Arquitecturas en streaming
© EDICIONES ROBLE, S.L.

Indice

Arquitecturas en streaming

I. Introducción

En la corta historia del Big Data hay dos tendencias tecnológicas que conviven y permiten afrontar los problemas de procesamiento de grandes volúmenes de datos de manera diferente.

La primera consiste en procesar los datos allí donde se encuentran y la encarna el ecosistema Apache Hadoop. Una de las limitaciones de esta tecnología es la dilatada latencia de procesamiento en lo que se conoce como *batch,* de modo que los resultados se obtienen en un rango de minutos u horas.

Una opción alternativa es procesar los datos *en tránsito* para que los resultados se obtengan en segundos e incluso en tiempo real. Consiste en procesar flujos de datos o *streams* para obtener resultados prácticamente inmediatos y poder utilizarlos para la toma de decisiones rápidas.

Esta unidad introduce el concepto de *streaming,* su utilidad y las soluciones arquitectónicas y tecnológicas que actualmente se usan para el procesamiento de grandes datos en tránsito.

Así pues, la presente unidad explora el concepto de *streaming,* la problemática que este resuelve, e introduce las tecnologías involucradas y la algoritmia aproximada que se usa en este tipo de procesamiento. Esta unidad tiene carácter conceptual; por ello, excepcionalmente no dispone de caso práctico, sino que al final solamente se facilita el cuestionario a modo de repaso sobre los conceptos teóricos introducidos.

II. Objetivos

Al término de esta unidad, los alumnos habrán alcanzado los siguientes objetivos:

- Diferenciar casos de uso en los que se aplica el procesamiento de *streams* de información.
- Interiorizar el patrón básico de arquitectura para procesamiento de *streams,* conocido como arquitectura Kappa.
- Reconocer las múltiples soluciones tecnológicas para el procesamiento de *streams.*

III. Conceptos y casos de uso

La clave que define el concepto de *streaming* es la capacidad de analizar los datos según llegan y, de forma opcional, tomar decisiones prácticamente de inmediato.

Para dar contenido a la definición informal anterior, pensemos en algunos tipos de fuentes que emiten información en forma de *flujos de datos:*

Eventos deportivos

Cualquiera que haya seguido la jornada futbolera dominical ha experimentado el flujo constante y atropellado de *eventos* en diferentes partidos a lo largo de las horas.

La bolsa

Los mercados bursátiles constituyen el ejemplo paradigmático de flujos de eventos de cotizaciones de valores. De hecho, las primitivas soluciones tecnológicas de soporte a la bolsa son las primeras diseñadas para trabajar con eventos en tiempo real.

Twitter

Esta red social es el ejemplo más citado en nuestros días. En Twitter se plantea de forma muy evidente el problema más básico que se aborda en flujos de datos: *contar ocurrencias* de eventos.

Clickstream[1]

El seguimiento de la actividad de un usuario de una página (por ejemplo, de comercio electrónico) es clave para medir y optimizar (por ejemplo, facilidad de uso, interés del usuario o productividad).

[1]Wikipedia: "Clickstream" (en inglés). [En línea] URL disponible en: https://en.wikipedia.org/wiki/Clickstream

Si acometemos el problema de tratar flujos de eventos desde una perspectiva técnica, tenemos dos opciones:

1. Almacenar los eventos o datos en disco y procesarlos después.
2. Analizar o procesar los eventos en tránsito directamente en memoria.

La primera opción es la que comúnmente se conoce como *procesamiento batch* o procesamiento diferido. Parece evidente que esa no es la aproximación que sigue Twitter a la hora de contar tuits, ni mucho menos la que usan los sistemas automáticos de *trading* algorítmico para operar en bolsa. Se hace necesario, en determinadas circunstancias, analizar la información en tránsito independientemente de que también se vuelque a disco para otro tratamiento posterior.

La segunda opción es la de *procesamiento de streams* y es el objeto de esta unidad.

3.1. Casos de uso de *streaming analytics*

Llevar a cabo un análisis en tiempo real puede aportar beneficios evidentes con respecto a realizar ese mismo proceso en diferido: es posible actuar en el momento. Veamos algunos ejemplos:

Detección de fraude

Uno de los productos básicos de los bancos son las tarjetas de crédito y débito. Los bancos registran todas las operaciones realizadas. Es una problemática muy estudiada (y de hecho solucionada) la de detectar fraudes prácticamente en tiempo real. Imaginemos que con una determinada tarjeta se realizan dos movimientos en el mismo día en puntos muy separados del globo. Es posible que el cliente esté de viaje y esa situación sea legítima. O que no lo sea y se trate de un fraude con tarjetas clonadas. En este caso, es evidente la necesidad de bloquear dicha tarjeta en tiempo real y avisar a su titular de la situación.

Detección de intrusiones

Es un caso de uso muy parecido al anterior y es posible experimentarlo fácilmente. Supongamos que accedemos a nuestro correo electrónico, por ejemplo de Gmail, tanto a través de nuestra conexión habitual en casa, en España, como a través de una VPN con salida en una región del mundo diferente, como Estados Unidos. Google interpreta de inmediato esta situación como una anomalía e informa de ello en el momento.

Monitorización de logs

El almacenamiento es hoy lo bastante barato como para conservar los registros de actividad de servidores y aplicaciones con varios propósitos. Sobre dichos *logs* es posible hacer estudios forenses o análisis de causas raíz de un problema técnico (una caída del servicio) o una intrusión. Si dichos problemas son recurrentes, tiene todo el sentido expresarlos como reglas que se puedan aplicar a la vez que se generan esos *logs* para actuar de inmediato. Por ejemplo, cuando en la monitorización de aplicaciones o servidores se observa un pico de demanda, es obvio reaccionar dotando al sistema automáticamente de más recursos para escalar adecuadamente. Este es uno de los casos de uso preferentes a la hora de experimentar por primera vez con infraestructuras tecnológicas de almacenamiento masivo y análisis en tiempo real.

