análisis de los datos. No obstante, también conlleva grandes retos a la hora de llevarlo a cabo, puesto que, a mayor cantidad de datos, mayor cantidad de cómputo es necesaria para obtener resultados en un tiempo aceptable.

Asimismo, es necesario integrar una gran diversidad de fuentes de datos, tanto estructurados como no estructurados, por lo que ha sido necesario la creación de una nueva arquitectura, distribuida y escalable, que satisficiera los requisitos de baja latencia y alto *throughput* que los sistemas relacionales tradicionales no podían asegurar. Dicha arquitectura está basada en cadenas de procesamiento de flujos de datos en memoria que ofrecen información a las aplicaciones sin la necesidad de tener que ser almacenada de forma persistente. Estas arquitecturas se conocen por "*Real-time Data Pipeline*" (*RT-Data Pipeline*) [2].

Como las tecnologías que permiten implementar dicha arquitectura actualmente carecen de madurez y documentación suficiente debido a su reciente aparición y adopción, en este trabajo se realiza un estudio de los parámetros de configuración más relevantes y se ejecutan un conjunto de *benchmarks* sobre diferentes fuentes de datos con el fin de caracterizarlas.

Por otro lado, los casos de uso que originariamente han motivado la generación de la tecnología que da soporte a las RT-Data Pipeline están relacionados con el comercio electrónico, las redes sociales y el análisis de datos en tiempo real en los que lo más relevante es el análisis de grandes volúmenes de datos a una gran velocidad (*Big Data*). Sin embargo, la reciente aparición de la Industria 4.0 y la necesidad de analizar los datos generados por toda la infraestructura industrial ha reafirmado la importancia de estas tecnologías. La Industria 4.0 consiste en una transformación digital de todo el proceso de fabricación, desde el productor al consumidor, mediante el uso de tecnologías inteligentes provenientes de análisis de los datos generados por el entorno, información contextual de los procesos y la aplicación de técnicas de inteligencia artificial [3]. Por lo tanto, sin las herramientas, técnicas y buenas prácticas necesarias para ingerir, transformar, procesar, gestionar y analizar los datos que generan, la Industria 4.0 quedaría coja ante la ausencia de uno de sus principales pilares.

Debido a que el concepto Industria 4.0 es muy reciente y hay poca experiencia sobre el comportamiento de un *RT-Data Pipeline* en este entorno, en este trabajo se trata de contribuir al avance de este campo mediante el desarrollo de un *benchmark* que evalúa su rendimiento en un caso de uso real. Dicho *benchmark,* que pertenece al ámbito sector eléctrico, tiene por objeto procesar los datos procedentes de los contadores eléctricos de los clientes de una compañía eléctrica para su facturación por consumo real. Además,

en el trabajo se utilizan otros dos *benchmarks* que evalúan otros dos casos de uso muy diferentes cuyo origen es sintético en vez de real. Un *benchmark* diseñado por Yahoo denominado *Yahoo Streaming Benchmark* [4], que trata el típico caso de uso de *clicks* en anuncios [4]. Y un *benchmark* diseñado por el equipo de S-Store [5], que trata un caso de uso que calcula la tabla clasificatoria de un programa de televisión de cantantes donde los votos llegan en tiempo real desde el público.

## Objetivo

El objetivo principal del trabajo consiste en la caracterización de aplicaciones *soft real time* implementadas utilizando *data streaming pipelines*. Para ello se realizarán las siguientes tareas:

- Explorar las diferentes tecnologías existentes, en particular, dentro del ecosistema Apache.
- Estudiar su manejo y configuración mediante la ejecución de *benchmarks* existentes y la implementación de pruebas de concepto.
- Diseño y creación de un *benchmark* para un caso de uso real.
- Despliegue del sistema en un clúster con diferentes configuraciones.
- Análisis de los experimentos de los cuales se extraerán criterios de diseño de *pipelines* de acuerdo con sus requisitos funcionales y no funcionales.

## Arquitecturas orientadas al dato*: Data Pipelines*

Las cadenas de flujos de datos o *data pipelines* consisten en la integración de una serie de sistemas encargado de la ejecución de un conjunto de tareas de transformación de datos descrito mediante un grafo dirigido acíclico.

Conceptualmente los *data pipelines* existen desde hace muchos años, ya que consisten en ejecutar unos procesos en forma de flujo con el fin de ingerir, transformar, procesar y, en su caso, persistir datos para que otro sistema pueda utilizarlos. Sin embargo, los sistemas que los implementan han ido evolucionando a lo largo del tiempo.

Tras el origen del término *Data Warehouse* en 1988 [6] surge la necesidad de explotar los datos almacenados. Esto da origen a sistemas de transformación de datos en *batch*, que se ejecutan cada cierto tiempo generando un conjunto de datos preparados para ser consumidos. Sin embargo, estos sistemas estaban diseñados para funcionar sobre mainframes y eran poco escalables. A medida que la tecnología iba evolucionando, la cantidad de datos a almacenar y procesar aumentaba considerablemente y estos sistemas se estaban quedando obsoletos ya que debido a su diseño no podían escalar acorde a las necesidades.

No obstante, en 2004 Google ideó un modelo de programación capaz de paralelizar el procesamiento de los datos mediante cálculos parciales para luego unir los resultados, dicho modelo es el conocido MapReduce [7]. A raíz de dicho modelo surgieron sistemas similares a los mencionados anteriormente, pero con la capacidad de escalar horizontalmente. En este aspecto el ecosistema Hadoop ha dominado el sector gracias a su sistema de ficheros basado en el Google File System, que utiliza el MapReduce para hacer procesamiento masivo de datos en *batch*.

Un año más tarde (2005) introdujeron los requisitos que debería tener un sistema de procesamiento de *streaming* en tiempo real [8]. Este planteamiento, que no constaba más que de términos principalmente teóricos, dio lugar años más tarde a tecnologías capaces de procesar los datos a medida que iban llegando al sistema, produciendo así un resultado en tiempo real.

Este nuevo concepto ha provocado un cambio radical en las arquitecturas de los sistemas de procesamiento de datos puesto que el paradigma de programación cambia por completo al pasar de un procesamiento en *batch* a uno en tiempo real. esto ha dado lugar a que las organizaciones de deban plantear si llevar a cabo la reprogramación de los sistemas existentes en *batch* para adaptarlos a un modelo en tiempo real o la convivencia de los sistemas anteriores en *batch* con los nuevos en tiempo real. En 2011, Nathan Marz introdujo la arquitectura Lambda, la cual permitía la convivencia de los sistemas en *batch* junto con los sistemas de tiempo real [9]. No obstante, en el año 2014, Jay Kreps cuestionaría la arquitectura Lambda, para dar paso a una nueva arquitectura que únicamente utilizara los sistemas de procesamiento en tiempo real. Esta arquitectura nacería bajo el nombre de arquitectura Kappa [10].

Ambas arquitecturas están compuestas por tres bloques claramente diferenciados:

- Un gestor de colas que funciona como un almacenamiento ordenado en forma de flujo de los datos de entrada. Este sistema es el que, por un lado, aporta elasticidad al sistema actuando como buffer de los datos de entrada y, por otro lado, facilita el reprocesamiento de los datos. Cada cola puede o no tener un tiempo de caducidad de los datos.
- Un motor de procesamiento de datos capaz de realizar todas las transformaciones y procesados correspondientes. Este es el núcleo de ambas arquitecturas ya que es el que contiene toda la lógica del sistema. En este caso puede haber dos tipos de motores: el motor de procesamiento en *batch* y el motor de procesamiento en tiempo real.
- Una base de datos que actúe como almacenamiento de los resultados obtenidos por el motor de procesamiento de datos. En este caso hay una gran cantidad de alternativas y su elección es muy dependiente de los casos de uso del sistema.

Cabe destacar que la modularidad de estos sistemas permite utilizar diferentes tecnologías por cada uno de los módulos mencionados. Así que, por ejemplo, se podría utilizar una base de datos para ciertos casos de uso y otra base de datos para otros casos de uso dentro del mismo sistema.

Actualmente ambas arquitecturas son las más recientes en el mundo del procesamiento de datos, y, por lo tanto, se va a profundizar en qué es cada una, cómo se diferencian entre sí y qué ventajas y desventajas tienen.

### Arquitectura Lambda

La arquitectura Lambda fue la primera arquitectura en dar a luz tras la aparición del procesamiento en tiempo real puesto que surgió de la necesidad de hacer convivir los sistemas en *batch* anteriores junto con los nuevos sistemas de procesamiento. Esto daría lugar a dos flujos de datos a diferentes velocidades que conjuntamente darían un resultado.

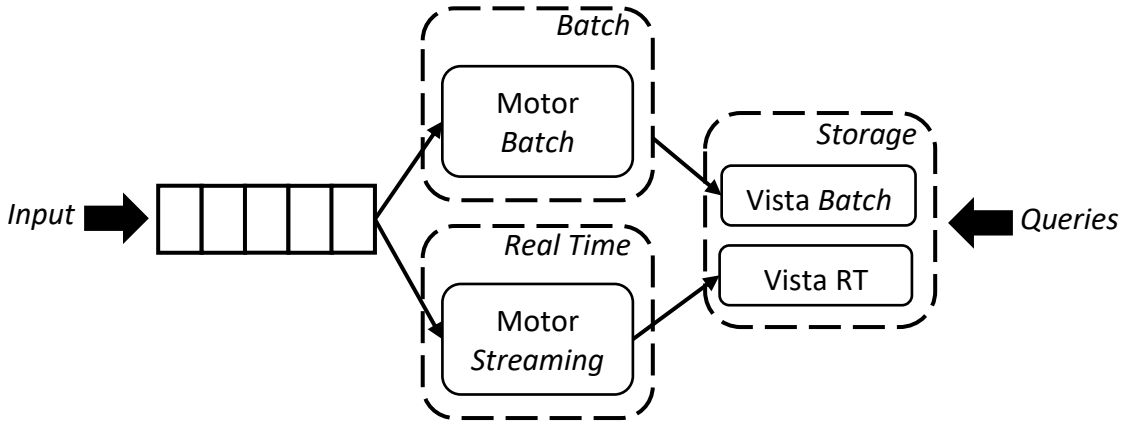


*Ilustración 1. Arquitectura Lambda*

Esta arquitectura presenta una ventaja que es la capacidad de reprocesamiento de los datos. La arquitectura Lambda, gracias a que utiliza un gestor de colas, enfatiza el almacenar indefinidamente los datos de entrada, de manera que en cualquier momento éstos pueden ser reprocesados para obtener un resultado diferente. Esta ventaja permite a las aplicaciones evolucionar a lo largo del tiempo y permite corregir fallos en las mismas pudiendo comparar los resultados calculados en *batch*, teniendo la imagen completa, con los resultados en tiempo real.

Sin embargo, el principal problema de esta arquitectura es que requiere mantener un código que produzca el mismo resultado en dos sistemas distribuidos cuyo paradigma de programación es completamente diferente.

A parte del mantenimiento del código, otro gran inconveniente es el de la configuración y depuración de los sistemas. Como se verá más adelante en este trabajo, cada sistema, no solo entre su programación en *batch* o en tiempo real sino dentro del mismo paradigma, es un mundo completamente diferente y ser capaz de configurar estos sistemas con el fin de extraer todo su potencial es una tarea compleja que requiere mucho conocimiento de cada tecnología concreta.

### Arquitectura Kappa

La arquitectura Kappa nace de LinkedIn como una supuesta evolución a la arquitectura Lambda, es supuesta porque, dependiendo de a quién se lea, existen profesionales dentro del sector que opina que simplemente es una alternativa, mientras que otra piensa que es una evolución.

La arquitectura Kappa simplemente consta de la capa de procesamiento en tiempo real, desprendiéndose así de la capa de procesamiento en *batch* y, por lo tanto, se deshace del principal punto débil de la arquitectura Lambda, que es el mantenimiento de dos sistemas.

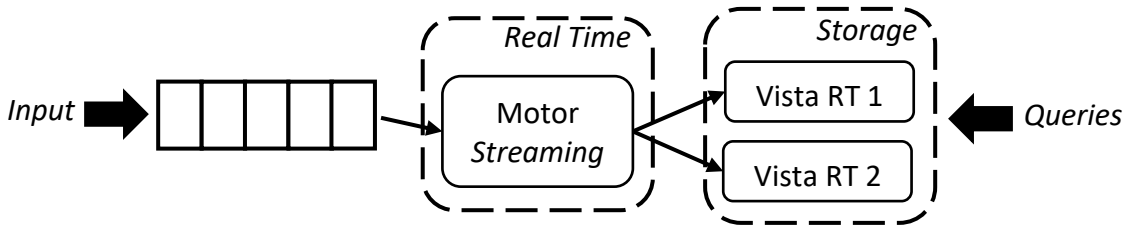


*Ilustración 2. Arquitectura Kappa*

Esta arquitectura simplifica bastante todo el sistema puesto que solo hay que mantener el sistema de tiempo real. Además, no pierde el beneficio de reprocesamiento de los datos que ofrece la arquitectura Lambda. El reprocesamiento solo es necesario en caso de que la lógica cambie o sea necesario añadir alguna nueva funcionalidad. Así que bastaría con lanzar un nuevo trabajo que consuma los datos desde el inicio y genere un nuevo resultado, de manera que simplemente habría que consumir los nuevos resultados desde la aplicación.

La principal ventaja respecto a la arquitectura Lambda es que basta con el desarrollo y mantenimiento de un único sistema, el de tiempo real. Sin embargo, puede haber ciertos casos muy concretos en los que el procesamiento en tiempo real y en *batch* no den los mismos resultados. Por ejemplo, en *machine learning* no se obtiene el mismo resultado utilizando un gran conjunto de datos de entrada de manera simultánea que ir insertando los datos de manera incremental, por ello, para cada caso de uso, la elección debe concretarse en función de los requisitos.

## Tecnologías Big Data

El avance de las tecnologías *Big* Data ha venido de la mano de las grandes empresas tecnológicas como LinkedIn, Facebook, Google… Cada una de ellas ha desarrollado sus propias soluciones para sus casos de uso. Por ejemplo, Twitter tenía la necesidad de tomar un conjunto de decisiones basadas en las interacciones de sus usuarios utilizando datos que acababan de ser creados y para ello creó Storm [11] o Facebook, que necesitaba un sistema de almacenamiento capaz de almacenar cantidades ingentes de datos estructurados distribuidos entre diferentes servidores y por ello creó Cassandra [12]. Además, la gran mayoría de ellas han sido liberadas bajo una licencia *open-source*, lo que ha dado lugar a una gran cantidad de soluciones disponibles para su uso donde cada una de ellas tiene unas características diferentes y, por lo tanto, unas pueden ser ideales para ciertos casos de uso mientras que para otros dan unos resultados pésimos.

Como se ha mencionado previamente, un *pipeline* que utiliza una arquitectura *Kappa* está compuesto de tres bloques principales, lo cual da lugar a diferentes tecnologías por cada uno de ellos. Por lo tanto, es necesario evaluar las posibles alternativas de los diferentes bloques para escoger las tecnologías a comparar.

### Gestor de colas

A diferencia de los otros dos módulos, actualmente existe una única tecnología indiscutible para el gestor de colas. Dicha tecnología es Apache Kafka, un gestor de colas desarrollado por LinkedIn [13] cuya popularidad no hace más que crecer como se puede observar en el análisis realizado por RedMonk [14]. En este análisis se demuestra un crecimiento exponencial de la popularidad de Kafka en sitios como en StackOverflow,

como se puede observar en la ilustración 3, la cual muestra el crecimiento del número de preguntas relativas a Kafka en StackOverflow.

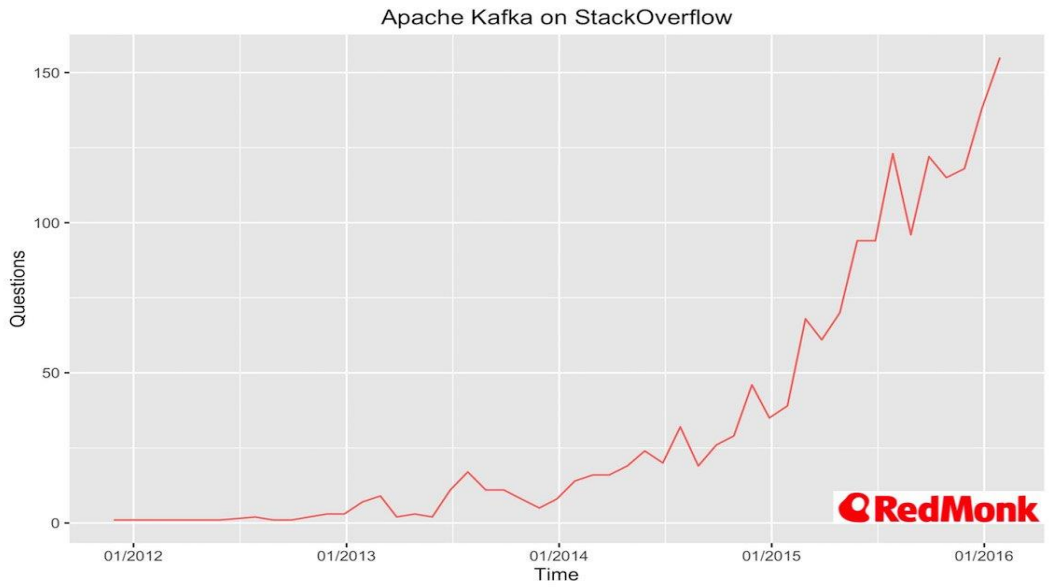


*Ilustración 3. Crecimiento del número de preguntas en StackOverflow relativas a Kafka. Gráfica realizada por RedMonk [14]*

Además, Apache Kafka está siendo utilizado en los *pipelines* de grandes compañías, como puede ser el de Yelp [15] o el de Pinterest [16] que utilizan Kafka como su gestor de colas.

Las únicas tecnologías medianamente cercanas a Apache Kafka podrían llegar a ser gestores de mensajes como RabbitMQ [17] o cualquier implementación del protocolo MQTT (*Message Queue Telemetry Transport*). Sin embargo, estas no aportan la facilidad de almacenar indefinidamente o durante una ventana temporal los mensajes recibidos, de manera que éstos puedan ser reprocesados en cualquier momento, lo cual es la principal ventaja de esta tecnología. Por otro lado, dichos gestores de mensajes gestionan el estado del consumidor del mensaje y utilizan un modelo de consumo de mensajes mediante *push,* mientras que Apache Kafka funciona mediante *polling*. Esto da una ventaja enorme a Apache Kafka en cuanto a rendimiento ya que este modo de funcionar le permite escalar horizontalmente.

### Motor de procesamiento en *Streaming*

Al contrario que el gestor de colas, el procesador de *streaming* presenta una gran variedad de alternativas. En la Tabla 1, podemos observar una comparativa de los principales motores de procesamiento que existen actualmente.