Comportamiento de clientes o usuarios

Soluciones como Google Analytics o equivalentes *open source* como Open Web Analytics[2] permiten seguir el comportamiento de los usuarios en nuestra web de forma diferida. La ventaja diferencial está en reaccionar en tiempo real a su comportamiento con una oferta personalizada durante la propia navegación o en la misma compra. Ocurre lo mismo con la publicidad *online* que se nos presenta en tiempo real en función de la microsegmentación que se tiene de nosotros como usuarios de múltiples webs y servicios *online.* La cuestión clave es ofertar servicios y productos en el momento y adaptados a nuestras preferencias.

[2]Página web de Open Web Analytics. [En línea] URL disponible en: http://www.openwebanalytics.com/

Trading algorítmico

La mayor parte del volumen de transacciones en bolsa la realizan las plataformas de *high-frequency trading*[3]. Estas plataformas operan con latencias enormemente bajas y es el ejemplo más extremo de procesamiento de eventos y acción en tiempo real.

[3]Wikipedia: "High-frequency trading" (en inglés). [En línea] URL disponible en: https://en.wikipedia.org/wiki/High-frequency_trading

Gestión de smart grids

Aplicaciones para *smart devices* (Smart Car, Smart Home). Las redes de generación y distribución de energía, las redes de gestión integral de aguas, los ferrocarriles de alta velocidad o la cadena de producción de automóviles son algunos ejemplos de redes que hoy se gestionan de forma telemática y remota. Estas redes generan un flujo de datos constante desde sensores y concentradores desplegados por toda la red. En general, la problemática que hoy conocemos bajo el nombre de IoT implica recibir, procesar y reaccionar en tránsito a la información generada por la red de sensores. Imaginemos, por ejemplo, un parque eólico de generación de electricidad en pleno vendaval: es necesario desarmar los molinos para no comprometer su integridad. Hoy es posible hacerlo en tiempo real a partir de la información generada en sensores meteorológicos y de los propios aerogeneradores que emiten señales para el desarmado. Es más, es posible incluso predecir tales situaciones mediante *machine learning* y anticipar estas actuaciones.

Monitorización de salud de pacientes

La tecnología permite mejorar la calidad de vida de pacientes crónicos y llevar un seguimiento de su estado de salud sin necesidad de hospitalización, a la vez que se optimizan los costes de la atención. Un paciente crónico monitorizado puede enviar alertas en tiempo real que desencadenan desde una llamada del personal sanitario para interesarse por lo que sucede hasta la planificación de recursos para anticipar un ingreso hospitalario.

Analítica deportiva

Las competiciones de deportes de equipo generan un volumen de negocio considerable y son muy sensibles a toda innovación. Un escenario en el que un partido de fútbol se monitorice en tiempo real y proporcione información de contexto al entrenador para adaptar la estrategia durante el partido es perfectamente viable desde el punto de vista técnico[4].

[4]Srinath Perera. *Football Analytics through a Stream Processing Lens.* YouTube. 2014. [En línea] URL disponible en: https://www.youtube.com/watch?v=nRl6buQ0NOM

Inventario en tiempo real

Los sistemas de inventario en tiempo real están diseñados para actualizar los datos de *stock* en tiempo real. Por ejemplo, en tiendas es posible actualizar el inventario tan pronto como se vende un ítem o incluso cuando se cambia de estantería dentro de una misma tienda. Los *retailers* necesitan este tipo de soluciones para implementar estrategias de venta multicanal (*online* y en tienda física, al menos). Los inventarios en tiempo real son imprescindibles para mejorar la acción de compra y la eficiencia operacional, por ejemplo, con la opción de recoger el artículo comprado en una tienda cercana al domicilio.

Monitorización de flotas, tráfico rodado

Los operadores de flotas como empresas de seguridad, operadores logísticos o *carriers* e incluso las aerolíneas se enfrentan desde hace tiempo al problema de optimizar las rutas adaptando el uso de los vehículos para maximizar los ingresos. Esta cuestión es sensible a múltiples factores (fenómenos climáticos, atascos de tráfico), de modo que la posibilidad de monitorizar y reconfigurar las rutas en tiempo real proporciona un valor diferencial con respecto a la competencia.

Optimización de la cadena de suministro

Se trata de un caso de uso que combina algunos otros de los mencionados y que pretende actuar en todo el ciclo empezando por la planificación de demanda, compras y producción hasta la distribución, venta, facturación y posventa. El reto está en que las empresas implantan un *software* para cada paso de la cadena (ERP, CRM, etc.), que actúan como silos de información en los que los intercambios de datos se basan en procesos de exportación e importación de bases de datos, lo que hace virtualmente imposible ver la cadena entera y actuar sobre ella en tiempo real.

Sea cual sea el caso de uso desde la perspectiva del negocio, el análisis en *streaming* aporta la posibilidad de anticipar necesidades, comprender cuándo cambian esas necesidades, adaptarse a los cambios y, en definitiva, tener la posibilidad de ser proactivo frente a un mundo cambiante.

IV. Patrones de arquitectura: arquitectura Kappa

Con sistemas de procesamiento *batch* no es viable detectar cambios lo bastante rápido como para responder a ellos antes de que empiecen a afectar a la base de clientes y a la operación de una organización.

Sin embargo, usando procesamiento de *streams* ya hemos dicho que podemos tomar decisiones sobre los datos en tránsito.

El procesamiento de *streams* complementa tecnologías como Hadoop, pero no las sustituye. Fijémonos en que en el patrón básico de arquitectura Lambda[5] visto en una unidad anterior se integran el procesamiento *batch* y el procesamiento en tiempo real (la *batch layer* y la *speed layer* respectivamente).

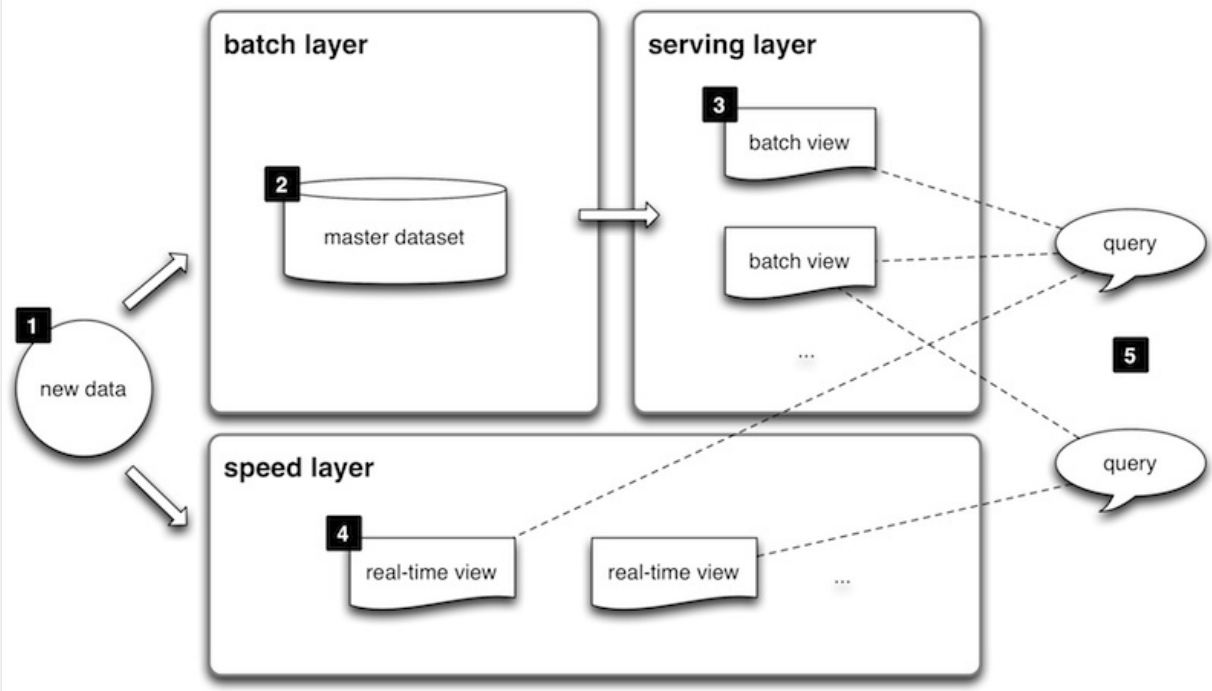

Figura 1. Patrón de arquitectura Lambda.
Fuente: http://lambda-architecture.net/

Introduciremos ahora un nuevo patrón básico de arquitectura específicamente dedicado a procesamiento de *streams:* la arquitectura Kappa.

[5]Wikipedia: "Lambda architecture" (en inglés). [En línea] URL disponible en: https://en.wikipedia.org/wiki/Lambda_architecture y http://lambda-architecture.net/

4.1. Arquitectura Kappa

El patrón de arquitectura Kappa surge del cuestionamiento de la aproximación Lambda[6], por parte de Jay Kreps[7], como bala de plata aplicable a cualquier problema de tratamiento y analítica de grandes volúmenes de datos.

[6]Artículo de Jay Kreps, publicado en 2014. [En línea] URL disponible en: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

[7]Jay Kreps desarrolló su carrera profesional en LinkedIn. Es uno de los creadores de proyectos *open source* más relevantes como Apache Kafka, Samza y Voldemort. Actualmente, es CEO en Confluent, empresa que ofrece una *suite* de procesamiento de *streaming* construida alrededor de Kafka. Su perfil profesional está en: https://www.linkedin.com/in/jaykreps/

Kreps argumenta que en la arquitectura Lambda es necesario mantener dos *codebases* para realizar el mismo análisis, un *codebase* en la *batch layer* y otro diferente en la *speed layer.*

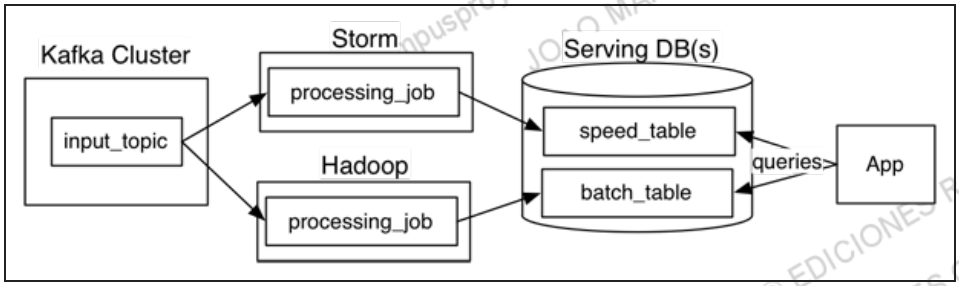

Figura 2. Lambda: dos codebases. Fuente: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

En la figura 2, para un mismo *stream* de información el procesamiento en tiempo real se implementaría en Java usando el *framework* Apache Storm, mientras que el procesamiento *batch* se implementa en algún *framework* o herramienta del ecosistema Apache Hadoop (Map Reduce, Pig, Hive) dando lugar al menos a dos programaciones diferentes para un mismo tratamiento de los datos, lo que obviamente es una duplicidad de esfuerzo que convendría evitar.