De todos ellos, en este trabajo se han escogido los siguientes:

- Spark nace del ecosistema Hadoop, como una evolución del procesamiento en *batch* que existía en HDFS. He elegido Spark debido a su gran facilidad para implementar un proceso debido al modelo *MapReduce* heredado de su antecesor. Es una de las soluciones más populares entre las empresas debido a su facilidad para integrarse con el ecosistema Hadoop.

| Nombre | Modelo computacional | Gestión del estado | Tolerancia a fallos | Orden de eventos |
|---|---|---|---|---|
| **Spark Streaming** | *Micro-batch* | *Stateful* | Exactly-once | Entre *batches* |
| **Storm** | Tuplas | *Stateless* | At-least-once | No |
| **Trident** | *Micro-batch* | *Stateful* | Exactly-once | Entre *batches* |
| **Heron** | Tuplas | *Stateless* | Exactly-once<br>At-most-once<br>At-least-once | No |
| **Flink** | Tuplas | *Stateful* | Exactly-one | No |
| **Samza** | Tuplas | *Stateful* | At-least once | En la partición |
| **MilWheel** | Tuplas | *Stateful* | Exactly-once | Si |
| **S-Store** | Tuplas | *Stateful* | Exactly-once | Sí, transacciones a nivel de proceso |

*Tabla 1. Tabla comparativa de motores de procesamiento en Streaming*

- Storm nace en Twitter bajo la necesidad de computar todos los flujos de información que posee dicha red social. Storm fue elegido debido a su gran rendimiento y sus bajas latencias, es además otra de las soluciones más elegidas por las empresas. La potencia de Storm en cuanto a latencias es prácticamente inigualable actualmente en el mercado.
- Por último, S-Store es una solución en desarrollo que se está realizando entre varias empresas estadounidenses entre las que están Brown y el MIT. Es un sistema que contiene tanto el almacenamiento como el procesamiento todo en uno pues utiliza el *core* de H-Store (la versión académica de VoltDB). S-Store ha sido elegido debido a que es el primero y único motor que garantiza un control transaccional a nivel de proceso. Es realmente interesante evaluar qué consecuencias puede llevar eso y si es capaz de estar a la altura de los procesadores de eventos.

### Base de datos

En este apartado existe una gran cantidad de soluciones puesto que los sistemas de almacenamiento que utilizan bases de datos llevan existiendo muchos años. La elección de la tecnología en este apartado depende mucho del caso de uso, el tipo de información a almacenar y el tipo de consultas que se van a realizar al sistema.

Sin embargo, la capacidad de cómputo en tiempo real que ofrecen estos sistemas limita la elección a aquéllos bajo el paradigma NewSQL y NoSQL, descartando completamente las relacionales tradicionales.

En este aspecto, se han contemplado las siguientes soluciones:

- MemSQL se ha elegido inicialmente debido a la similitud que tiene con H-Store, el *core* que utiliza S-Store y, por lo tanto, sería interesante compara el rendimiento de ambas. MemSQL es una base de datos NewSQL construida sobre el *core* de MySQL donde puedes definir tablas relacionales almacenadas en memoria y permitiendo además escalar añadiendo nodos. Sin embargo, esta solución tiene una limitación importante y es que sigue una arquitectura *master-slave* por lo que nunca va a llegar a tener una escalabilidad horizontal real.

- Por otro lado, tenemos Cassandra. Cassandra es una base de datos NoSQL basada en familias de columnas con una arquitectura *master-to-master*. La propia arquitectura de Cassandra le permite tener una gran escalabilidad y, además, su modelo de datos basado en familia de columnas hace que las búsquedas sean mediante *hashing*, obteniendo rendimientos muy altos. asimismo, gracias a su estructura interna, Cassandra está optimizada para poder realizar un gran volumen de escrituras por segundo. Por lo tanto, esto es una gran ventaja para estos sistemas puesto que lo más probable es que su principal interacción con el sistema de almacenamiento sea para escribir.
- También se ha utilizado Redis, una base de datos basada en un modelo clave-valor, cuyo rendimiento es muy alto gracias a su capacidad para gestionar la memoria del sistema. De hecho, esta base de datos se utiliza muchas veces como *caché* intermedia con otras bases de datos. Redis se ha utilizado debido a que se utilizaba en uno de los *benchmarks* ya implementados.
- Por último, S-Store se ha utilizado por obligación como almacenamiento, puesto que si queremos evaluar su rendimiento como procesador en *streaming* es necesario utilizarlo como almacenamiento también. El procesador en *streaming* funciona mediante los procedimientos almacenados de la base de datos, por lo que están altamente acoplados. Como se ha mencionado anteriormente, S-Store utiliza H-Store como motor de base de datos, el cual es la versión académica de VoltDB, por lo que es una base de datos NewSQL que almacena la información en memoria.

## Entorno tecnológico: descripción y configuración de las herramientas

Llevar a cabo el desarrollo de casos de uso simples, con configuraciones por defecto, es relativamente sencillo en estas tecnologías, el llegar a entender el efecto de cada uno de los parámetros de configuración para sacar el máximo rendimiento del sistema es una tarea compleja que lleva horas de trabajo y experiencia.

Por lo tanto, en este apartado se van a introducir conceptos básicos del funcionamiento de cada una de las herramientas utilizadas en este trabajo y cuáles son los parámetros de configuración necesarios para hacer funcionar un clúster de las mismas, así como los parámetros que más influencia tienen en el rendimiento del sistema.

Además, la mayoría de estas tecnologías tienen una documentación muy limitada. El hecho de que son plataformas *open-source* implica que colabore una gran cantidad de gente de diferentes entornos en el desarrollo de estas. Esta es la principal razón por la cual no existe una documentación clara y precisa.

Sin embargo, el hecho de ser *open-source* y de tener una comunidad, también significa que hay mucha gente dispuesta a solucionar problemas y resolver dudas. Y, por lo tanto, en cierto modo compensa a la falta de documentación.

### Apache Zookeeper

Aunque no haya sido mencionado anteriormente puesto que Apache Zookeeper no entra dentro de las tecnologías de procesamiento de datos, es necesario conocer su

funcionamiento puesto que muchas de las tecnologías mencionadas anteriormente requieren de un clúster de Zookeeper para su funcionamiento.

Apache Zookeper es un servicio centralizado capaz de coordinar sistemas distribuidos basándose en un protocolo de consenso que utiliza votaciones para elegir a un líder. Asimismo, no solo permite coordinar sistemas, sino que también puede almacenar información relativa a la configuración del sistema. Por ejemplo, Apache Kafka utiliza Zookeeper para coordinar el clúster y guardar los metadatos del sistema necesarios para que este funcione.

Zookeeper puede funcionar tanto en una sola máquina como en varias máquinas simultáneamente en modo clúster con el fin de garantizar su disponibilidad. Debido al protocolo de sincronización del clúster de Zookeeper, es recomendable utilizar un número impar de nodos ya que requiere de un mínimo del 50% de los nodos del clúster activos para su funcionamiento. Por ejemplo, si tenemos 6 nodos en el clúster, necesitamos 3 para el correcto funcionamiento del mismo, esto permitiría 3 fallos, pero en el caso de tener 5 nodos, el clúster podría funcionar con 2 nodos y también permitiría 3 fallos, así que un sexto nodo no aporta ningún valor. De hecho, recomiendan utilizar 3, 5 o 7 nodos, ya que a partir de 7 el rendimiento general del clúster se ve reducido.

### Configuración de un clúster

Para configurar un clúster de Zookeeper primero es necesario decidir cuántos nodos se van a utilizar. Esto depende principalmente de tres aspectos: uno, el nivel de tolerancia a fallos que necesitemos en el sistema; dos, el número de recursos de los que disponemos; tres, el número de nodos que Zookeeper se va a encargar de coordinar. En este caso tomaremos tres nodos como ejemplo, que es el mínimo número de nodos para soportar un fallo en uno de ellos. Por lo tanto, este ejemplo tomará como configuración tres nodos cuyas direcciones son:

- Nodo 1: 11.11.11.11:2181
- Nodo 2: 22.22.22.22:2181
- Nodo 3: 33.33.33.33:2181

Zookeeper requiere asignar un número único a cada uno de los nodos para identificarlos. En este caso se utilizará el número especificado en la enumeración anterior. Una vez asignados los números que identifican a los nodos, es necesario modificar el fichero de configuración en todos los nodos con los siguientes valores:

```
clientPort=2181
dataDir=/tmp/zookeeper #o el directorio deseado
server.1=11.11.11.11:2888:3888
server.2=22.22.22.22:2888:3888
server.3=33.33.33.33:2888:3888
```

Ésta sería la configuración mínima para que los tres nodos se conectaran entre sí formando un clúster. Una vez modificado el fichero de configuración hay que crear el directorio '*dataDir*' que se ha elegido en la configuración.

```
mkdir /tmp/zookeeper
```

Después hay que crear un fichero denominado '*myid*' con el número del nodo establecido en la configuración dentro de dicho directorio, por ejemplo, el nodo 1 tendrá un fichero 'myid' con un 1 y así con cada uno de los nodos.

```
echo '1' >> /tmp/zookeeper/myid
```

Con esto ya estaría configurado el clúster de zookeeper y bastaría con ejecutar el siguiente comando en cada uno de los nodos para hacer funcionar el clúster:

```
<ZK_DIR>/bin/zkServer.sh start <ZK_DIR>/config/zookeeper.properties
```

## Apache Kafka

Como se ha mencionado anteriormente, el núcleo de Apache Kafka es un gestor de colas que aporte a los sistemas de procesamiento de datos una capacidad de reprocesamiento y elasticidad que permita al sistema poder corregir errores y evolucionar con el mínimo impacto posible [18].

Actualmente, Apache Kafka es más que un gestor de colas puesto que se ha transformado en una plataforma de *streaming* con un motor de procesamiento en *streaming* y otras herramientas que facilitan el trabajo a los desarrolladores. Sin embargo, cuando se empezó a realizar este trabajo Kafka no era más que un gestor de colas, así que ha sido el único elemento de toda la plataforma que se ha utilizado en este trabajo. Por lo tanto, se va a explicar únicamente el funcionamiento de dicho gestor.

Conceptualmente Apache Kafka no es más que un conjunto de *logs* distribuidos, también conocidos como *topics*, a lo largo de los nodos de un clúster. Para facilitar la distribución del log, Kafka realiza una partición de los *topics* siguiendo una estrategia especificada por el usuario.

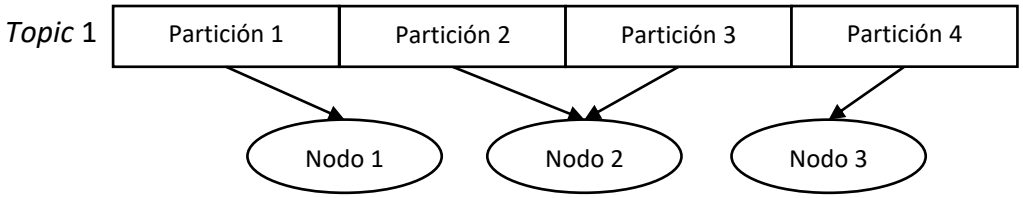


*Ilustración 4. Particionamiento de los topics de Kafka*

Además, para garantizar la disponibilidad de dichos *topics*, Apache Kafka permite realizar una replicación de los mismos repartiendo las réplicas entre diferentes nodos del clúster. El factor de replicación también es configurable por el usuario y su valor dependerá de la tolerancia a fallos deseada por el usuario, aunque puede conllevar una pérdida de rendimiento según su configuración.

Tanto para generar datos como para consumirlos de los *topics*, Kafka facilita dos APIs, una de productor (*producer*) y otra de consumidor (*consumer*). La simplicidad de uso de un log de mensajes hace que tanto leer como escribir datos en los *topics* tenga un rendimiento muy alto y además sea sencillo de utilizar.

La API del productor, la encargada de escribir datos al *topic*, se encarga de hacer lo que sería hacer un "*append*" a un *log* o lo que es lo mismo, añadir el dato al final. Por lo tanto, esto provoca que se guarden los mensajes según su orden de llegada facilitando así su procesamiento.

El funcionamiento del API del consumidor es un poco más complejo debido a que permite crear grupos de consumidores. Un grupo de consumidores es un conjunto de aplicaciones que consumen de un *topic* con un mismo 'id'. De esta manera, se reparten las particiones entre los diferentes nodos con el fin de paralelizar el trabajo. No obstante, a parte de esta peculiaridad, que es transparente de cara al usuario, el uso del API es muy sencillo ya que simplemente consta de un *offset* que avanza a lo largo del

*topic*. Este comportamiento provoca el consumo de los mensajes del *topic* de una manera ordenada dentro de cada partición.

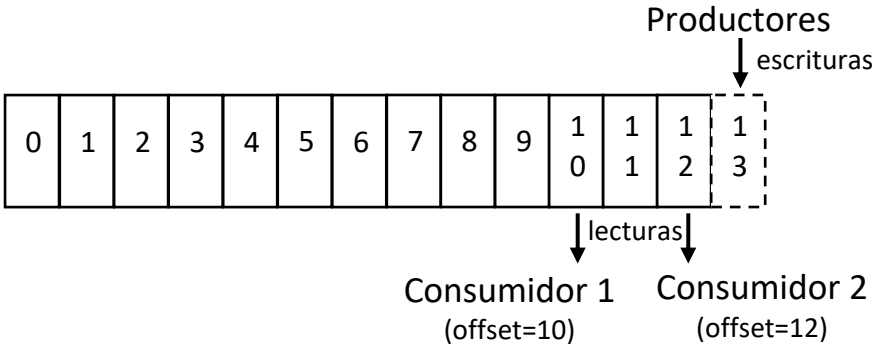


*Ilustración 5. Funcionamiento de las APIs de Apache Kafka*

## Configuración de un clúster

Una vez entendido el funcionamiento básico de Kafka, podemos proceder a desplegar un clúster del mismo. Kafka hace uso de Apache Zookeeper para coordinar los diferentes nodos del clúster y almacenar los metadatos del sistema. En Zookeeper se gestionan los *topics*, los *offsets* leídos por cada consumidor y otros datos que sirven para coordinar el clúster y elegir a un nodo máster. Por lo tanto, es necesario desplegar un nodo o un clúster de Zookeeper (según la disponibilidad y tolerancia a fallos que deseemos) para que Kafka pueda funcionar.

Así que el primer paso es configurar Zookeeper el cual viene ya incluido en la distribución de Kafka, aunque se puede utilizar la versión '*standalone*' si ya tienes un clúster corriendo y lo quieres aprovechar o quieres utilizar otra versión. Como ya se ha explicado en el apartado anterior cómo configurar un clúster de Zookeeper, supongamos que ya tenemos un clúster de 3 nodos de Zookeeper con las siguientes direcciones:

- Nodo 1: 11.11.11.11:2181
- Nodo 2: 22.22.22.22:2181
- Nodo 3: 33.33.33.33:2181

Una vez configurado el clúster de Zookeeper podemos configurar el clúster de Kafka. No hay nada que impida instalar el clúster de Kafka en los mismos nodos que está el clúster de Zookeeper instalado. A pesar de ello, no es recomendable que compartan máquina porque hacen un uso intensivo del disco y pueden llegar a competir por él y reducirse el rendimiento drásticamente en ciertas situaciones. En caso de instalar ambos clústeres en las mismas máquinas, es recomendable que cada nodo tenga al menos un disco dedicado a cada uno de los sistemas, de manera que no compitan entre sí.

Siguiendo las recomendaciones, elegimos tres nodos diferentes para Kafka, cuyas direcciones serán las siguientes:

- Nodo 4: 44.44.44.44
- Nodo 5: 55.55.55.55
- Nodo 6: 66.66.66.66

Para configurar el clúster es necesario modificar el fichero de configuración de cada uno de los nodos. En este caso vamos a tomar como ejemplo el nodo 4, pero para el resto de los nodos el fichero de configuración será similar, cambiando la dirección del nodo.

```
broker.id = 1 #el número del nodo, tiene que ser único en el clúster
port = 9092
host.name = 44.44.44.44 #dirección del nodo
zookeeper.connect = 11.11.11.11:2181, 22.22.22.22:2181, 33.33.33.33:2181
```

En las últimas versiones de Kafka se han convertido los parámetros '*port*' y '*host.name*' en obsoletos, pasando a estar '*deprecated*' aunque se mantengan por compatibilidad. En estas versiones habría que sustituir dichos parámetros por el parámetro '*listeners*' que toma como valor una lista de direcciones más un protocolo, separados por comas. Por ejemplo, en este caso, el fichero quedaría de la siguiente manera.

```
broker.id = 1 #el número del nodo, tiene que ser único en el clúster
listeners = PLAINTEXT://44.44.44.44:9092
zookeeper.connect = 11.11.11.11:2181, 22.22.22.22:2181, 33.33.33.33:2181
```

En este caso PLAINTEXT sería el protocolo, pero Kafka soporta diferentes protocolos, como puede ser el SSL para realizar conexiones seguras a cada uno de los nodos por parte de los clientes.

Una vez configurado cada uno de los nodos, ya estarían listos para iniciar el clúster con la mínima configuración posible y asumiendo el resto de los parámetros por defecto.

Para iniciar el clúster de Kafka habría que ejecutar el siguiente comando en cada uno de los nodos, asumiendo que el clúster de Zookeeper está funcionando correctamente y en ejecución.

```
<KAFKA_DIR>/bin/kafka-server-start.sh <KAFKA_DIR>/config/server.properties
```

### Parámetros de configuración más relevantes

De todas las tecnologías evaluadas en este trabajo, Kafka probablemente sea la que más parámetros de configuración tiene. Y es que tiene parámetros de configuración tanto para los *brokers* y los *topics* como para los clientes que consumen (*consumer*) y los que inyectan datos (*producer*). Debido a la necesidad de configurar dichos elementos, existen más de 100 parámetros de configuración que se pueden ajustar.

Afortunadamente, los parámetros por defecto en Kafka están muy bien ajustados y en la mayoría de los casos no es necesario modificar ninguno de ellos. Sin embargo, hay ciertos parámetros que tienen un impacto considerable en el rendimiento del sistema.

El primero y más importante de todos es el número de particiones de un *topic*. El particionado en Kafka equivale a su paralelismo, y, es que, si solo existe una partición, el consumo de los datos se realizará de manera secuencial. Por lo tanto, en el caso de que Kafka fuera el factor limitante de nuestro sistema, aumentando el número de particiones (siempre y cuando los *consumers* puedan operar de ese modo), aumentaríamos el rendimiento del sistema de manera considerable.