La propuesta de Kreps es usar una única aproximación que sirva tanto para los datos históricos como para los datos que llevan en *streams.*

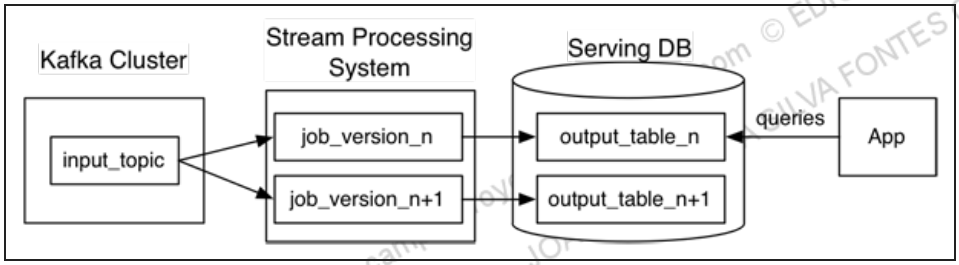

Figura 3. Mejora propuesta por Jay Kreps a la arquitectura Lambda. Fuente: https://www.oreilly.com/ideas/questioning-the-lambda-architecture

La propuesta de Jay Kreps (figura 3) se basa en solucionar la capa de procesamiento con una solución de procesamiento de *streams.* En caso de necesitar reprocesar el histórico porque las reglas de procesamiento cambien (job_version_n+1 en la figura), solo es necesario alimentar el sistema con los eventos históricos. La aplicación consumidora de los datos procesados leerá entonces los datos de una nueva versión del almacenamiento (output_table_n+1 en la figura) y se elimina la necesidad de que esa aplicación que usa Lambda tuviese que consultar tanto en resultados de la *speed layer* como en los de *batch layer* (ver figura 2).

La figura 4 formaliza el patrón de arquitectura Kappa de tal modo que es posible compararlo fácilmente con la Lambda.

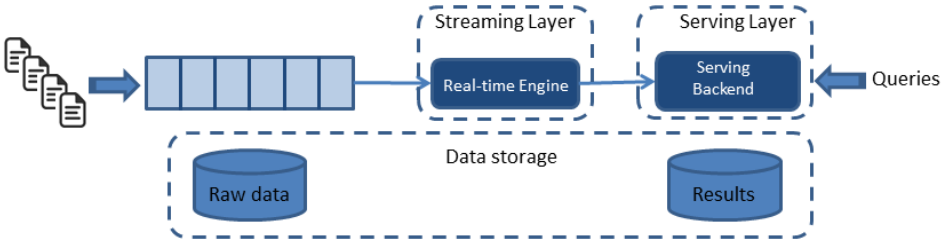

Figura 4. Arquitectura Kappa. Fuente: https://www.oreilly.com/ideas/applying-the-kappa-architecture-in-the-telco-industry

Las diferencias entre Kappa y Lambda son las siguientes:

- Ya no existen dos niveles de procesamiento diferentes para los datos frescos y los históricos.
- La capa de exposición de datos procesados ahora solo usa una base de datos de resultados.

Hasta ahora hemos hablado de la arquitectura Kappa en contraposición con la original Lambda. Hoy en día, Kappa es un patrón ampliamente utilizado y de viabilidad comprobada.

Principios básicos

La arquitectura Kappa en sí misma representa cuatro principios básicos:

1. **Todo es un *stream* de datos.** Desde esta perspectiva el procesamiento *batch* es un subconjunto del procesamiento de *streams,* de ahí que todo sea un *stream.*
2. **Las fuentes de datos son inmutables.** Los datos deben conservarse tal cual llegan de las fuentes. Siempre es posible recalcular el estado de un dato dado que tenemos disponibles su estado inicial y sus cambios.
3. **Se usa un único *framework* para análisis y procesamiento.** Se usa el principio KISS[8] para simplificar la arquitectura y, como consecuencia, los esfuerzos de codificación, mantenimiento y actualización disminuyen.
4. **Recálculo de resultados.** Si los cálculos necesarios en el procesamiento cambian o evolucionan en el tiempo, siempre es posible recalcular los resultados simplemente reintroduciendo los datos históricos en el *stream.* Y todo ello es posible porque, como vimos en el segundo principio, tenemos almacenados los datos originales desde el inicio. Para que se cumpla este cuarto principio, es necesario que se repitan los cálculos usando el mismo orden en el procesamiento de los eventos. Si esto no se garantiza, los resultados serán inconsistentes.

[8]Wikipedia: "Principio KISS". [En línea] URL disponible en: https://es.wikipedia.org/wiki/Principio_KISS

Elementos

Para implementar una arquitectura Kappa, necesitamos entonces los siguientes elementos:

- Una capa de transmisión de datos (*message passing layer*).
- Un *framework* de procesamiento los datos (*stream processing layer*).
- Un sistema de almacenamiento y exposición de resultados (*serving layer*). Lo mismo que en la arquitectura Lambda.

Los componentes tecnológicos habituales para implementar las dos primeras capas los introduciremos en el siguiente epígrafe.