Esto se debe a que el consumo de datos de Kafka se realiza utilizando un ID, por lo que si varios *consumers* (pueden pertenecer a la misma aplicación ejecutándose en diferentes *threads*) consumen datos del mismo *topic* simultáneamente, el propio Kafka se encargará de repartir las particiones entre los diferentes *consumers* de manera que se reparta la carga entre todos. Obviamente realizar particiones conlleva sus

consecuencias puesto que Kafka mantiene el orden de los datos dentro de la misma partición, sin embargo, no garantiza el orden de los datos entre particiones. Por lo tanto, hay que tener mucho cuidado a la hora de realizar particiones ya que puede influir directamente en el comportamiento de la aplicación.

Otro parámetro a tener en cuenta es la compresión de los datos, aunque no tenga tanto impacto en el rendimiento como el anterior, este parámetro puede ayudar mucho a la hora de aprovechar los recursos de almacenamiento y red. Como los *brokers* de Kafka no tienen constancia de los datos que entran y salen, para ellos simplemente entran y salen secuencias de bytes, la compresión la realiza el *producer* y la descompresión la realiza el *consumer*. Como se puede observar en los resultados obtenidos de este *benchmark* [19], utilizar la compresión no va a incrementar el *throughput* del sistema puesto que hay que realizar más cómputo en ambos extremos, pero puede llegar a reducir considerablemente el uso de red y disco del sistema. Por lo tanto, en caso de que el factor determinante del sistema sea la red o el disco, es muy probable que utilizar la compresión lz4, que maximiza el rendimiento, pueda llegar a aumentar el *throughput*.

Por otro lado, está el parámetro de configuración del tamaño del *batch* de datos que genera el *producer*. Kafka inserta los datos en los *topics* mediante *batches*, y para ello utiliza el parámetro '*batch.size*'. Como en todos los sistemas, este parámetro presenta un compromiso entre *throughput* y latencia, a mayor tamaño del *batch*, mayor *throughput* y mayor latencia mientras que cuanto menor sea el *batch*, ocurrirá lo contrario. El parámetro similar en la parte del *consumer* sería el '*fetch.min.bytes*', que define el mínimo número de bytes necesarios para realizar una lectura. Por defecto este parámetro está a 1, de manera que cada vez que hay un dato escrito, el *consumer* lo va a leer. Sin embargo, se puede aumentar este valor para leer varios mensajes en una única lectura aumentando así el *throughput* del sistema a costa de aumentar también la latencia.

### Apache Storm

Como se ha comentado previamente, Apache Storm es un motor de procesamiento de datos en tiempo real capaz de llegar a tener latencias de unos pocos milisegundos. Sin embargo, este rendimiento conlleva utilizar una política '*at-least-once*' y un procesamiento a nivel de tupla, por lo tanto, sería necesario contemplar el reprocesamiento de los datos ya consumidos en caso de fallo.

Para comprender el funcionamiento de Storm, es necesario conocer de qué elementos lógicos están compuestas sus aplicaciones. Las aplicaciones en Storm cuentan de tres elementos principales:

- ***Bolts.*** Los *bolts* son clases Java que incluyen toda la lógica de la aplicación, es decir, realizan todo el procesamiento de los datos. Pueden realizar prácticamente cualquier función, desde filtrado y agregaciones hasta consultas en bases de datos. Una aplicación de Storm suele estar compuesta por varios *Bolts*, cada cual suele implementar una función concreta. Un *bolt* recibe como datos de entrada una tupla procedente de otro elemento de Storm y puede emitir una o varias tuplas como resultado en caso de que sea necesario.
- ***Spouts***. Los *spouts* son clases Java encargadas de generar/obtener los datos para luego ser procesados. Normalmente los *spouts* van a conectarse a una fuente externa de datos, como puede ser un gestor de colas o una base de datos. Emiten

una o varias tuplas con los datos obtenidos para luego ser consumidos por cualquier *bolt*.

- ***Topology***. Una topología o *topology* es el núcleo de una aplicación de Storm, en ella se declaran todos los elementos de la aplicación y las interacciones entre ellos. De esta manera, una topología no es más que un grafo de *spouts* y *bolts* conectados entre sí formando un flujo de datos. Las topologías en Storm son ejecutadas indefinidamente y no son finalizadas hasta que haya un error en ella o se haga explícitamente realizando un *kill* mediante la interfaz de administración de Storm.

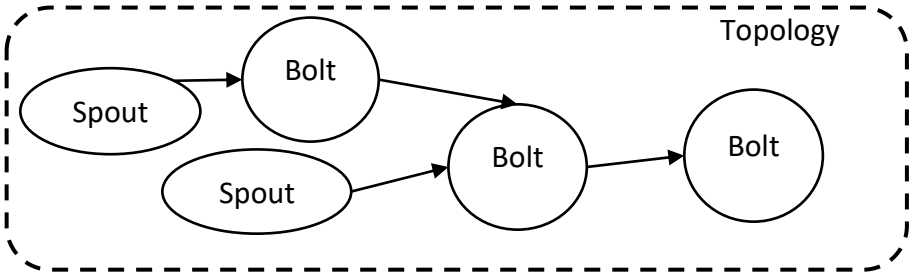


*Ilustración 6. Diagrama de ejemplo de los elementos de Storm*

A parte de los elementos mencionados anteriormente, Storm tiene dos elementos que definen la capacidad de paralelización que tiene una aplicación.

Por un lado, existen las *tasks* las cuales corresponden a un *thread* en el sistema operativo y se encargan de ejecutar la lógica de un *bolt* o un *spout* concreto. Cada *bolt* o *spout* puede tener asignadas muchas *tasks* con el fin de paralelizar su ejecución. Este es el principal parámetro para definir el paralelismo de una aplicación, que además es independiente por cada *bolt*/*spout*, de manera que se pueden asignar más *threads* a las tareas más costosas.

Por otro lado, los *workers* son instancias físicas de una máquina virtual Java y que ejecutan un subconjunto del total de *tasks* de la aplicación. Normalmente Storm intenta repartir equitativamente el número de *tasks* entre los diferentes *workers*, de manera que la carga esté repartida de manera homogénea. Por ejemplo, si tenemos 50 *tasks* y 10 *workers*, se ejecutarían 5 *tasks* por cada *worker*. Se pueden desplegar tantos *workers* por nodo como se desee, aunque no es recomendable tener muchos ya que competirán por los recursos físicos del nodo.

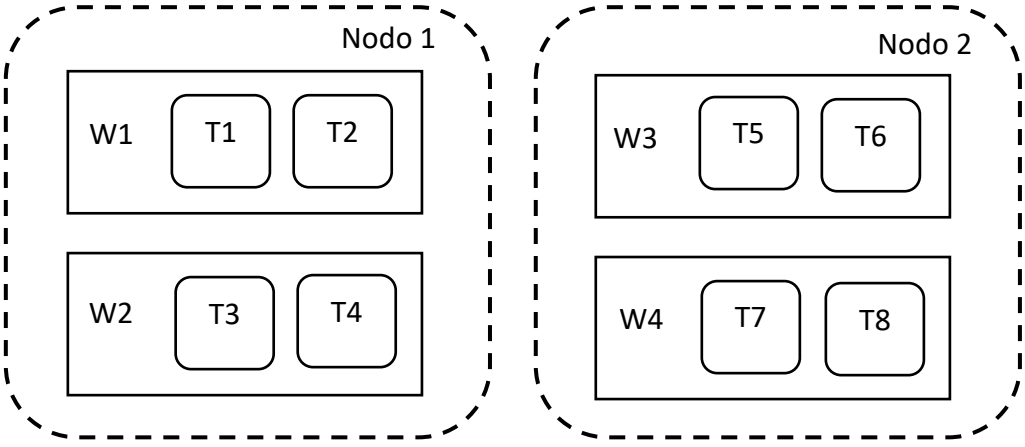


*Ilustración 7. Diagrama de ejemplo de los elementos de paralelización en Storm*

Aunque no se haya hecho uso de ella en este trabajo, Storm también facilita una capa de abstracción construida sobre todos los elementos mencionados anteriormente con el fin de facilitar el desarrollo de las aplicaciones. Esta abstracción se llama Trident y al contrario que Storm, tiene una política '*exactly-once*' por lo que el desarrollador de la aplicación no necesitará preocuparse por la duplicación de los datos provocada por un problema de replicación en el gestor de colas o la lectura del mismo dato.

En dicha abstracción se utilizan tanto *streams* como *states* y además permite utilizar elementos de Storm como los *spouts*. Por un lado, un *state* representa un sistema interno o externo a Storm, capaz de almacenar datos y con el que Trident interactuará tanto para leer datos como para guardarlos. Por otro lado, un *stream* no es más que un conjunto de transformaciones (filtrados, agregaciones, agrupaciones…) sobre un flujo de datos entrante. En este caso, un *stream* sería el equivalente a un conjunto de *bolts* que realicen dichas transformaciones sobre los datos entrantes.

### Configuración de un clúster

Al igual que Kafka, Storm utiliza Zookeeper para coordinar los diferentes nodos del clúster y almacenar los metadatos. Por lo tanto, para hacer funcionar Storm, es necesario tener un clúster de Zookeeper configurado y en funcionamiento. Para ello vamos a partir de la misma configuración que se ha utilizado con Kafka anteriormente, utilizando un clúster de Zookeeper de 3 nodos cuyas direcciones son las siguientes:

- Nodo 1: 11.11.11.11:2181
- Nodo 2: 22.22.22.22:2181
- Nodo 3: 33.33.33.33:2181

Cabe señalar que no es recomendable utilizar el mismo clúster de Zookeeper para Kafka y Storm a la vez debido a razones de rendimiento, así como tampoco es recomendable que el clúster de Zookeeper y el de Storm compartan recursos por las mismas razones que ocurría con Kafka. Aunque en el caso de Storm son más compatibles ya que Storm utiliza principalmente el procesador y la memoria mientras que Zookeeper hace mucho uso del disco.

Una vez desplegado el clúster de Zookeeper podemos comenzar a configurar el clúster de Storm. Al igual que en el resto de los sistemas de este trabajo, la configuración se realiza mediante un fichero, que en este caso está en formato 'yaml'.

Antes de escribir la configuración en dicho fichero, es necesario conocer los tres tipos de procesos que se pueden ejecutar en Storm a nivel de sistema en cada nodo.

- El *nimbus*, que sería el proceso encargado de realizar la función de *master* en el clúster. Puede haber tantos procesos *nimbus* como nodos en el clúster hay, de manera que existirá uno activo como *master* y el resto estarán a la escucha pendientes de si el nodo principal falla.
- El *supervisor* es el proceso encargado de administrar y ejecutar los *workers* en un nodo. Por lo tanto, si no existe el proceso *supervisor* en un nodo, éste no tendrá ninguna capacidad de procesamiento en Storm. Cada puerto configurado en el *supervisor* equivale a un *worker* en el nodo.
- El *ui* es un proceso encargado de desplegar una interfaz web en el puerto 8080, desde la que se pueden realizar tareas de monitorización y administración. Este

proceso no es necesario para el funcionamiento de Storm, pero sí muy recomendable para conocer el estado del sistema.

Vistos los tipos de procesos existentes, se va a realizar un ejemplo de despliegue con tres nodos, donde cada nodo va a tener una instancia de *nimbus* y otra de *supervisor* con dos *workers*, mientras que se instalará únicamente en uno de los nodos la interfaz web. Para realizar dicha configuración se utilizarán los siguientes nodos:

- Nodo 4: 14.14.14.14
- Nodo 5: 25.25.25.25
- Nodo 6: 36.36.36.36

De esta manera, el fichero de configuración en cada uno de los nodos quedaría de la siguiente manera:

```
storm.zookeeper.servers:
 - "11.11.11.11"
 - "22.22.22.22"
 - "33.33.33.33"
storm.local.dir: "/tmp/storm" #el directorio de datos que utilizará Storm
nimbus.seeds: ["14.14.14.14","25.25.25.25","36.36.36.36"] #nodos candidatos a master
supervisor.slots.ports:    #puertos de cada uno de los workers del nodo
   - 6700
   - 6701
```

Una vez configurados cada uno de los nodos bastaría con ejecutar los procesos correspondientes en cada nodo. El ejecutable 'storm´ nos permite lanzar todos los procesos dependiendo de la variable pasada como parámetro. Así que para ejecutar el *nimbus*, habría que ejecutar el siguiente comando.

```
<STORM_DIR>/bin/storm nimbus
```

Para ejecutar el supervisor sería el siguiente comando.

```
<STORM_DIR>/bin/storm supervisor
```

Y, por último, para ejecutar la interfaz web, habría que ejecutar el siguiente comando.

```
<STORM_DIR>/bin/storm ui
```

De esta manera, ejecutando dichos procesos en cada uno de los nodos correspondientes, tendríamos un clúster de Storm listo para ser utilizado.

### Parámetros de configuración más relevantes

Storm probablemente sea el sistema más sencillo de optimizar de los utilizados en este trabajo. Y es que, aparte de no tener una gran cantidad de parámetros de configuración, ofrece una interfaz de usuario que aporta mucha información sobre el proceso que se está ejecutando, lo cual facilita el trabajo de optimizar el proceso.

El principal parámetro a tener en cuenta en Storm es el paralelismo de cada uno de los elementos de las topologías, es decir, los *bolts* y *spouts*. Storm permite elegir la cantidad de *threads* que se van a ejecutar por cada uno de los elementos de una topología. De esta manera, si la lógica de la aplicación se ha diseñado correctamente de manera modular, es muy sencillo ajustar el paralelismo de cada elemento para obtener el mejor rendimiento posible.

Para aprovechar al completo los recursos del sistema, es necesario ejecutar como mínimo un *thread* por cada *core* asignado a un *worker*, de manera que cada *thread* pueda ser ejecutado en un *core*. Así que lo recomendable es calcular el número de *cores* disponibles para todo el sistema y repartirlos entre los diferentes elementos.

Con el fin de obtener un rendimiento óptimo, es necesario repartir los *threads* correctamente en función de la carga que suponga cada elemento. Para ello es recomendable ejecutar la aplicación y observar su comportamiento desde la interfaz web de Storm. En esta interfaz se puede observar la latencia de cada elemento y la cantidad de tuplas que ha consumido y emitido. Por lo tanto, a partir de dicha información es fácil evaluar cuál es el cuello de botella de la aplicación para poder asignarle un mayor número de *threads* a esa tarea.

Por otro lado, existe un parámetro muy importante que evita que el sistema se sature. El parámetro '*topology.max.spout.pending*', define la cantidad máxima de tuplas que el sistema permite almacenar en el *buffer* para ser consumidas. Es decir, que, si un *spout* emite tuplas más rápido de lo que los *bolts* son capaces de procesarlas, entonces los buffers se irán llenando a la espera de que los *bolts* puedan procesarlos. Este parámetro permite establecer un máximo de datos en los *buffers*, de manera que cuando se llega a ese límite, los *spouts* dejan de emitir tuplas en el sistema hasta que los *buffers* se vayan liberando. Este parámetro no afecta directamente al rendimiento, pero sí a la estabilidad del sistema, ya que, con un parámetro demasiado alto, puede llegar a perderse información e incluso a sobrecargar el sistema llenando la memoria.

### Apache Spark

Apache Spark es la evolución natural del procesamiento en *batch* del ecosistema Hadoop. De hecho, el propio motor de Spark hereda el modelo de ejecución en *batch* mediante MapReduce. La diferencia entre uno y otro es que Hadoop funciona sobre el sistema de ficheros HDFS y, por lo tanto, hace mucho uso de disco mientras que Spark aprovecha el uso de la memoria RAM obteniendo un mejor rendimiento.

Spark está compuesto por diversos *frameworks* que permiten al sistema ser muy versátil ya que pueden interactuar entre sí aportando cada uno un valor diferente. Tiene cuatro *frameworks* principales:

- **Spark SQL**. Permite ejecutar sentencias SQL para realizar transformaciones, filtrado, agregaciones… sobre los datos con los que trabaje la aplicación.
- **Spark Streaming**. Es el *framework* encargado del procesamiento de datos en tiempo real a pesar de tener por debajo un motor de procesamiento en *batch*. No obstante, el hecho de tener que ser compatible con Spark, provoca que su modelo de computación sea mediante *micro-batches* y utilice el modelo de programación MapReduce. Esto da lugar a latencias de procesamiento a nivel de segundos, aunque puede llegar a dar muy buen *throughput* dependiendo del caso de uso.
- **MLib**. Es el *framework* que permite a Spark aplicar *machine learning* sobre los datos utilizados en la aplicación.
- **GraphX**. Este se encarga de realizar operaciones y cómputo sobre datos cuya estructura sea la de un grafo, dando así mayor importancia a las relaciones entre ellos.

En este trabajo únicamente se ha utilizado el *framework* de Spark Streaming puesto que el resto no han sido necesarios para el *benchmark*. Para comprender mejor el funcionamiento de Spark Streaming, es necesario conocer qué es un *resilient distributed dataset* (RDD), el cual es la abstracción de los datos en Spark. Un RDD es un conjunto de datos particionados y distribuidos a lo largo de un clúster de Spark. Cada transformación realizada sobre un RDD en Spark retorna un RDD nuevo con la transformación realizada, por lo tanto, cada RDD es inmutable una vez es inicializado.

Debido a que es complicado representar un flujo de datos mediante un RDD, ya que no crece en el tiempo, Spark Streaming define su propia abstracción de los datos, los *discretized streams* (DStreams). Los DStreams representan un flujo de datos continuo. Este flujo está representado mediante un conjunto de RDDs donde cada RDD representa una ventana temporal que coincidiría con el *micro-batch* mencionado anteriormente. Al igual que los RDDs, los DStreams son inmutables y, por lo tanto, las transformaciones sobre los mismos retornan un nuevo DStream.

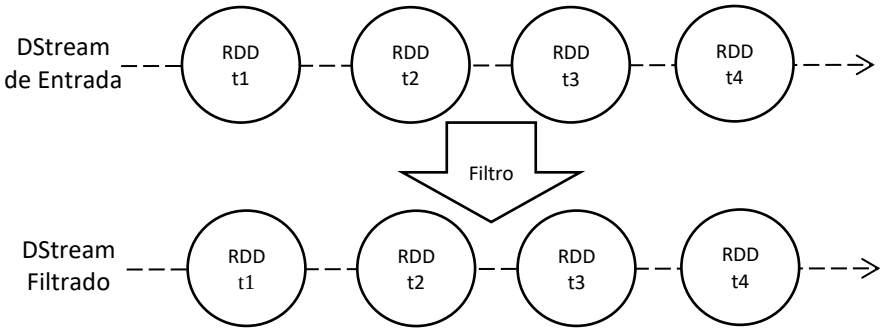


*Ilustración 8. Representación de un ejemplo de transformación de un DStream de Spark Streaming*

Por lo tanto, al igual que ocurría con Trident en Storm, una aplicación de Spark Streaming no es más que un conjunto de transformaciones sobre un flujo de datos. Esto hace que la programación de una aplicación sea relativamente sencilla, aunque conlleva un gran esfuerzo inicial por comprender bien el paradigma MapReduce y aprender a programar fluidamente con él.