Características

En una arquitectura Kappa desearíamos tener resultas las características siguientes:

- **Escalable.** Nos gustaría que el sistema se adaptase en el tiempo y de la forma más transparente posible a diferentes cargas de trabajo. Imaginemos el uso de Twitter durante una catástrofe o un evento de carácter mundial, o bien el flujo de clics de una web de comercio *online* durante días de oferta como el *Black Friday* o similares.
- **Tolerante a fallos.** Que nos aporte garantías de que, aun con fallos en el sistema, los eventos se siguen procesando. Es decir, que no pierde eventos o que todos los eventos se procesan.
- Que los eventos solo se procesan **una vez.** Esto puede verse como un añadido a la necesidad anterior o como un fin en sí mismo. Veremos más adelante que se trata de un deseo difícil de cumplir.

En la próxima unidad detallaremos los componentes tecnológicos habituales y cómo cumplen tales características.

V. Componentes tecnológicos

En este epígrafe haremos una breve recopilación de herramientas, *frameworks* o productos del ámbito *open source* que se usan habitualmente para construir sistemas de procesamiento de *streams* basados en la arquitectura Kappa.

En la próxima unidad trataremos con más detalle los más importantes.

5.1. Adquisición y transmisión de datos

A la hora de construir el *pipeline* de adquisición de datos se suele usar principalmente Apache Kafka. Kafka es un producto de publicación y suscripción de mensajes enormemente escalable y de alto rendimiento. Kafka proporciona las "tuberías" en las que se "mueven" los eventos del *stream.*

Visto como un sistema de colas de mensajes persistentes en disco, Kafka da solución al almacenamiento de datos en crudo (ver figura 4) de la arquitectura Kappa. Nos sirve entonces como almacenamiento inmutable de los datos del *stream* (segundo principio de la arquitectura Kappa). Si quisiéramos reprocesar de nuevo el *stream* solo tendríamos que reprocesar toda la cola de mensajes o eventos desde el principio (cuarto principio de la arquitectura Kappa).

Kafka dispone de conectores con distintas fuentes y destinos (denominados *sinks*) de datos. Con estos conectores podemos alimentar Kafka desde bases de datos, ficheros, otras plataformas de mensajería y hasta cadenas de *blockchain.* Del mismo modo, podemos volcar mensajes de Kafka a dichos sistemas.

A la hora de la recepción de datos en Kafka, Apache Flume es un complemento ideal allí donde Kafka no tiene un conector. Apache Flume solo implementa la capacidad de adquisición de datos y no las de persistencia o reprocesado, de tal modo que Flume no es un reemplazo de Kafka, pero sí un complemento que merece la pena valorar.

En cuanto a soluciones en la nube, Amazon AWS proporciona el servicio Kinesis Streams, en Google Cloud Platform se ofrece Pub/Sub y, por su parte, Microsoft Azure dispone del servicio Event Hubs. Es importante hacer notar que estos servicios en la nube no proporcionan *per se* el almacenamiento que sí tiene Kafka, de modo que deberíamos complementarlos con servicios de almacenamiento para ser capaces de cumplir el segundo principio de la arquitectura Kappa.

5.2. Procesamiento de datos

Las posibilidades de elección de un *framework* de procesamiento de datos en *streaming* son amplias hoy en día.

Apache Spark Streaming y Apache Storm son quizá las más nombradas y utilizadas, de modo que podríamos considerarlas los estándares de facto. Storm ha tenido momentos mejores y hoy en día puede considerarse una solución en retirada si la comparamos con el uso que ha llegado a tener Spark.

Apache Flink, Apache Beam y Apache Apex son soluciones muy válidas, aunque menos populares en cuanto a base instalada. Todas ellas (al igual que Spark) dicen proporcionar un *framework* unificado para procesamiento *batch* y de *streams* y son los dos últimos los que mejor cumplen esa promesa (aunque también los menos extendidos). Apache Beam tiene en Google un potente aliado para su extensión en el mercado dado que Google Cloud Platform ofrece un servicio gestionado de ejecución de *pipelines* desarrollados en Beam.

Apache Samza es una opción más para añadir al maremágnum de soluciones. Lo mencionamos aquí más como curiosidad dado que se sigue mencionando como uno de los primeros ejemplos de este tipo de plataformas, que en su día cuando no existía tanta diversidad fue notable dado que venía de ser una solución (como Kafka) desarrollada y probada en LinkedIn, una de las empresas que reconoce estar, desde el punto de vista tecnológico, construida bajo un paradigma de *streaming.*

Y a las alternativas anteriores cabe añadir que Kafka cuenta hoy con una solución básica de procesamiento de nombre Kafka Streams. Confluent (la empresa actual de Jay Kreps, que da soporte y servicios empresariales con Apache Kafka) ha anunciado recientemente la próxima liberación de Kafka KSQL para procesamiento de *streams* usando sintaxis SQL.

Otros criterios

Frente a tanta diversidad de opciones, hay que recordar que las diferencias técnicas entre todas ellas en muchos casos son sutiles y los criterios técnicos de selección de una u otra lo serán en la misma medida. Otros criterios muy aconsejables son el volumen de desarrolladores con experiencia disponibles en el mercado y la actividad de la comunidad *open source* que los desarrolla y mantiene.

1

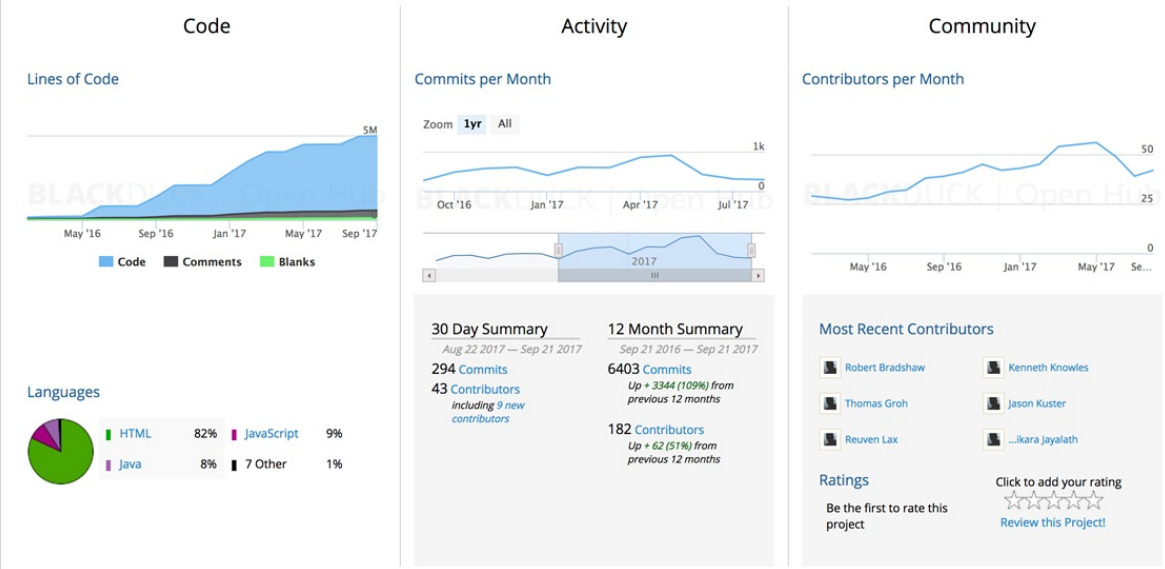

Figura 5. Estadísticas de evolución del proyecto open source Apache Beam. Fuente: Open Hub, https://www.openhub.net/p/apache-beam

2

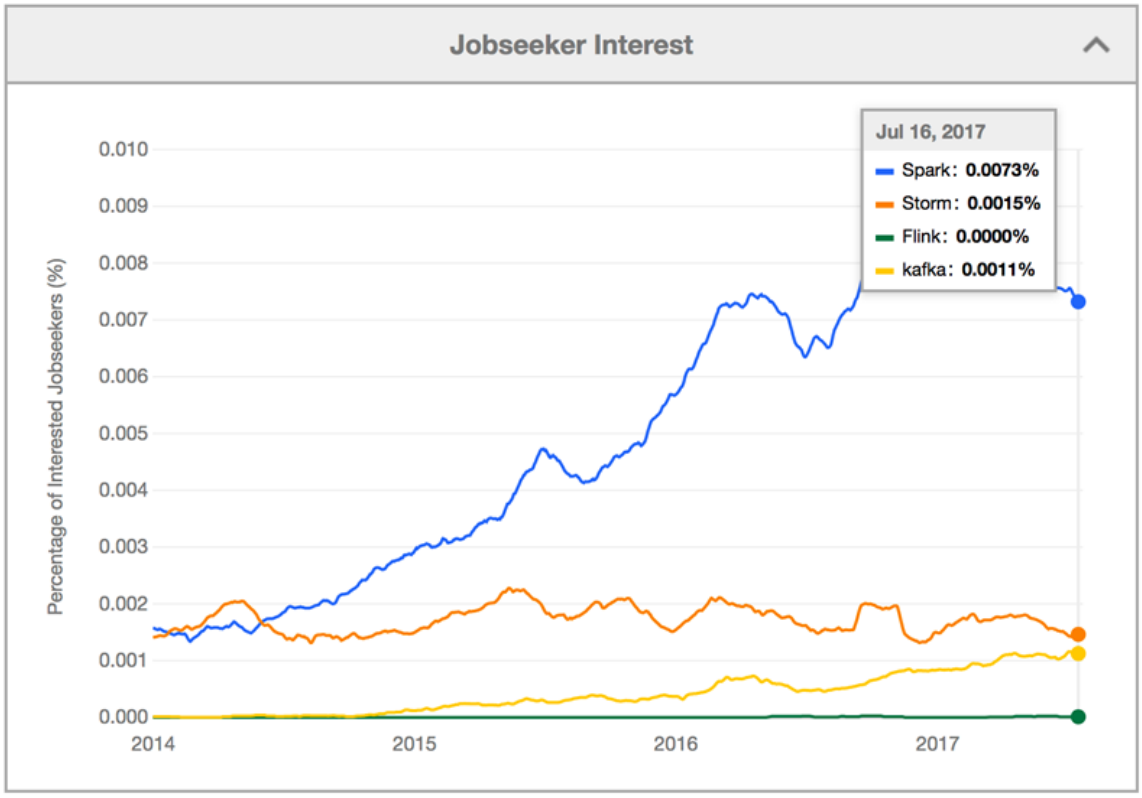

Figura 6. Interés de reclutadores en EE. UU. para diferentes frameworks de procesamiento de streams. Fuente: Indeed, https://www.indeed.com/jobtrends/q-Spark-q-Storm-q-Flink-q-kafka.html

3

La figura 7 presenta una categorización de diferentes herramientas técnicas en función del tamaño de los datos en tránsito frente al tiempo aceptable para su procesamiento.