### Configuración de un clúster

Spark permite tres sistemas de gestión de recursos diferentes: Yarn, Mesos y el propio Spark. Cada uno de ellos tiene sus ventajas e inconvenientes, aunque el más utilizado, principalmente a nivel empresarial es Yarn. Sin embargo, en este trabajo se ha utilizado el gestor propio de Spark por su simplicidad de instalación y porque al no ser un entorno de producción no es necesario hacer uso de las ventajas que aportan los otros gestores como la tolerancia a fallos.

Por lo tanto, para crear un clúster de Spark utilizando su gestor de recursos propio bastaría con crear una instancia *master* en un nodo con el siguiente comando.

```
<SPARK_DIR>/sbin/start-master.sh
```

Con esto se lanzará una instancia máster que se encargará de realizar las tareas del driver de Spark coordinando así el clúster y de proporcionar una interfaz web de monitorización y administración accediendo a la URL 12.12.12.12:8080, siendo '12.12.12.12' la dirección IP del nodo máster.

Una vez ejecutado el nodo *master* basta con lanzar instancias de *slaves* en tantos nodos como deseemos. Para ello, será necesario desplegar las instancias indicándolas la dirección de acceso al *master* para que este sea capaz de coordinar cada uno de los nodos. De esta manera, habría que ejecutar el siguiente comando.

```
<SPARK_DIR>/sbin/start-slave.sh spark://12.12.12.12:7077
```

Una vez ejecutado este comando en cada uno de los nodos ya tendríamos un clúster de Spark funcional con un nodo *master* y tantos nodos *slaves* como sean necesarios.

### Parámetros de configuración más relevantes

El parámetro más importante en Spark Streaming es el tamaño de la ventana. El modelo de procesamiento de Spark Streaming es en forma de *batch* y, por lo tanto, es necesario definir el tamaño de dicho *batch*. Esto se define mediante el tamaño de la ventana deslizante, la cual genera un *batch* periódicamente cada vez que finaliza una ventana. Por regla general, a mayor sea el tamaño del *batch*, mayor será el *throughput* y a su vez mayor será la latencia. No obstante, no siempre es así, si el procesamiento realizado en la aplicación es a nivel de tupla, en vez de a nivel de *batch*, entonces reducir el tamaño del *batch* puede llegar a dar mejores resultados.

Por otro lado, por defecto Spark únicamente asigna 1GB de memoria a los procesos que realizan el cómputo de la aplicación en cada nodo. Por lo tanto, si solo se ejecuta una aplicación dentro de un clúster y cada nodo tiene más de 1GB de memoria, entonces los recursos están infrautilizados. Así que es necesario asignar en el fichero de configuración de Spark el parámetro '*spark.executor.memory*' acorde con los recursos disponibles en el clúster.

Por último, al igual que en Storm, es necesario limitar los datos que entran en el sistema para ser procesados en caso de que el cómputo no sea capaz de procesar al mismo ritmo al que entran datos al sistema. Así que es necesario activar el modo *backpressure*, este modo se encarga de frenar el consumo de datos ajustándose automáticamente en función de lo que es capaz la aplicación de consumir.

## Apache Cassandra

Apache Cassandra es una base de datos NoSQL que utiliza un modelo basado en familias de columnas y una arquitectura *master-to-master*.

La familia de columnas sigue una estructura similar al modelo relacional puesto que utiliza tablas cuyas filas están identificadas con una *primary key*. Sin embargo, la familia de columnas permite que cada fila dentro de una tabla tenga un número variable de columnas, evitando así los nulos y dando cierta flexibilidad al modelo de datos. En Cassandra tampoco existen las *foreign key*, por lo tanto, las relaciones tienen que ser gestionadas por la aplicación.

En Cassandra no es muy recomendable utilizar índices, puesto que el rendimiento que ofrecen no es comparable a la redundancia de una la familia de columnas utilizando otra *primary key*. Por lo tanto, es muy importante diseñar el modelo físico de los datos teniendo esto en mente, de hecho, es recomendable hacer las búsquedas exclusivamente utilizando la *primary key*. Además, la *primary key* está dividida en dos partes. La primera parte es la *sharding key*, esta clave se utiliza para realizar el particionado de los datos para poder repartirlos entre todos los nodos del clúster

mediante rangos o *hashing*. La segunda parte es la *clustering key*, esta parte de la clave se encarga de ordenar los datos dentro de cada partición. Esto se ha de tener en cuenta también en la fase de diseño y atendiendo a las consultas frecuentes que se vayan a realizar.

Por otro lado, el modelo *master-to-master*, permite que todos los nodos de un clúster de Cassandra puedan gestionar consultas redirigiendo la petición al nodo donde se encuentre la partición donde está almacenado el dato correspondiente, de esta manera permite una gran escalabilidad horizontal añadiendo nodos al clúster.

### Configuración de un clúster

Configurar un clúster de Cassandra es relativamente sencillo puesto que solo requiere modificar tres parámetros de configuración.

Lo primero es necesario darle un nombre al clúster y asignárselo mediante el parámetro '*cluster_name*' a todos los nodos, el nombre del clúster se podría considerar como un ID del clúster, por lo tanto, es necesario asignárselo a todos los nodos.

Una vez configurado el nombre del clúster es necesario asignar en cada nodo o bien su interfaz de red (utilizando el parámetro '*listen_interface*') o bien su dirección IP (utilizando el parámetro '*listen_address*') mediante la cual se comunicará cada nodo con el resto.

Ya configurada la dirección de acceso de cada nodo, basta con asignar al parámetro '*seeds*' una lista con las direcciones de los nodos del clúster. Cabe destacar que, una vez creado y arrancado el clúster, no es necesario modificar este parámetro en caso de que se añadan nuevos nodos al clúster, el propio Cassandra se encarga de gestionarlo.

## MemSQL

MemSQL es una base de datos NewSQL que se podría considerar como una evolución de MySQL puesto que utiliza el mismo núcleo.

MemSQL utiliza una arquitectura *master-slave*, donde los nodos *master* se encargan de coordinar las consultas y realizar las agregaciones, de hecho, son conocidos como '*master aggregator*'. Mientras que los nodos *slave*, son los que contienen la información y ejecutan las consultas.

Con el fin de aportar cierta escalabilidad al sistema, MemSQL permite utilizar varios nodos *master*, los cuales atenderán cualquier tipo de consulta, facilitando así el balanceo de carga. Además, realiza un particionado de los datos entre los nodos *slave*, de manera que cada nodo *slave* ejecuta la consulta en paralelo retornando los resultados al nodo *master* repartiendo así la carga de la consulta.

Por último, MemSQL permite dos tipos de modelos físicos diferentes: uno modelo relacional, el cual se almacena exclusivamente en memoria y optimizado para datos en tiempo real y un modelo columnar, almacenado en disco el cual está optimizado para consultas sobre históricos.

El conjunto de estos dos tipos de modelos aporta a MemSQL una gran flexibilidad y rendimiento siempre y cuando utilices el modelo relacional para datos de tiempo real y el modelo columnar para datos históricos.

Al contrario que otras tecnologías utilizadas en este trabajo, MemSQL tiene una versión comercial con más funcionalidades que la versión *open-source*. Por lo tanto, también tiene una documentación extensa con mucha información, aunque, por lo contrario, carece de comunidad en la red para resolver cualquier tipo de problema.

### Configuración de un clúster

En este caso MemSQL no se ha utilizado en modo clúster por lo que no se ha llegado a configurar uno. Sin embargo, realizar la instalación de un clúster es igual de sencillo que instalarlo en un único nodo. El propio instalador guía al usuario para realizar la instalación nodo por nodo.

Basta con descargar el instalador en uno de los nodos del clúster y ejecutar el script '*install.sh*' con privilegios de administrador. También es necesario configurar el acceso SSH entre las máquinas del clúster, puesto que el propio instalador se encarga de conectarse a cada una de ellas.

## Redis

Redis es un sistema de almacenamiento en memoria que inicialmente fue diseñado para actuar como *caché* y que ha evolucionado hasta convertirse en un sistema complejo que, aparte de funcionar como *caché*', también funciona como base de datos o como gestor de mensajes.

No obstante, en este trabajo se ha utilizado Redis como sistema de almacenamiento exclusivamente, puesto que su sistema de almacenamiento en memoria permite un *throughput* muy alto.

Además, Redis utiliza un modelo físico de datos basado en clave-valor, por lo tanto, todas las consultas realizadas en Redis consisten en realizar un *hashing*, lo cual contribuye a su alto rendimiento, ya que no tiene que mantener más índices ni realizar búsquedas por otros campos.

Inicialmente Redis no fue diseñado para funcionar en modo clúster. Sin embargo, la necesidad de leer y procesar una gran cantidad de datos a velocidades muy altas ha provocado su evolución dando lugar a una arquitectura *master-to-master* para realizar el particionado de los datos y *master-slave* para garantizar disponibilidad.

Por lo tanto, si tenemos tres nodos *master*, cada uno de ellos se encargará de almacenar un subconjunto de la información y, en el momento en el que alguno de ellos no esté disponible, los datos que almacenan tampoco. Así que, por cada nodo *master* es recomendable que tenga al menos un nodo *slave* con el fin de tener tolerancia a fallos.

En este caso tampoco se ha realizado un despliegue en modo clúster y su instalación es bastante diferente a la de instalar un único nodo puesto que requiere activar el modo clúster, que funciona de una manera diferente.

## S-Store

S-Store es el primer y único motor de procesamiento en *streaming* capaz de garantizar un control transaccional a lo largo de todo el proceso. Además, no es únicamente un motor de procesamiento en *streaming* puesto que también posee capacidad para gestionar el almacenamiento. Por consiguiente, posee dos piezas clave de un *data pipeline* en uno, lo cual es una ventaja puesto que es un sistema menos a mantener.

S-Store está aún siendo desarrollado por parte de universidades estadounidenses como Brown University o el MIT junto con la ayuda de Intel. Puesto que está aún en desarrollo, S-Store no posee la capacidad de ser desplegado en forma de clúster puesto que aún no tiene implementado ningún tipo de *scheduler*.

El núcleo de S-Store es H-Store, la versión académica de VoltDB, por lo que su almacenamiento es NewSQL, es decir, sigue un esquema relacional, pero aprovechando la memoria como almacenamiento.

Con el fin de dar soporte a la capacidad de procesamiento en *streaming*, han introducido dos tipos de datos nuevos a H-Store a parte de las tablas: las ventanas y los *streams*. También han creado un nuevo tipo de *triggers* que son los encargados de desencadenar los procesos transaccionales.

De esta manera, S-Store gestiona los *streams* de datos utilizando procedimientos almacenados en conjunto con *triggers* para garantizar el control transaccional del proceso.

Esto se puede observar en la ilustración 9, en ella se explica cómo un conjunto de procedimientos almacenados encadenados son los encargados de realizar el procesamiento apoyándose de los *streams* y las ventanas.

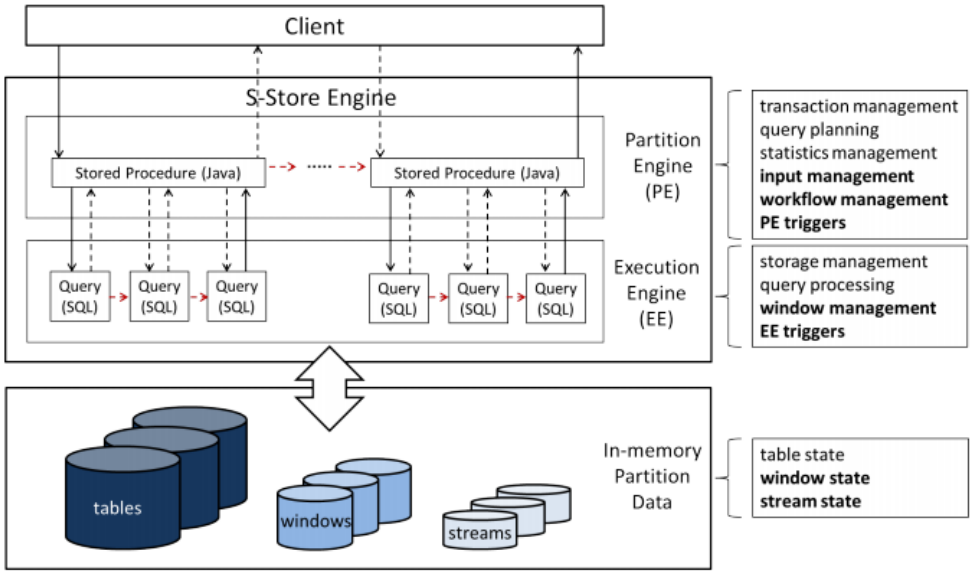


*Ilustración 9. Arquitectura de S-Store [5]*

## Estudio del comportamiento de las tecnologías

Aún no existe ningún *benchmark* estándar diseñado para evaluar el comportamiento de estos sistemas como puede ser el TPC para evaluar el comportamiento de sistemas transaccionales en diferentes casos de uso [20]. Esto ha derivado en dos vertientes: uno, que cada empresa tenga que diseñarse sus propios *benchmarks* para evaluar sus propios casos de uso debido a la falta de *benchmarks* como es el caso de Yahoo [4] o de Intel [21] y dos, que los propios creadores del sistema creen su propio *benchmark*, como es el caso de S-Store [5] o de Kafka [22].

Además, la gran mayoría de dichos *benchmarks* son sintéticos, como es el caso de los *benchmarks* mencionados en el párrafo anterior. Por lo tanto, muchos de ellos no reflejan cargas de trabajo de casos de uso reales por lo que puede que no obtengan los resultados que se obtendrían en una aplicación real.

Por lo tanto, es necesaria la creación y estandarización de una batería de *benchmarks* que evalúen el rendimiento de estos sistemas en diferentes casuísticas del mundo real.

Para ello en este trabajo inicialmente se eligieron dos *benchmarks* ya existentes y se realizó la implementación de ambos usando otras tecnologías con el fin de comparar su comportamiento. A continuación, se diseñó un *benchmark* basado en un caso de uso real del sector eléctrico. La tabla 2 resume los experimentos realizados indicando el caso de uso y las tecnologías en las que se implementaron:

| Benchmark | Gestor de colas | Procesamiento | Almacenamiento |
|---|---|---|---|
| ***Yahoo Streaming* [4]** | Kafka | Storm<br>Spark<br>S-Store | Redis<br>S-Store |
| ***Contest Voters* [5]** | - | Spark<br>S-Store | S-Store<br>MemSQL |
| **Curvas de consumo eléctrico** | Kafka | Storm<br>Spark | Cassandra |

*Tabla 2. Tabla de benchmarks a evaluar en este trabajo*

A continuación, se explicará la arquitectura utilizada para cada uno de ellos y los resultados de su ejecución. Por último, se analizarán los resultados obtenidos.

### *Yahoo Streaming Benchmark*

El caso de uso de Yahoo corresponde al análisis de *clicks* realizado en anuncios. De manera sintética y aleatoria se generan eventos de *clicks* a diferentes anuncios. Como se observa en la ilustración 10, una vez insertados los datos en Kafka, el motor de procesamiento se encarga de consumirlos y unirlos con su campaña publicitaria correspondiente filtrando los eventos que no pertenecen a ninguna campaña publicitaria. Después se realiza una agregación de los diferentes eventos de una campaña dentro de una ventana temporal para después almacenar esa información en la base de datos (Redis).

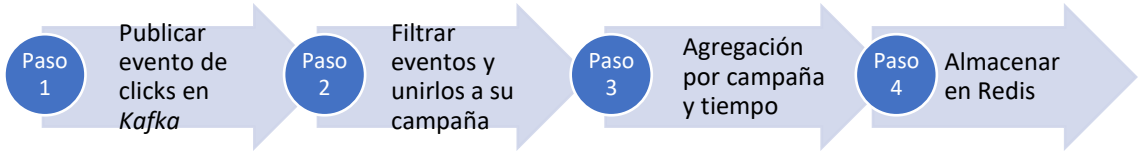


*Ilustración 10. Flujo de ejecución del Yahoo Streaming Benchmark*

En este caso, el *benchmark* estaba implementado para Flink, Spark y Storm por parte de Yahoo, así que se implementó en S-Store utilizándolo tanto como motor de procesamiento como de almacenamiento. Como las versiones utilizadas por el *benchmark* inicialmente eran bastante antiguas, fue necesario actualizar la API para utilizar una versión más reciente y adaptar la aplicación para que funcione con dicha API.

Más información relacionada con el *benchmark* de Yahoo se puede encontrar en su repositorio de GitHub [4].

## Arquitectura

S-Store actualmente solo puede ser ejecutado en un nodo ya que aún está en desarrollo y están trabajando en el *scheduler* encargado de planificar el reparto de las tareas entre los diferentes nodos de un clúster. Por lo tanto, este *benchmark* fue ejecutado en su totalidad en una única máquina. Quedando la arquitectura del sistema como en la ilustración 11.


*Ilustración 11. Arquitectura del sistema en el benchmark de Yahoo Streaming*

La máquina utilizada para realizar las pruebas disponía de 4 *cores* a 3.2GHz, 32GB de RAM y un HDD a 7200rpm. En cuanto a las versiones de cada uno de los sistemas, se han utilizado las siguientes:

- Spark Streaming v2.0.1
- Kafka v0.10.0.1
- Storm 1.0.4
- Redis 3.2.12

S-Store al estar aún en desarrollo no dispone de versionado, así que simplemente se ha utilizado la última versión publicada en su repositorio.

## Experimento y resultados

Actualmente S-Store no tiene implementado ningún sistema de *backpressure* para limitar la velocidad de los datos de entrada en el sistema. Por lo tanto, en esta prueba no ha sido habilitado en ninguna de las tecnologías. Sin embargo, sí que se ha detectado el punto de saturación en el que los sistemas dejan de ser estables y comienzan a perder datos.

Las pruebas realizadas en este *benchmark* fueron las que se relacionan en la Tabla 3. Para cada una de las pruebas se realizaron 10 ejecuciones durante 10 minutos y se calculó la media de los resultados obtenidos.

Las pruebas se realizaron inyectando datos en el sistema a diferentes velocidades para comprobar el comportamiento del sistema. Por lo tanto, no quiere decir que al terminar la ejecución del *benchmark* se haya obtenido dicho *throughput* de entrada, sino que se han insertado datos a esa velocidad. De hecho, solo se llega a obtener dicho *throughput* mientras que el sistema no llegue al punto de saturación, que es cuando pierde datos y, por lo tanto, su *throughput* no equivale a la velocidad de entrada de los datos.

Para entender los valores de las latencias, es importante tener en cuenta que la latencia es calculada a partir de la ventana temporal de la campaña publicitaria, que es de 10 segundos. Por lo tanto, todas las latencias máximas van a ser siempre mayores de dicho

| Colas | Procesamiento | Almacenamiento | Métricas |
|---|---|---|---|
| Kafka con un topic con 8 particiones. | Storm con un paralelismo de 1 para todos los *spouts* y *bolts* y 2 para el *spout* de guardado | Redis utilizando el esquema clave-valor | Latencias máxima, mínima y media dependiendo de la velocidad de entrada de los datos. |
| Kafka con un topic con 8 particiones. | Spark Streaming con una Ventana de 2s | Redis utilizando el esquema clave-valor | Latencias máxima, mínima y media dependiendo de la velocidad de entrada de los datos. |
| Kafka con un topic con 8 particiones. | S-Store | S-Store utilizando el esquema de Redis adaptado al relacional | Latencias máxima, mínima y media dependiendo de la velocidad de entrada de los datos. |

*Tabla 3. Pruebas a realizar en el benchmark de Yahoo Streaming*

valor ya que el primer dato que entre en la ventana no va a ser procesado hasta que la ventana finalice, así que como mínimo la latencia máxima siempre va a tener un valor de 10 segundos.

A continuación, se muestran las gráficas con los resultados de las ejecuciones de cada una de las pruebas.

Como se puede observar en las ilustraciones 12, 13 y 14, Spark obtiene mejores resultados que Storm y S-Store, puesto que es capaz de soportar un mayor *input* de datos que los otros dos sistemas.

La principal razón de esta diferencia es que Spark Streaming está optimizado para realizar cómputos en *batch*, por lo tanto, este caso de uso encaja a la perfección con su paradigma.

Por otro lado, Storm realiza el procesamiento a nivel de tupla, así que en casos de uso en los que el procesamiento es en *batch* puede salir perjudicado. A pesar de todo, obtiene un rendimiento considerablemente bueno y está a la par de S-Store.

Por último, S-Store, a pesar de tener la capacidad de realizar el procesamiento a nivel de *batch*, ha obtenido un rendimiento muy similar al de Storm y se ha quedado por detrás de Spark. Esto puede ser una contrapartida a la principal ventaja que tiene S-Store por encima de cualquier otro sistema similar y es que ofrece un control transaccional a nivel de proceso. Dicho control transaccional puede ser la razón por la cual se quede por detrás de Spark. Otra razón puede ser que Spark Streaming lleva años en el mercado y probablemente esté mejor optimizado puesto que hay muchas grandes empresas detrás mientras que S-Store está aún en desarrollo.

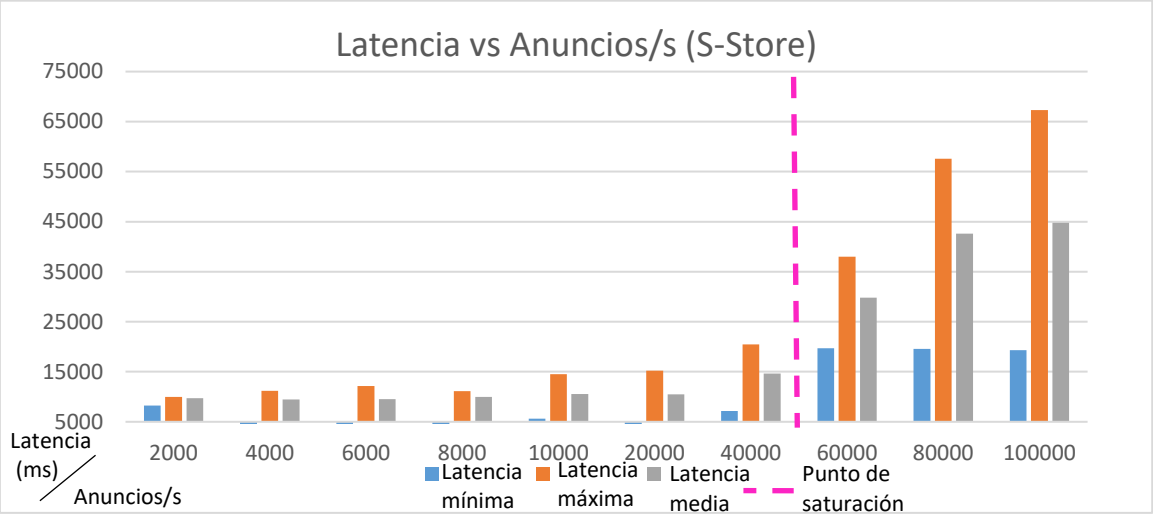


*Ilustración 12. Resultados de la ejecución del benchmark Yahoo Streaming en S-Store*

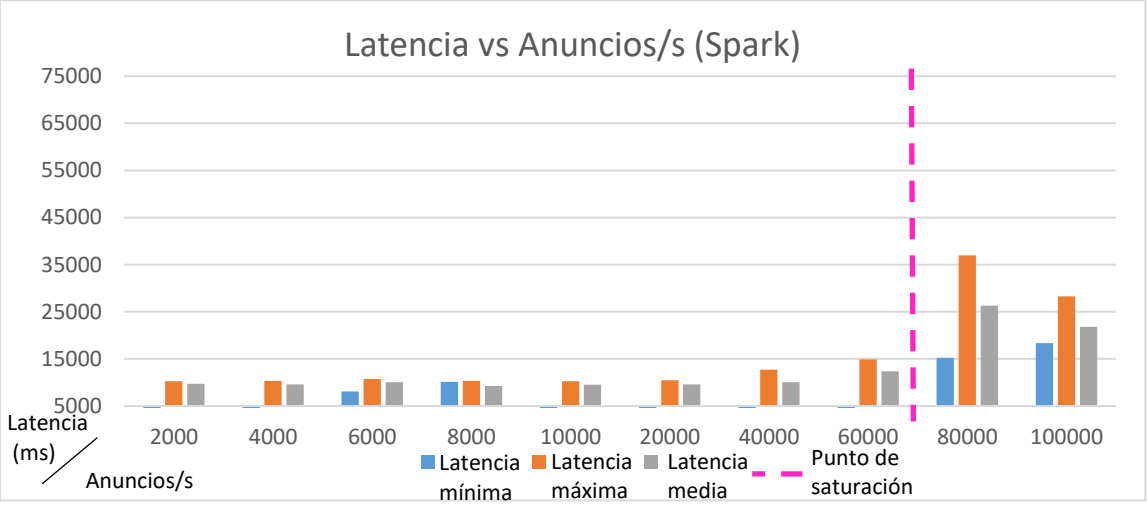


*Ilustración 13. Resultados de la ejecución del benchmark Yahoo Streaming en Spark*

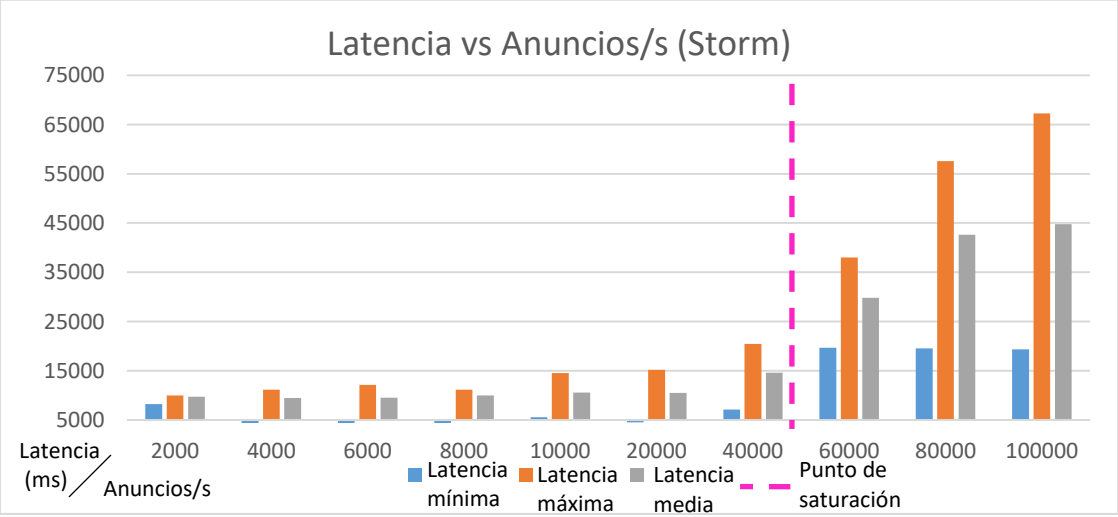


*Ilustración 14. Resultados de la ejecución del benchmark Yahoo Streaming en Storm*

### *Contest Voters Benchmark*

En este caso, el *benchmark* es el realizado por el equipo de S-Store, un *benchmark* que ya fue creado para la anterior H-Store y que también existe en VoltDB, su versión comercial.

El *benchmark* consiste en la votación de un ranking de un concurso de cantantes. En dicha votación un grupo de personas realiza votaciones mediante llamadas telefónicas. Cada vez que llega una votación se valida si dicha persona puede votar y si el voto es para un cantante válido. Si el voto cumple todos los requisitos, entonces se actualiza la tabla de clasificación de los cantantes y se comprueba el número de votos. Una vez se han realizado 100 votos, se elimina al cantante en la última posición y se solicita una nueva votación a las personas que hayan votado a dicho cantante. Por lo tanto, el flujo del proceso quedaría como en la ilustración 15.

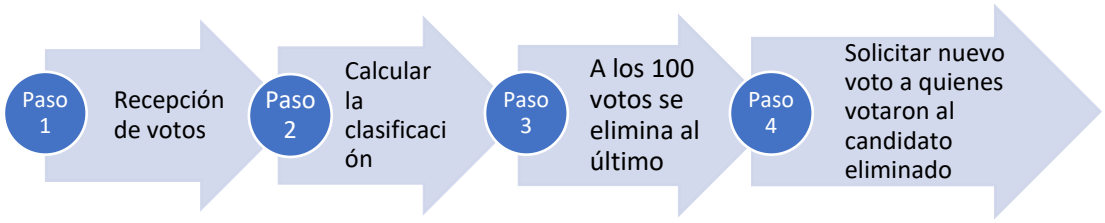


*Ilustración 15. Flujo del proceso del benchmark Contest Voters.*

Los fundadores de S-Store utilizan este *benchmark* para realizar una comparación entre realizar el proceso en S-Store y H-Store con el fin de demostrar la utilidad de las transacciones a nivel de proceso [5]. Sin embargo, en este trabajo se va a utilizar para evaluar el rendimiento de los sistemas.

El *benchmark* en sí genera automáticamente los votos de manera pseudoaleatoria y los inserta directamente en el proceso de S-Store. Así que, con el fin de aprovechar las ventajas de Kafka, se ha modificado el *benchmark* para que los votos se generen fuera de S-Store y se publiquen en Kafka, al cual se subscribirá S-Store para insertar los votos en el sistema. También se ha implementado el mismo proceso con Spark Streaming, puesto que su naturaleza en *batch* permite realizar un control transaccional similar al de S-Store dentro de cada *batch* y utilizando MemSQL como almacenamiento puesto que también es un motor de base de datos NewSQL, es decir, utiliza el esquema relacional, pero con la particularidad de poder escalar horizontalmente y almacenar la información en memoria (similar al que utiliza S-Store).

#### Arquitectura

Al igual que ocurre en el anterior *benchmark*, S-Store únicamente puede ser ejecutado en un único nodo. Por lo tanto, se ha ejecutado el *benchmark* en su totalidad en una única máquina. Quedando la arquitectura del sistema como en la ilustración 16.


*Ilustración 16. Arquitectura del sistema en el benchmark de Contest Voters*

La máquina utilizada para realizar las pruebas disponía de 4 *cores* a 3.2GHz, 32GB de RAM y un HDD a 7200rpm. En cuanto a las versiones de cada uno de los sistemas, se han utilizado las siguientes:

- Spark Streaming v2.0.1
- Kafka v0.10.0.1
- Redis 3.2.12

S-Store al estar aún en desarrollo no dispone de versionado, así que simplemente se ha utilizado la última versión publicada en su repositorio.

### Experimento y resultados

En este *benchmark* el objetivo es comparar el rendimiento en un caso de uso en el que el orden de llegada de los datos importa y que, por lo tanto, el control transaccional juega un papel muy importante.

Para ello se va a comparar el *throughput* de S-Store en función de su principal parámetro de configuración, que es el máximo número de transacciones por segundo ('*txnrate*') que permite, con el *throughput* de Spark Streaming en función de su tamaño de ventana, la cual juega un papel muy importante en este caso de uso puesto que Spark solo puede garantizar el orden dentro un mismo *batch*.

| Colas | Procesamiento | Almacenamiento | Parámetros | Métricas |
|---|---|---|---|---|
| Kafka | S-Store | S-Store | Transacciones por segundo | *Throughput* |
| Kafka | Spark Streaming | MemSQL | Tamaño de ventana | *Throughput* |

*Tabla 4. Tabla de pruebas a realizar en el benchmark de Contest Voters*

Al igual que en el *benchmark* anterior, por cada una de las posibles configuraciones se han realizado 10 ejecuciones durante 10 minutos en las cuales se han insertado los votos generados de manera pseudoaleatoria en Kafka.

En este caso, las gráficas simplemente representan el *throughput* real obtenido por cada uno de los sistemas con el fin de comparar su capacidad.

Como se puede observar en las ilustraciones 17 y 18, la diferencia de rendimiento entre S-Store y Spark es sorprendentemente enorme. Al contrario que en el *benchmark* anterior, donde Spark obtenía un rendimiento considerablemente superior a S-Store, en este caso es todo lo contrario. Y es que hay una gran diferencia de 7500 votos por segundo que es capaz de procesar S-Store a 800 votos por segundo que es capaz de procesar Spark.

Esto demuestra lo comentado al inicio del capítulo de la necesidad de crear una batería de *benchmarks* para diferentes aplicaciones. Cada sistema está diseñado para satisfacer ciertos casos de uso y, por lo tanto, es lógico que cada sistema del mejor rendimiento en sus propios *benchmarks*, puesto que realizan un *benchmark* para evaluar el sistema en el caso de uso para el que el sistema fue creado.

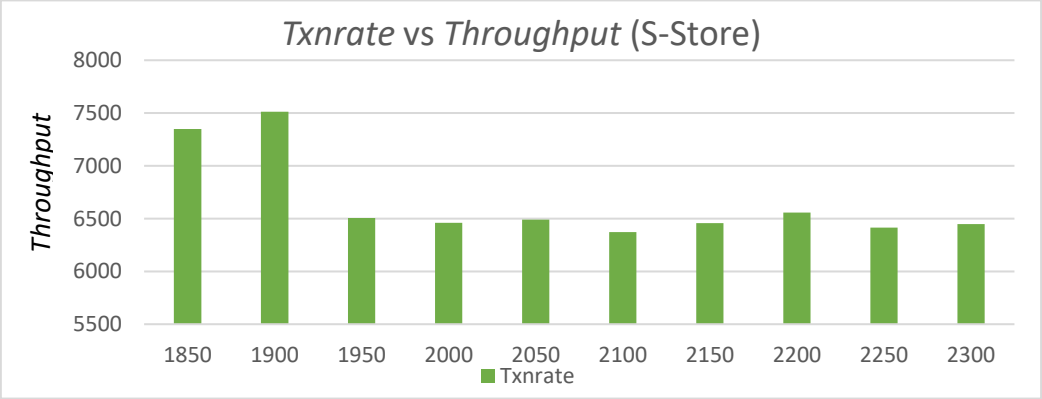


*Ilustración 17. Resultados de la ejecución del benchmark Contest Voters en S-Store*

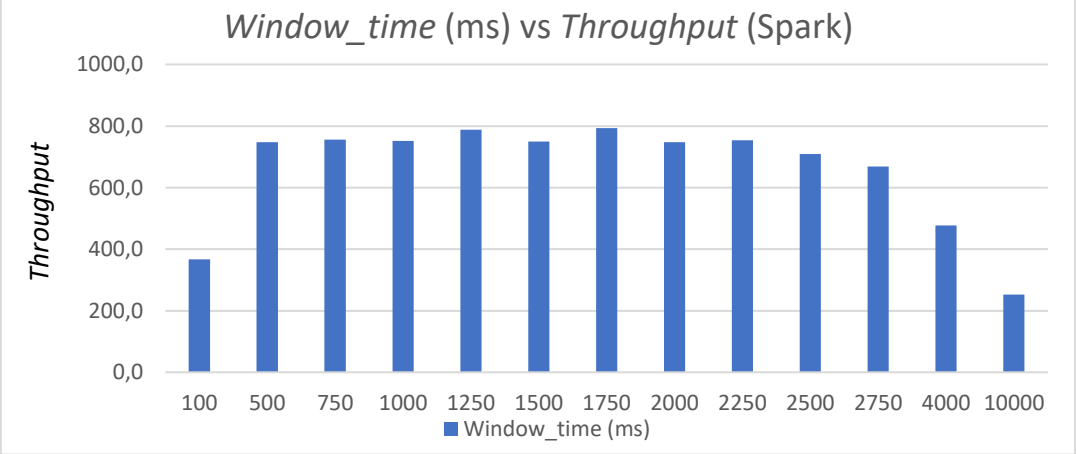


*Ilustración 18. Resultados de la ejecución del benchmark Contest Voters en Spark*

Otra razón de la diferencia de rendimiento puede deberse a los controles que han sido necesarios realizar a nivel de código en Spark para acercarse lo máximo posible al comportamiento transaccional de S-Store. S-Store sin embargo tiene esa capacidad de forma nativa y por lo tanto destaca en este aspecto.

### Diseño y creación de un *benchmark para la lectura diaria de contadores eléctricos*

La evolución de los contadores eléctricos de analógicos a digitales ha ocasionado una considerable mejora de los servicios que proporcionan las compañías eléctricas. Los nuevos contadores permiten una medición de lectura automática, es decir, tanto la recolección de los datos como su envío se hacen de manera remota. De esta modo no es necesario que el personal de la compañía se desplace hasta los contadores de cada uno de sus clientes.

Sin embargo, esta evolución requiere de una infraestructura capaz de recoger y procesar las lecturas horarias de los contadores de cada uno de los clientes. Por lo tanto, si la compañía eléctrica tiene una gran cantidad de clientes, será necesario utilizar tecnologías capaces de tratar una cantidad ingente de datos con requisitos *soft real time*, lo que hace necesario utilizar tecnologías orientadas al *streaming* implementando un pipeline.