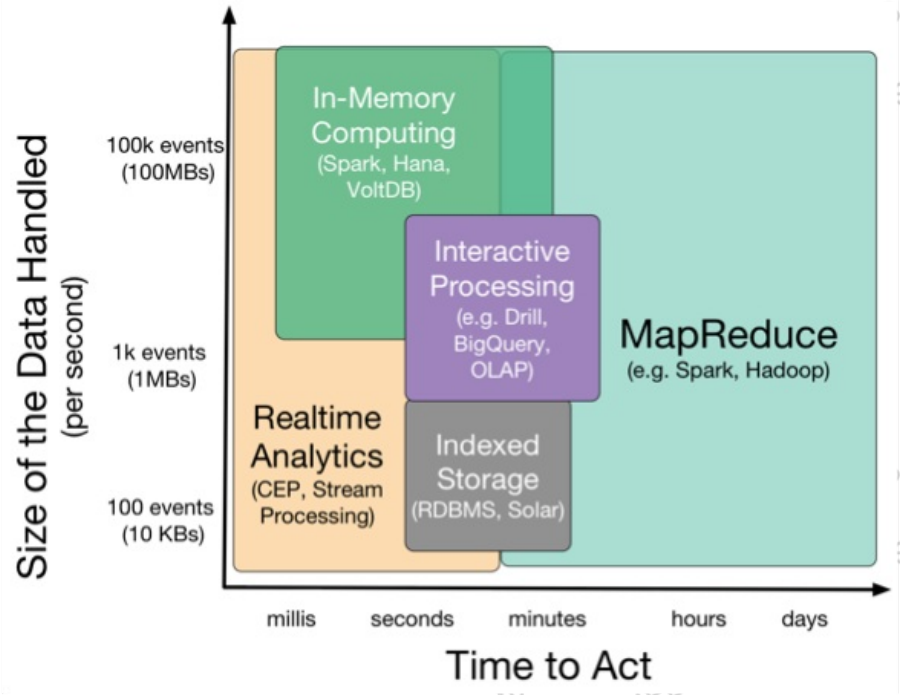

Figura 7. Tipos de herramientas en función del volumen de los datos y el tiempo de procesamiento. Fuente: http://srinathsview.blogspot.com.es/2014/12/realtime-analytics-with-big-data-what.html

4

Se puede observar que las soluciones de *streaming* tratan de cubrir las necesidades de procesamiento en tiempo casi real (de milisegundos al minuto), que es precisamente donde las soluciones de procesamiento *batch* no tienen tiempos de procesamiento aceptables en ese rango. La diferencia fundamental entre unas y otras es su aproximación técnica básica, en la que Hadoop opera sobre disco persistente mientras que en *streaming,* para grandes volúmenes de datos, es necesario operar en memoria para evitar la latencia de las operaciones de lectura y escritura en disco.

En la figura 8 se pueden observar algunas diferencias técnicas de las plataformas de procesamiento de *streams.*

Property	Structured Streaming	Spark Streaming	Apache Storm	Apache Flink	Kafka Streams	Google Dataflow
Streaming API	incrementalize batch queries	integrates with batch	separate from batch	separate from batch	separate from batch	integrates with batch
Prefix Integrity Guarantee	✔	✔	✘	✘	✘	✘
Internal Processing	exactly once	exactly once	at least once	exactly once	at least once	exactly once
Transactional Sources/Sinks	✔	some	some	some	✘	✘
Interactive Queries	✔	✔	✘	✘	✘	✘
Joins with Static Data	✔	✔	✘	✘	✘	✘

Figura 8. Diferencias entre plataformas de procesamiento de streams. Fuente: https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

5.3. Almacenamiento y exposición de resultados

En una arquitectura Kappa, las necesidades de almacenamiento son de dos tipos, tal como vimos en la figura 4:

1. El histórico del *stream.*
2. Los resultados del procesamiento del *stream.*

El histórico del *stream* es imprescindible para satisfacer el cuarto principio básico de la arquitectura dado que nos permite recalcular todos los resultados en caso de que adaptemos o modifiquemos las reglas de procesamiento del *stream.* Adicionalmente, el histórico del *stream* nos aporta tolerancia a fallos de modo que es posible reprocesar los datos justo desde el momento del fallo.

Apache Kafka nos permite almacenar[9] el histórico del *stream.* Kafka almacena en disco persistente las colas con los mensajes en tránsito. Un *back up* de estos ficheros desde el comienzo de la llegada de mensajes nos permitirá entonces recuperar todo el histórico del *stream* desde el principio de su existencia. Kafka no es una solución exclusiva para esta problemática, pero la apuntamos aquí como claramente ventajosa dado que nos permite solucionar con una sola pieza técnica tanto el problema de la transmisión de datos en *streaming* como el del almacenamiento persistente del mismo.

Por otra parte, el procesamiento del *stream* genera nuevos datos, desde simples agregaciones a resultados más complejos. En este caso se podrá utilizar toda la panoplia de soluciones de almacenamiento que también se usa en la arquitectura Lambda, desde bases de datos NoSQL (MongoDB, Cassandra, HBase, etc.) hasta bases relacionales más tradicionales. Hay que tener en cuenta que la *serving layer* es necesaria tanto en la arquitectura Lambda como en la Kappa, que nos ocupa en esta unidad.

[9] Introducción a Apache Kafka como sistema de almacenaje (en inglés). [En línea] URL disponible en: https://kafka.apache.org/intro#kafka_storage

VI. Ejemplos de arquitecturas de streaming

A continuación, presentaremos dos ejemplos de arquitecturas Kappa que aportan mejoras sustanciales con respecto a su resolución con una aproximación *batch.*

Los ejemplos suministrados son arquitecturas *bus-centric* articuladas alrededor de Apache Kafka para datos en tránsito, en contraposición con soluciones basadas en arquitecturas *data hub-centric* sobre un almacenamiento persistente de los datos.

6.1. Procesamiento de registros de actividad

El procesamiento de registros de actividad[10] (los comúnmente denominados *logs*) es un caso muy común de Big Data para el que el procesamiento en *streaming* supone una ventaja diferencial.