Viesgo actualmente tiene alrededor de 700.000 CUPS (clientes), por lo tanto, recibe 700.000 lecturas cada hora. Sin embargo, tal cual está su infraestructura actual, las lecturas se realizan de manera horaria pero no se procesan hasta tener la lectura completa de un día entero. Por lo tanto, reciben alrededor de 700.000 lecturas al día, donde cada una registra las lecturas de cada hora del día.

Por otro lado, la infraestructura actual no tiene una fiabilidad del 100% puesto que los contadores pueden fallar o dar lecturas incorrectas, así que se realiza un recalculo de las lecturas incorrectas en función del consumo histórico del cliente y de un perfil de consumo que se le ha asignado con anterioridad siguiendo un algoritmo establecido por la Red Eléctrica de España.

A pesar de ello, estos cálculos no reflejan el consumo real del cliente, sino una estimación, por ello, se realiza una lectura real mensual por cliente para obtener el consumo real de cada CUP. Con esta lectura real, se reajustan las curvas previamente ya recalculadas para aquellos valores que llegaron erróneos utilizando el valor real. Finalmente, se calcula el consumo mensual del cliente a partir de las curvas con mayor nivel de precisión.

De esta manera, los datos generados por la infraestructura actual tienen el siguiente caudal:

- 700.000 lecturas (curvas horarias) / día
- 700.000 lecturas reales / mes

## Casos de uso

Esta problemática da luz a dos casos de uso diferentes:

- El procesamiento de las curvas horarias

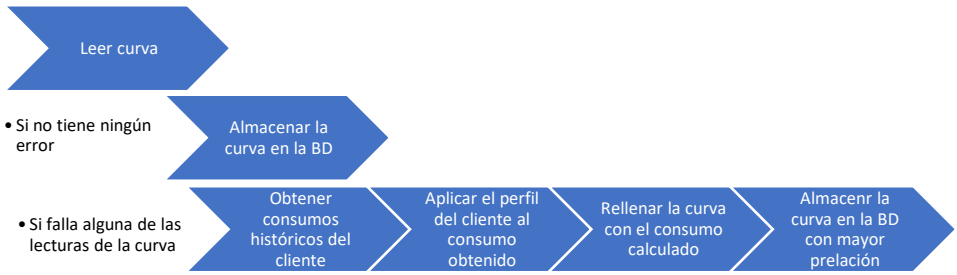


*Ilustración 19. Flujo de procesamiento de las curvas horarias*

- El procesamiento de las curvas reales

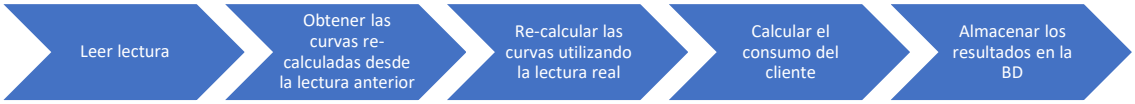


*Ilustración 20. Flujo de procesamiento de las curvas reales*

## Arquitectura

Para dar respuesta a esta problemática se ha optado por implementar un *pipeline* con un flujo diferente para cada uno de los procesos mencionados, de manera que la arquitectura del sistema quedaría como recoge la ilustración 21.

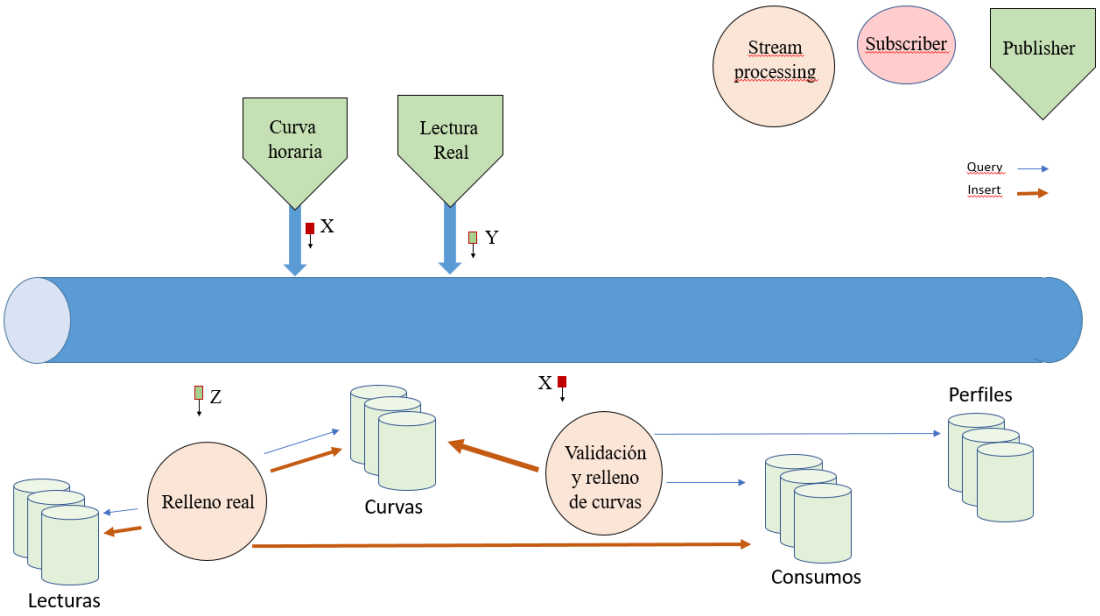


*Ilustración 21. Arquitectura del pipeline para el benchmark del sector eléctrico*

Como se puede observar en la ilustración 21, se han creado dos *topics*, uno por cada flujo de datos. Por simplicidad ambos tendrán la misma configuración y su factor de replicación dependerá del número de nodos de Apache Kafka desplegados. Este número es una variable que utiliza el *benchmark*, por lo tanto, el factor de replicación de los *topics* dependerá de esta variable.

Por otro lado, existen dos procesos de *streaming*, el de relleno de curvas y el del uso de lecturas reales para recalcular dichas curvas. Ambos procesos serán ejecutados en todos los nodos del motor de procesamiento utilizado cuya cantidad es una variable del *benchmark*. Por lo tanto, la capacidad de paralelización de los procesos depende de la configuración concreta en cada prueba del *benchmark*.

Por último, en la Ilustración 21 se puede observar que existen 4 almacenes de datos diferentes. Sin embargo, pueden ser almacenados todos en una única base de datos (en el gráfico están separados para facilitar la visualización del movimiento de los datos. Al no existir ninguna relación entre cada uno de los datos, el esquema resultará en 4 tablas independientes como se puede observar en la ilustración 22.

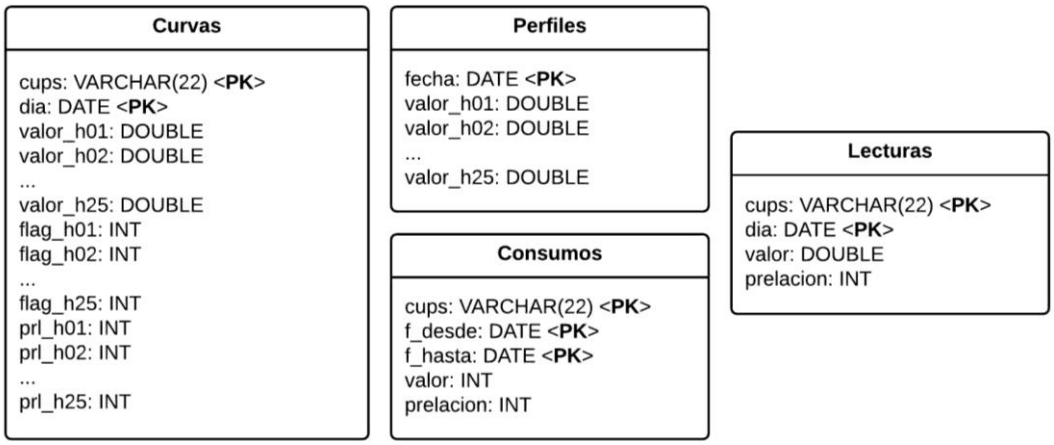


*Ilustración 22. Esquema de la base de datos del benchmark del sector eléctrico*

Sin embargo, con el fin de facilitar la manipulación la velocidad a la que se inyectan datos en el sistema con facilidad, en el *benchmark* solo se va a utilizar el caso de uso del procesamiento de las curvas horarias, puesto que si no sería necesario sincronizar ambos *inputs* para poder imitar a la realidad y eso podría llegar a limitar la velocidad de entrada de los datos en el sistema.

En este caso se ha prescindido de S-Store y, por lo tanto, se va a jugar con la escalabilidad de los nodos de todos los componentes del sistema. Así que la arquitectura cambia considerablemente respecto a los *benchmark* anteriores. En este caso se va a evaluar las diferencias entre Spark y Storm, utilizando Kafka como gestor de colas y Cassandra como gestor de almacenamiento.

En este *benchmark*, se han realizado pruebas en dos entornos diferentes debido a problemas que surgieron utilizando el primer entorno y a los cuales no llegué a encontrar ninguna solución.

Inicialmente se utilizaron 16 nodos reservados del clúster 3Mares de la Universidad de Cantabria como muestra la ilustración 23. Cada nodo utilizaba un Xeon con 8 *cores* y entre 10 y 16 GB de RAM. Así que se decidió dedicar 5 nodos a cada componente y un nodo para ocuparse de la inyección de los datos en el sistema. Hay que destacar que tanto en el clúster de Kafka como en el de Storm hubo que instalar Zookeeper en tres de los nodos de su clúster ya que dependen de él para su funcionamiento.

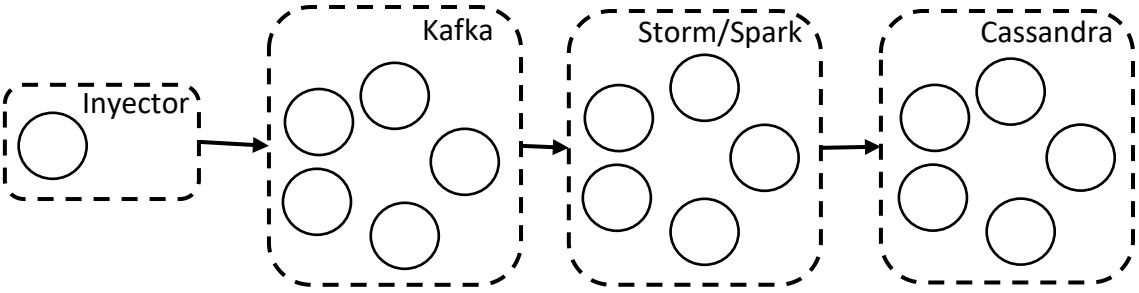


*Ilustración 23. Arquitectura inicial del benchmark del sector eléctrico*

Previo a la reserva de los nodos se decidió utilizar el sistema de colas implementado en el clúster. Así pues, tuve que desarrollar *scripts* de arranque que se adaptaran al sistema de colas. A pesar de ello, me encontré con varios problemas respecto a la utilización de sistemas Java mediante el sistema de colas. Por lo que, tras contactar con el

administrador del sistema, descubrimos que existía un *bug* en el sistema de colas cuando se utiliza la máquina virtual de Java. Como solución rápida me ofreció reservar ciertos nodos en el sistema y conectarme a ellos para realizar las pruebas mediante SSH, estos son los 16 nodos descritos anteriormente.

Una vez dispuse de las máquinas para hacer las pruebas procedí a instalar todos los sistemas para asegurarme de que funcionaban correctamente. Y otra vez me encontré con problemas, esta vez de permisos. Los nodos disponían de dos sistemas de ficheros, uno local que era temporal y se borraba periódicamente y uno distribuido donde se encontraban las carpetas de usuario.

Para subsanarlo, intenté instalar todo en la carpeta local, haciendo una estructura de carpetas para cada nodo puesto que todos leían del mismo sistema de ficheros. No obstante, también me encontré con errores, al intentar instalar cualquier base de datos (hice pruebas con MemSQL, Cassandra e Ignite, con la esperanza de que alguna funcionara) me daban todas ellas un error de permisos, el cual se arregló moviendo los ficheros de instalación al sistema de ficheros local y temporal de los nodos.

A partir de ahí conseguí hacer funcionar los sistemas y empecé a ejecutar pruebas. Sorprendentemente todas las pruebas me daban resultados muy similares, independientemente de los nodos y parámetros de configuración que utilizara. Así que intenté monitorizar todos los sistemas y la utilización de los recursos. Todos los recursos tenían un porcentaje de uso, tanto de CPU como de memoria, muy bajos y los sistemas estaban comportándose como lo esperado. Tras probar modificando una gran cantidad de parámetros en cada uno de los sistemas fui incapaz de obtener un rendimiento diferente, por lo que llegué a la conclusión de que podía tener algún tipo de restricción a nivel de usuario en el SO ya que no disponía del control total del sistema.

Como no fui capaz de hacer funcionar el sistema en ese entorno, decidí montar un sistema similar en la nube. Por lo que se alquilaron 16 nodos en la nube los cuales tenían 4 *cores* y 8GB de RAM, la mitad que el sistema anterior. El despliegue del sistema se realizó al igual que en el otro entorno, como se muestra en la ilustración 23. En este caso no tuve ningún inconveniente a la hora de realizar el despliegue y sorprendentemente obtuve mejores resultados que en el otro clúster teniendo menos recursos, además en este entorno los parámetros sí afectaban al rendimiento y los recursos estaban siendo utilizados al 100% durante las ejecuciones.

### Experimento y resultados

El objetivo de este experimento era obtener la mejor configuración posible para este caso de uso, tanto como Storm como con Spark. el objeto era por tanto evaluar la capacidad de escalado horizontal del sistema y el impacto de los parámetros de configuración más relevantes para cada uno de los sistemas.

Las ejecuciones del experimento consisten en la inserción de datos a Kafka a una velocidad más rápida de la que puede alcanzar el sistema, de esta manera actúan los sistemas de *backpressure* y podemos obtener los rendimientos más altos posibles.

Inicialmente se va a probar la escalabilidad de cada uno de los sistemas, partiendo de clúster de 2 nodos para cada uno de ellos y añadiendo progresivamente nodos a los clústeres. Así que primero se realizó el escalado del procesamiento puesto que era el núcleo del sistema y lo que probablemente nos aumentara en mayor medida el

rendimiento. Una vez escalado el procesamiento se realizó el escalado en Kafka y después en Cassandra.

Las métricas a tener en cuenta son el *throughput* total del sistema y las latencias mínimas, máximas y medias desde que el dato es consumido por el motor de procesamiento hasta que queda consolidado en el sistema de almacenamiento.

Una vez evaluada la escalabilidad, utilizando los sistemas con la mejor configuración de nodos, se evaluó el impacto de los siguientes parámetros de configuración:

- Las particiones en Kafka
- El tamaño de la ventana en Spark
- El paralelismo de los elementos en Storm
- Dos tipos de consultas en Cassandra

Las pruebas quedan resumidas en la Tabla 5:

| Colas | Procesamiento | Almacenamiento | Parámetros | Métricas |
|---|---|---|---|---|
| Kafka | Spark | Cassandra | Número de nodos de los sistemas, tamaño de ventana en Spark, particiones de Kafka y consultas de Cassandra. | *Throughput*, latencia mínima, latencia media y latencia máxima. |
| Kafka | Storm | Cassandra | Número de nodos de los sistemas, paralelismo en Storm, particiones de Kafka y consultas de Cassandra. | *Throughput*, latencia mínima, latencia media y latencia máxima. |

*Tabla 5. Pruebas a ejecutar en el benchmark del sector eléctrico*

Como en los casos anteriores, para cada una de las pruebas se han realizado 10 ejecuciones durante 10 minutos y se ha calculado la media de los resultados de dichas ejecuciones. La única diferencia respecto a los anteriores *benchmarks* es que los datos utilizados en este experimento son reales y no sintéticos.

A continuación, se incluyen las gráficas con los resultados obtenidos. En las primeras gráficas, las ilustraciones 24 y 26, se comparan los *throughputs* entre ambas arquitecturas usadas para comprobar la diferencia del escalado entre una y otra con el fin de ver reflejado en los datos por qué fue necesario el cambio de arquitectura.

Como se puede observar en la ilustración 24, aparte de conseguir un rendimiento pésimo, en el clúster 3Mares, independientemente del número de nodos utilizados el rendimiento es el mismo mientras que en el *cloud*, a medida que se aumenta el número de nodos, aumenta el *throughput*. A pesar de aumentar el rendimiento aumentando el número de nodos, la latencia se mantiene más o menos estable, por lo que a priori aumentar el número de nodos no conlleva ningún *trade-off*.

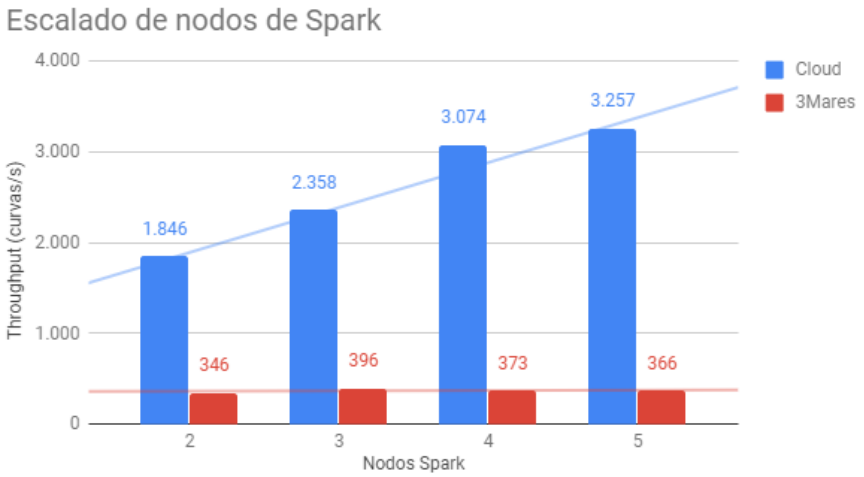


*Ilustración 24. Throughput obtenido en el benchmark del sector eléctrico dependiendo del número de nodos de Spark*

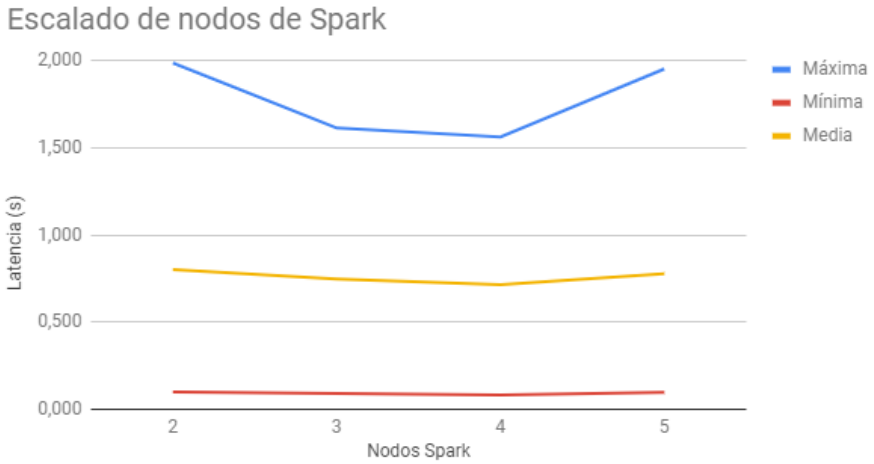


*Ilustración 25. Latencia obtenida en el benchmark del sector eléctrico dependiendo del número de nodos de Spark*

Al igual que en Spark, Storm se beneficia del número de nodos que pertenecen al clúster, aunque su rendimiento no escale en exceso, lo cual puede ser debido a que haya algún otro cuello de botella en el sistema. Por otro lado, al igual que en Spark, en el 3Mares el rendimiento se mantiene plano independientemente del número de nodos utilizados.

Sorprendentemente, al aumentar el número de nodos, las latencias tanto máxima como media se reducen. Aunque el cambio no es significativo, sí que es sorprendente puesto que hay más nodos que tienen que coordinarse. A simple vista todo son ventajas al añadir nodos a un clúster de Storm.

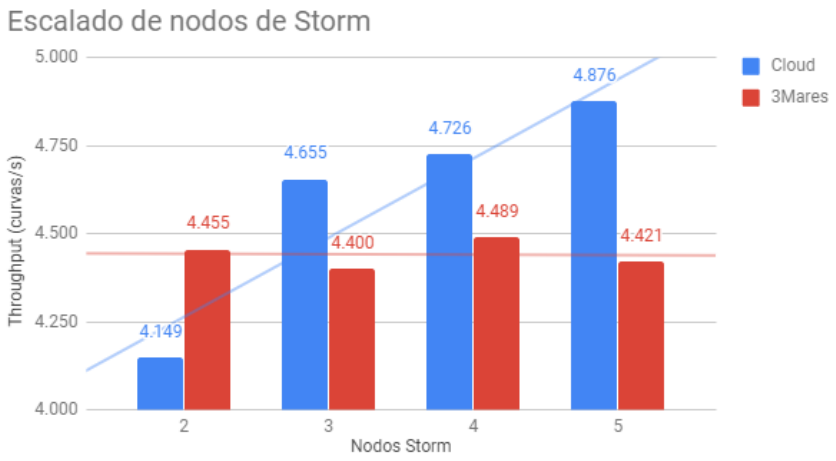


*Ilustración 26. Throughput obtenido en el benchmark del sector eléctrico dependiendo del número de nodos de Storm*

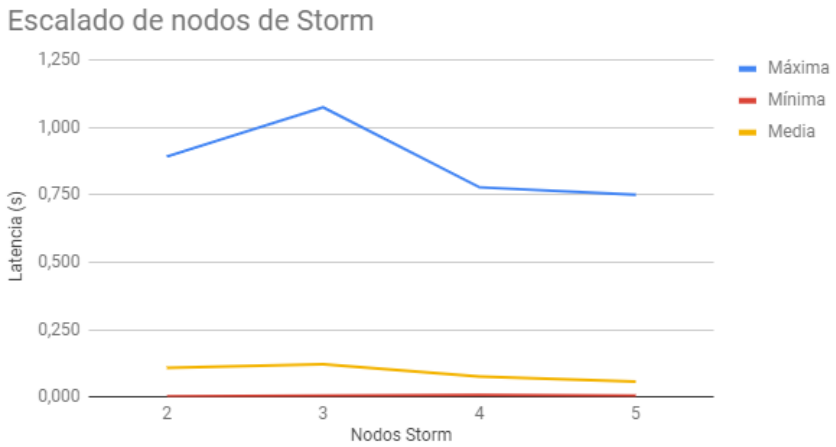


*Ilustración 27. Latencia obtenida en el benchmark del sector eléctrico dependiendo del número de nodos de Storm*

A continuación, se incrementará el número de nodos de Kafka utilizando la mejor configuración de las pruebas anteriores, es decir, utilizando 5 nodos.

Aumentar el número de nodos en Kafka aparentemente no conlleva ningún impacto en el sistema considerable a la vista de las gráficas que se muestran en las ilustraciones 28 y 29. Aunque sí que es cierto que el rendimiento se ve un poco reducido al utilizar 5 nodos con Storm, los valores obtenidos están dentro de unos rangos similares. Estos resultados no son para nada sorprendentes puesto que Kafka puede manejar cientos de miles de lecturas por segundo en cada nodo, por lo que, para estos *throughputs*, el número de nodos no va a afectar al rendimiento del sistema.

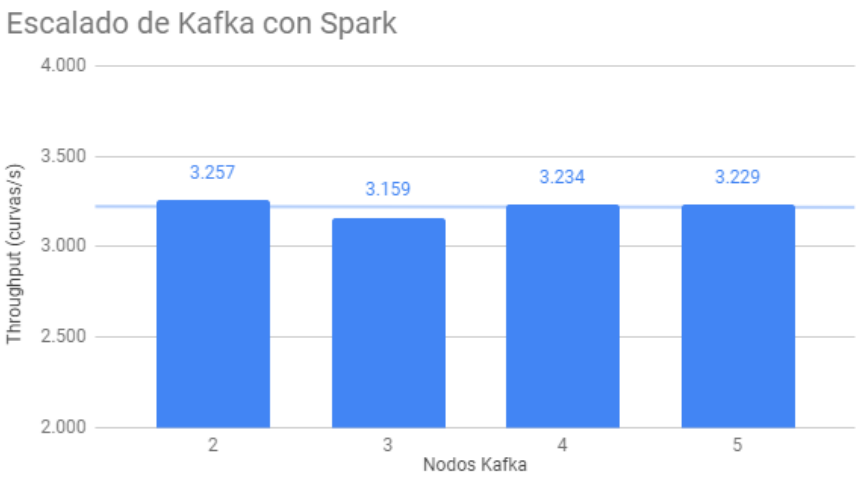


*Ilustración 28. Throughput obtenido en el benchmark del sector eléctrico dependiendo del número de nodos de Kafka con Spark*

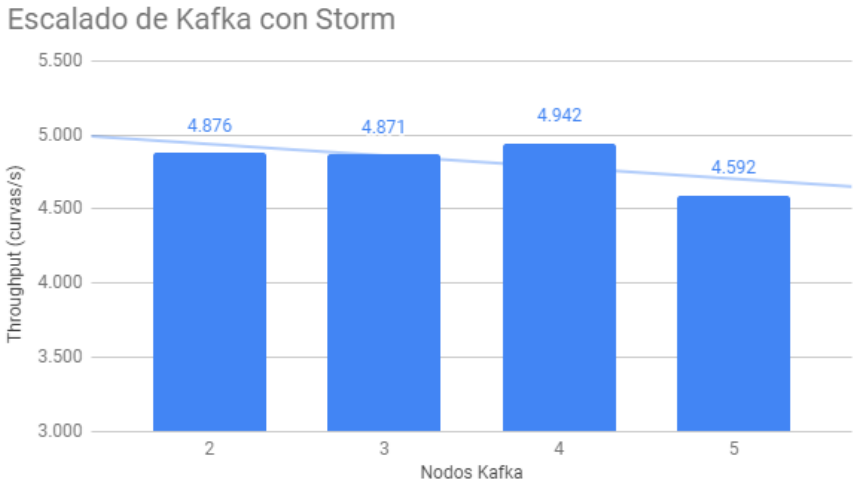


*Ilustración 29. Throughput obtenido en el benchmark del sector eléctrico dependiendo del número de nodos de Kafka con Storm*

A continuación, se ha realizado el correspondiente escalado con Cassandra, utilizando 5 nodos en los otros dos sistemas.

Como era de esperar, aumentar los nodos de Cassandra aumenta el rendimiento tanto con Storm como con Spark como se puede observar en las ilustraciones 30, 31, 32 y 33 respectivamente. Esto se debe principalmente a dos razones: uno, la facilidad de escalado horizontal que tiene Cassandra gracias a su particionado y su modelo *master-to-master* y, dos, debido a que la aplicación tiene una gran cantidad de operaciones contra la base de datos, por cada dato que llega, realiza 2 lecturas y una inserción, así que aumentar el número de nodos reparte la carga entre ellos.

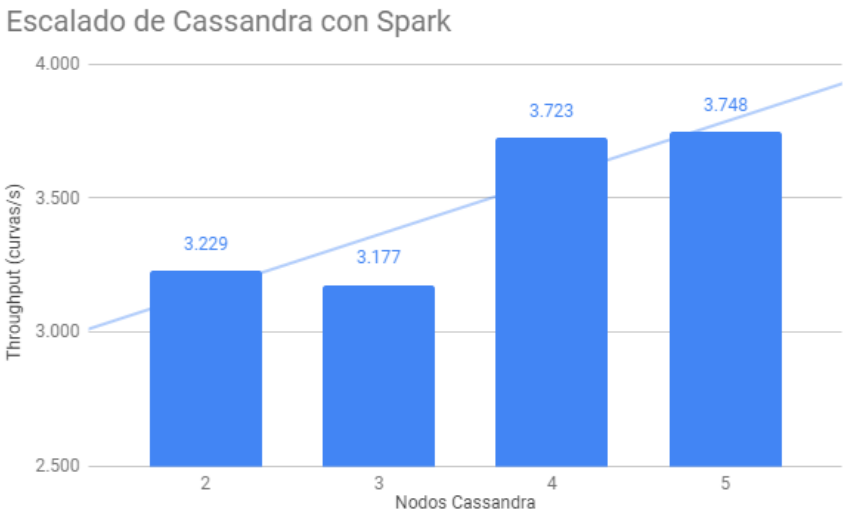


*Ilustración 30. Throughput obtenido en el benchmark del sector eléctrico dependiendo del número de nodos de Cassandra con Spark*

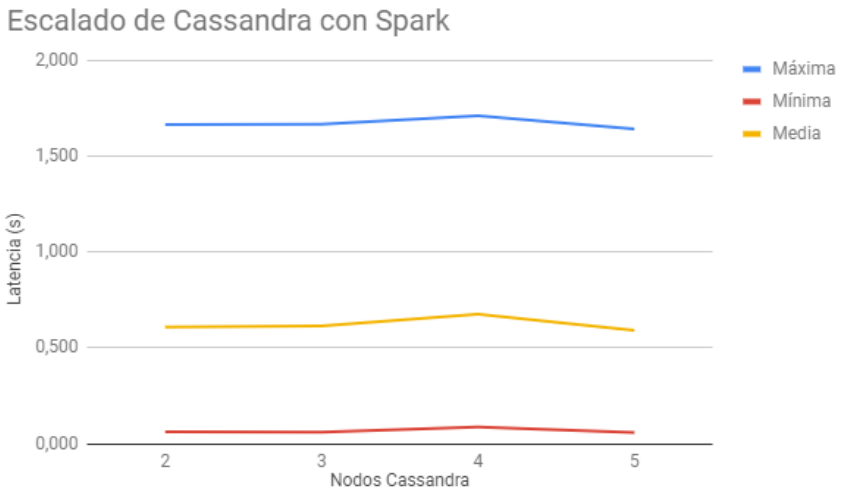


*Ilustración 31.Latencia obtenida en el benchmark del sector eléctrico dependiendo del número de nodos de Cassandra con Spark*

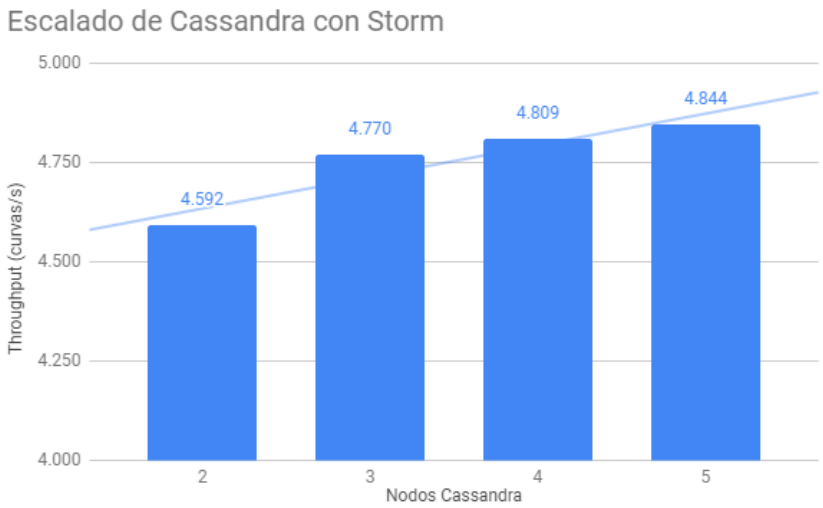


*Ilustración 32. Throughput obtenido en el benchmark del sector eléctrico dependiendo del número de nodos de Cassandra con Storm*

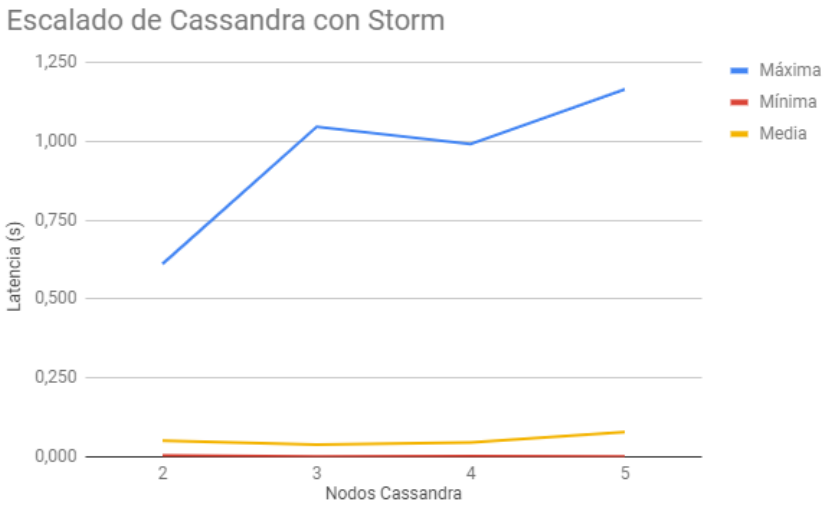


*Ilustración 33. Latencia obtenida en el benchmark del sector eléctrico dependiendo del número de nodos de Cassandra con Storm*

Una vez evaluado el escalado de los sistemas donde la conclusión es que a mayor número de nodos mejor rendimiento, exceptuando a Kafka debido a su gran rendimiento. Procedemos a evaluar los parámetros de los diferentes sistemas.

Primero el número de particiones de Kafka. Una partición Kafka, como se ha mencionado anteriormente, es la unidad de paralelismo, así que a mayor número de particiones en principio debería haber un mejor rendimiento. Puesto que si solo existe una partición la lectura de los datos solo podrá realizarla un *thread*.

Como podemos observar en las ilustraciones 34 y 35, Spark y Storm tienen comportamientos completamente opuestos. Por un lado, El rendimiento de Spark aumenta con el número de particiones como lo esperado, aunque empieza a decaer a partir de las 64 particiones. La razón es la necesidad de gestionar cada una de las particiones por parte de Spark, al aumentar drásticamente el número y no disponer de más recursos, Spark no puede aprovechar dicho paralelismo. Si aumentáramos el número de nodos del clúster de Spark lo más probable es que el rendimiento con 64 particiones fuera mayor que con 32. Por otro lado, Storm obtiene un rendimiento considerablemente superior con una única partición. Este comportamiento es probablemente el que más me ha sorprendido hasta ahora, parece ser que Storm no trabaja bien con el número de particiones, independientemente de si tiene más o menos *threads* asignados para consumirlas. Investigando en la documentación de Storm y los foros de la comunidad no he encontrado una explicación clara a este comportamiento, pero sí a más personas que les ocurre exactamente lo mismo.

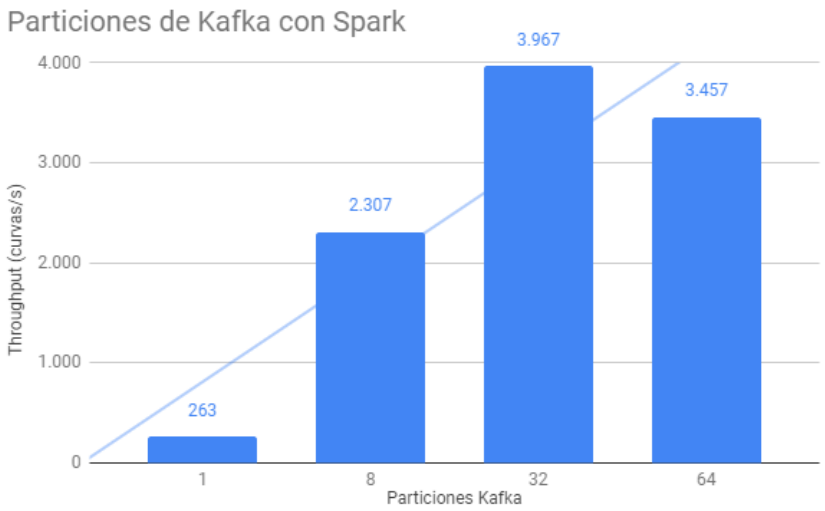


*Ilustración 34. Throughput obtenido en el benchmark del sector eléctrico dependiendo del número de particiones en Kafka con Spark*

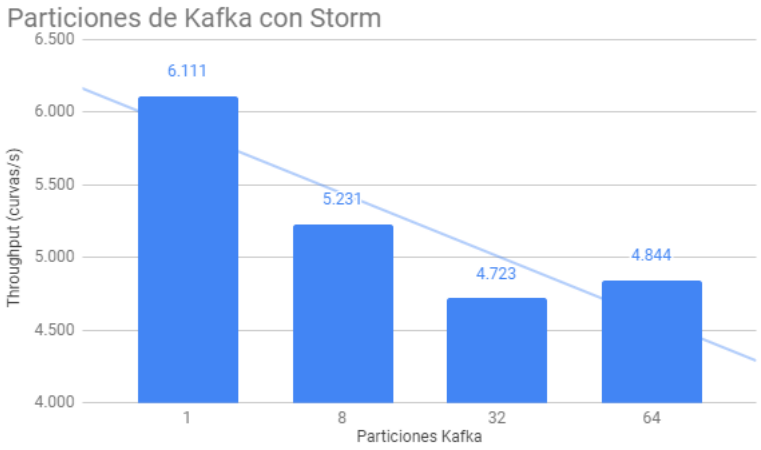


*Ilustración 35. Throughput obtenido en el benchmark del sector eléctrico dependiendo del número de particiones en Kafka con Storm*

Tras este curioso e inesperado dato, pasamos a evaluar los parámetros tanto de Storm como de Spark. En cada uno de ellos, con el fin de obtener los mejores rendimientos se ha utilizado su mejor configuración en Kafka, es decir, una partición para Storm y 32 para Spark.

El impacto del tamaño de la ventana da los resultados esperados. El procesamiento en este caso de uso es a nivel de tupla, por lo que, a mayor tamaño de ventana, mayor número de tuplas se tendrán que procesar al final del *batch* que genera la ventana. Esto podría ser una ventaja en caso de que el cómputo fuera en *batch*, pero al no ser así, se realizan muchas peticiones a la base de datos simultáneamente, lo cual impacta en el rendimiento.

Sin embargo, tener una ventana demasiado pequeña también puede ser contraproducente puesto que el hecho de crear y gestionar conlleva un *overhead* por lo que, si la ventana es demasiado pequeña, el *overhead* es demasiado grande en comparación con el cómputo.

Respecto a las latencias, están relacionadas directamente con el tamaño de la ventana puesto que hasta que no se termina una ventana un dato no llega a la base de datos y, por tanto, eso aumenta la latencia.

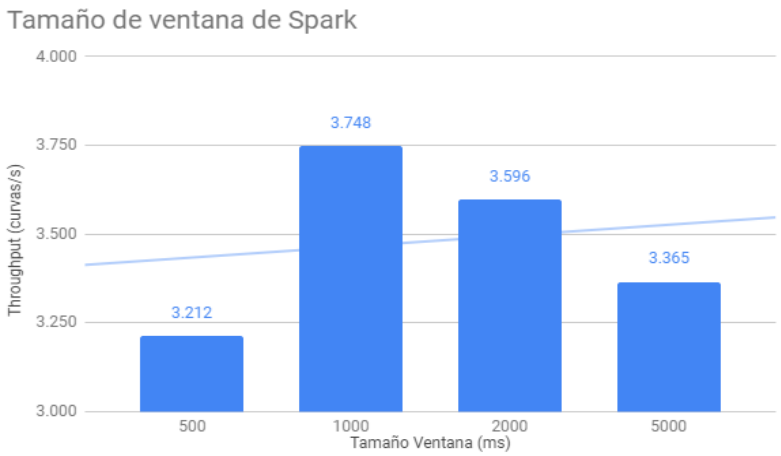


*Ilustración 36. Throughput obtenido en el benchmark del sector eléctrico dependiendo del tamaño de la ventana de Spark*

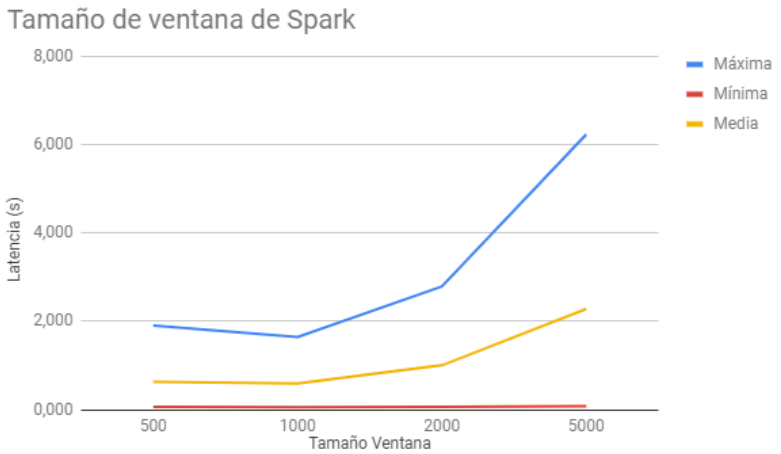


*Ilustración 37. Latencia obtenida en el benchmark del sector eléctrico dependiendo del tamaño de la ventana de Spark*

Para entender el paralelismo las ilustraciones 38 y 39, es necesario entender cómo está diseñado el *benchmark* en Storm. La aplicación de Storm consta de 5 elementos (uno por cada valor del eje X en la gráfica). Los elementos por orden son los siguientes:

- Un *spout* que consume de Kafka.
- Un *bolt* que parsea el dato.
- Un *bolt* que consulta a la BD los datos necesarios para el cómputo.
- Un *bolt* que procesa los datos.
- Un *bolt* que guarda el resultado en Cassandra.

Por lo tanto, cada número del eje X equivale a uno de los elementos de la aplicación, es decir, el dato 8/8/8/8/8, tendría 8 *threads* por cada uno de los elementos.

Aunque inicialmente se han asignado el mismo número de *threads* a todos los elementos, para el resto se han seguido los siguientes criterios: uno, la partición de Kafka

es 1, por lo tanto, más de un *thread* en el *spout* significa tener *threads* ociosos y dos, el cuello de botella está en las conexiones a Cassandra, por lo que la gran mayoría de los recursos han ido dedicado a esos procesos.

De hecho, como se puede observar en las gráficas, tanto la mejor latencia como el mejor *throughput* se dan cuando completamente todos los *threads* están asignados a los procesos de Cassandra y un único *thread* en el resto de los procesos.

La diferencia que se puede obtener ajustando este parámetro es considerable y puede ayudar a reducir cuellos de botella. Es un parámetro muy importante a tener en cuenta.

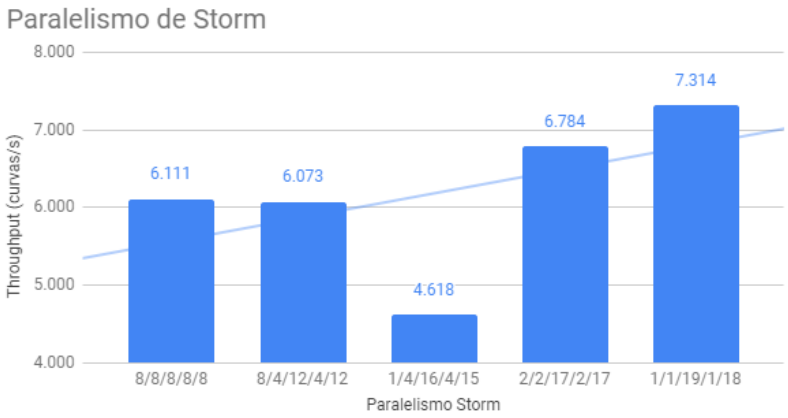


*Ilustración 38. Throughput obtenido en el benchmark del sector eléctrico dependiendo del paralelismo en Storm*

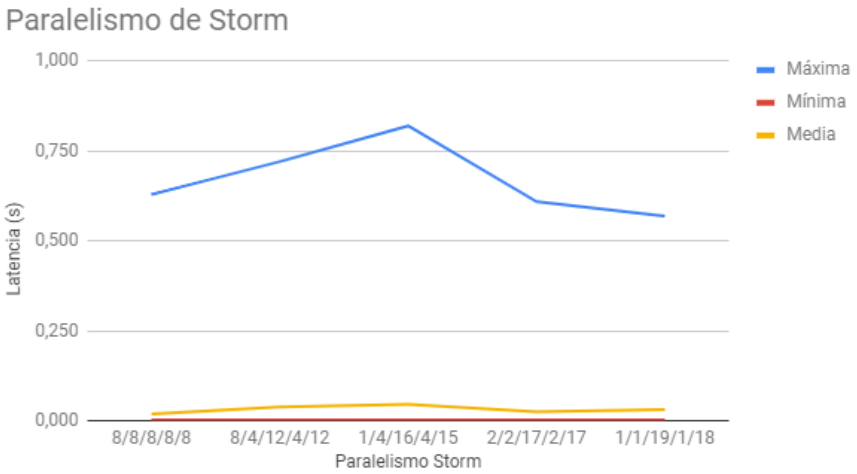


*Ilustración 39. Latencia obtenida en el benchmark del sector eléctrico dependiendo del paralelismo en Storm*

Por último, debido a que en el clúster de 3Mares el rendimiento estaba limitado, se intentó buscar cuellos de botella en el propio sistema intentando optimizar los procesos. A raíz de ello, surgieron dos tipos de consultas en Cassandra, uno en el que la búsqueda por rangos en las fechas de los consumos se realiza en la consulta, y otro en el que la consulta únicamente busca los consumos de un cliente y es Spark o Storm el que hace el filtrado por fecha. Al igual que antes, se han utilizado las mejores configuraciones posibles tanto en Spark como en Storm.

Curiosamente el comportamiento en ambos sistemas es diferente. Probablemente dicha diferencia sea debida a que en Spark el cómputo se realiza al finalizar el RDD en conjunto. Esto produce una gran carga para la base de datos en un momento dado

incurriendo en retrasos. Por lo tanto, Spark obtiene mejor rendimiento sobrecargando menos a Cassandra en ese instante realizando una consulta más liviana. Mientras tanto, en Storm el tipo de consulta no tiene ningún impacto ya que las consultas se realizan a nivel de tupla.

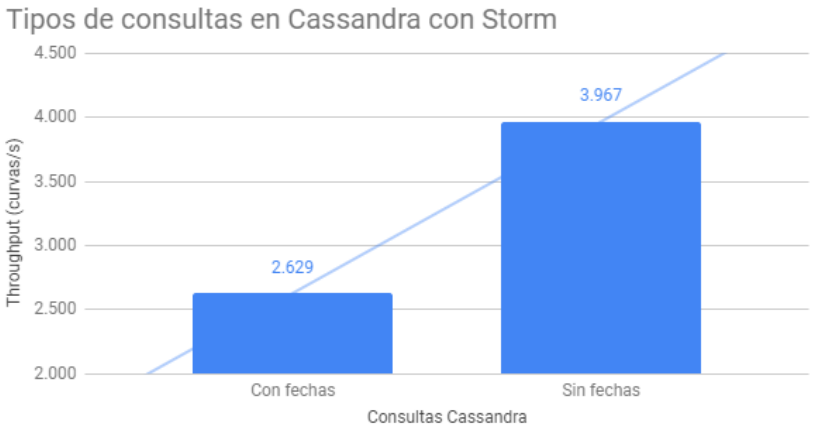


*Ilustración 40. Throughput obtenido en el benchmark del sector eléctrico dependiendo del tipo de consulta en Storm*

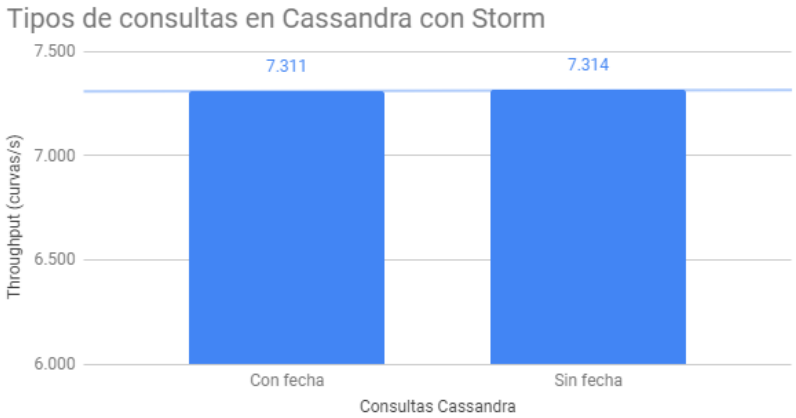


*Ilustración 41. Throughput obtenido en el benchmark del sector eléctrico dependiendo del tipo de consulta en Storm*

Como se puede observar en los resultados a lo largo de todo el *benchmark*, Storm es un claro vencedor y, es que, el cómputo del caso de uso es a nivel de tupla, que precisamente es donde Storm obtiene sus mejores rendimientos, mientras que Spark se queda atrás debido al factor del *batch*, que le limita en este aspecto.

## Conclusiones

La arquitectura orientada al dato surge como marco arquitectónico para ingerir, procesar y generar valor de las ingentes cantidades datos de distinta tipología que llegan al sistema a gran velocidad, lo que se conoce como *streaming*. Debido a la reciente aparición de estas tecnologías y la falta de experiencia real para su configuración y despliegue, este trabajo fin de máster tenía por objeto implementar diferentes configuraciones de esta arquitectura para atender a distintos casos de uso de la Industria 4.0 y, como consecuencia de su comportamiento, caracterizar las tecnologías que intervienen.

Se ha trabajado con tres casos de uso y su despliegue se ha realizado en un servidor *on-premise*, un supercomputador con 16 nodos reservados para la tarea, así como contratando 16 nodos en la nube.

Este trabajo nos permite concluir que hay una gran cantidad de tecnologías en este ámbito y todas ellas sirven para un conjunto de casos de usos concretos y dependiendo de cada situación es mejor utilizar unas u otras, no existe un claro vencedor, excepto en el ámbito de gestión de colas en el que Kafka es el líder indiscutible. Sí se puede decir que, si el procesado de eventos requiere operaciones de agregación dentro de un marco temporal, Spark sería la mejor elección por su uso de RDDs y, en cambio, Storm sería adecuado si el procesamiento se realiza a nivel de filas. Otro aspecto a considerar es el comportamiento necesario de tolerancia a fallos, siendo *exactly-once* en Spark streaming y *at-least-once* en Storm.

Respecto al desarrollo y mantenimiento del código, las librerías de Spark permiten una programación muy sencilla puesto que con muy pocas líneas permite realizar una gran cantidad de operaciones, mientras que Storm tiene una librería de más bajo nivel, por lo que puede llegar a ser más complejo el desarrollo de la aplicación. No obstante, la interfaz gráfica de Storm es bastante más potente a la hora de monitorizar los procesos ya que debido a la granularidad con la que se implementan las aplicaciones te permite identificar cuellos de botella rápidamente.

Por otra parte, este trabajo también nos permite señalar la necesidad de desarrollar herramientas de *benchmark* que faciliten la ejecución sistemática de pruebas de rendimiento fácilmente configurables para los casos de uso de las organizaciones, iniciativa que está llevando a cabo el grupo ISTR [23].

En resumen, este trabajo aporta las siguientes contribuciones:

- Orientaciones para la elección y diseño de una arquitectura orientada al dato con tecnologías *big data*.
- Información detallada para la configuración en clúster de las tecnología probadas.
- Descripción de los parámetros de configuración más relevantes de cada tecnología y cómo ajustarlos para obtener un mejor rendimiento.
- Análisis del comportamiento de cada tecnología en diferentes casos de uso.
- Configuración óptima para resolver un caso de uso real.

No obstante, conviene señalar que solo se ha cubierto un subconjunto de las tecnologías disponibles en el mercado. Por lo tanto, de cara al futuro sería necesario realizar el mismo estudio otras tecnologías disponibles con el fin de ofrecer un mayor abanico de posibilidades. Por otro lado, la necesidad de diseñar un *benchmark* estándar con una batería de casos de uso para probar estas tecnologías es algo necesario e incluso urgente, puesto que cada vez un mayor número de empresas están adoptando estas tecnologías.

# Bibliografía


[1] SINTEF, «Big Data, for better or worse: 90% of world's data generated over last two years.,» ScienceDaily, 22 Mayo 2013. [En línea]. Available: www.sciencedaily.com/releases/2013/05/130522085217.htm. [Último acceso: 10 Marzo 2018].

[2] G. Orenstein, C. Doherty, K. White y S. Camiña, «Building Real-Time Data Pipelines,» O'Reilly Media, Inc., 2015.

[3] i-SCOOP, «Industry 4.0: the fourth industrial revolution - guide to Industry 4.0,» i-SCOOP, [En línea]. Available: https://www.i-scoop.eu/industry-4-0/. [Último acceso: 10 Marzo 2018].

[4] Yahoo, «GitHub - Yahoo Streaming Benchmarks,» [En línea]. Available: https://github.com/yahoo/streaming-benchmarks.

[5] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, N. Tatbul, K. Tufte, H. Wang y S. Zdonik, «S-Store: A Streaming NewSQL System for Big Velocity Applications,» *VLDB Endowment,* vol. 7, nº 13, pp. 1633 - 1636, 2014.

[6] P. T. M. B. A. Devlin, «An architecture for a business and information system,» vol. 27, nº 1, 1988.

[7] S. G. Jeffrey Dean, «MapReduce: simplified data processing on large clusters,» vol. 6, 2004.

[8] Ç. U. S. Z. Stonebraker M., «The 8 requirements of real-time stream processing,» *Newsletter ACM SIGMOD,* vol. 34, nº 4, pp. 42-47, 2005.

[9] N. Marz, «How to beat the CAP theorem,» 13 Octubre 2011. [En línea]. Available: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[10] K. J., «Questioning the lambda architecture,» Julio 2014. [En línea]. Available: https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

[11] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal y D. Ryaboy, «Storm@twitter,» *SIGMOD '14 Proceedings of the 2014 ACM SIGMOD International Conference on Management,* pp. 147-156, 2014.

[12] A. Lakshman y P. Malik, «Cassandra - A Decentralized Structured Storage System,» *ACM SIGOPS Operating Systems Review,* vol. 44, nº 2, pp. 35-40, 2010.

[13] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps y J. Stein, «Building a Replicated Logging System with Apache Kafka,» vol. 8, nº 12, Agosto 2015.

[14] F. Ryan, «The Rise and Rise of Apache Kafka,» RedMonk, 4 Febrero 2016. [En línea]. Available: https://redmonk.com/fryan/2016/02/04/the-rise-and-rise-of-apache-kafka/. [Último acceso: 10 Marzo 2018].

[15] J. C., «Billions of Messages a Day - Yelp's Real-time Data Pipeline,» 14 Julio 2016. [En línea]. Available: https://engineeringblog.yelp.com/2016/07/billions-of-messages-a-day-yelps-real-time-data-pipeline.html. [Último acceso: 10 Marzo 2018].

[16] K. Gade, «Real-time analytics at Pinterest,» 18 Febrero 2015. [En línea]. Available: https://medium.com/@Pinterest_Engineering/real-time-analytics-at-pinterest-1ef11fdb1099. [Último acceso: 10 Marzo 2018].

[17] Pivotal, «Messaging that just works — RabbitMQ,» [En línea]. Available: https://www.rabbitmq.com.

[18] Apache, «Apache Kafka Documentation,» [En línea]. Available: http://kafka.apache.org/documentation.

[19] R. Yao, «Kafka 0.10 Compression Benchmark,» 3 Enero 2017. [En línea]. Available: http://blog.yaorenjie.com/2017/01/03/Kafka-0-10-Compression-Benchmark/.

[20] TPC, «About the TPC,» [En línea]. Available: http://www.tpc.org/information/about/abouttpc.asp.

[21] M. Zhang y S. Zhong, «GitHub - Storm Benchmark,» Intel, [En línea]. Available: https://github.com/intel-hadoop/storm-benchmark.

[22] J. Kreps, «Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines),» 27 Abril 2014. [En línea]. Available: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines.

[23] L. Martín de la Rubia, M. Algorri, M. Zorrilla y J. M. Drake, «Descripción de pruebas de benchmark para plataformas de tercera generación,» de *JISBD 2018*, 2018.