Pensemos en grandes empresas como bancos o telecos de enorme diversidad tecnológica con cientos o miles de sistemas y aplicaciones que generan registros de actividad. Estos registros contienen información valiosa susceptible de convertirse en conocimiento. Por ejemplo, cabe analizar los *logs* para prever posibles escenarios de problemas (la infraestructura no escala, una o varias aplicaciones fallarán en un escenario concreto, etc.) y anticiparse a ellos tomando medidas proactivas (aumentando la capacidad *hardware* o *software,* sustituyendo piezas antes de que se averíen, etc.).

Esta casuística se conoce comúnmente como *mantenimiento predictivo* y se aplica tanto a sistemas IT como a aviones, trenes o *smart grids.* Implica el procesamiento de grandes volúmenes de ficheros de registro de actividad (los *logs*) de múltiples fuentes (*hardware, software,* sensores, etc.). El procesamiento cercano al tiempo real de dicha información nos da esa posibilidad de actuar proactivamente poniendo soluciones frente a problemas que prevemos que aparecerán (la saturación o caída de un sistema informático, la avería en una pieza de un tren, etc.).

El procesamiento de esa información contenida en los *logs* consiste, en resumidas cuentas, en la detección de anomalías y la generación de una alerta en el momento de su detección. Una arquitectura Kappa que nos permite el procesamiento en *streaming* de los *logs* se muestra en la figura 9.

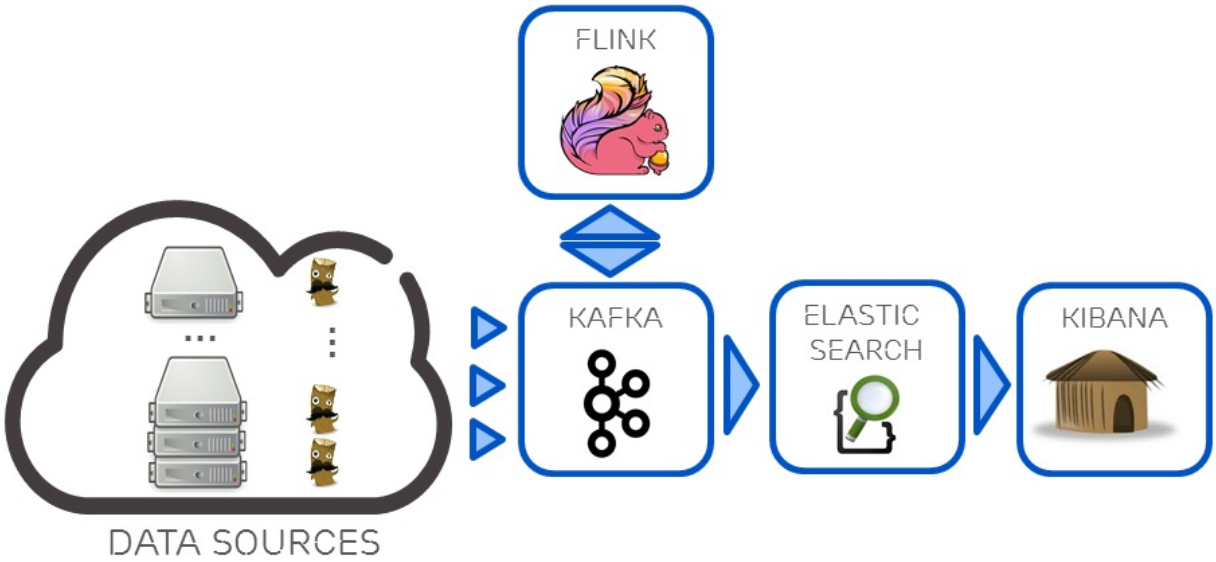

Figura 9. Arquitectura Kappa para un pipeline de datos. Fuente: https://www.oreilly.com/ideas/applying-the-kappa-architecture-in-the-telco-industry

Las fuentes de datos son servidores *hardware* y aplicaciones que se ejecutan sobre ellos y que generan registros de actividad en tiempo real. Para la entrada de *logs* se usa Logstash, que alimenta el *bus* de datos en Apache Kafka. Kafka distribuye los eventos recibidos a los diferentes *jobs* de procesamiento.

Un conjunto de *jobs* de Flink nos permitirá implementar la detección de anomalías sobre el *stream* de datos. Estos *jobs* implementan un procesamiento estadístico (por ejemplo, el cálculo de la frecuencia de *logging*) y la aplicación de un detector de anomalías (por ejemplo, un modelo bayesiano) para finalmente emitir nuevos eventos para las anomalías detectadas.

La información procesada y los nuevos eventos generados en ese procesamiento se almacenan para su visualización. En el ejemplo se usa Elasticsearch para almacenar e indexar la información y Kibana para su visualización.

[10]Nicolas Seyvet e Ignacio Mulas Viela: "Applying the Kappa architecture in the telco industry". [En línea] URL disponible en: https://www.oreilly.com/ideas/applying-the-kappa-architecture-in-the-telco-industry

6.2. Entrenamiento de modelos en *streaming*

Hemos introducido previamente Spark Streaming como herramienta técnica de procesamiento de *streams.* Apache Spark en su conjunto es una plataforma de computación distribuida que se ejecuta en un clúster para escalado horizontal transparente a los programas. Spark incluye el módulo MLlib con implementaciones de algoritmos comunes de *machine learning* que se ejecutan de manera distribuida en ese clúster. MLlib implementa además versiones compatibles con *streaming* de algunos de esos algoritmos, como por ejemplo k-means[11] y la regresión lineal[12]. Esto último significa que podemos entrenar y predecir con dichos algoritmos en tiempo casi real sobre *streams* de datos.

La figura 10 muestra una arquitectura lógica de procesamiento de *streams* en el que usamos parte de los datos de entrada para entrenar un *clustering* usando k-medias y usando el modelo de clasificación obtenido para predecir o clasificar los datos del *stream.*

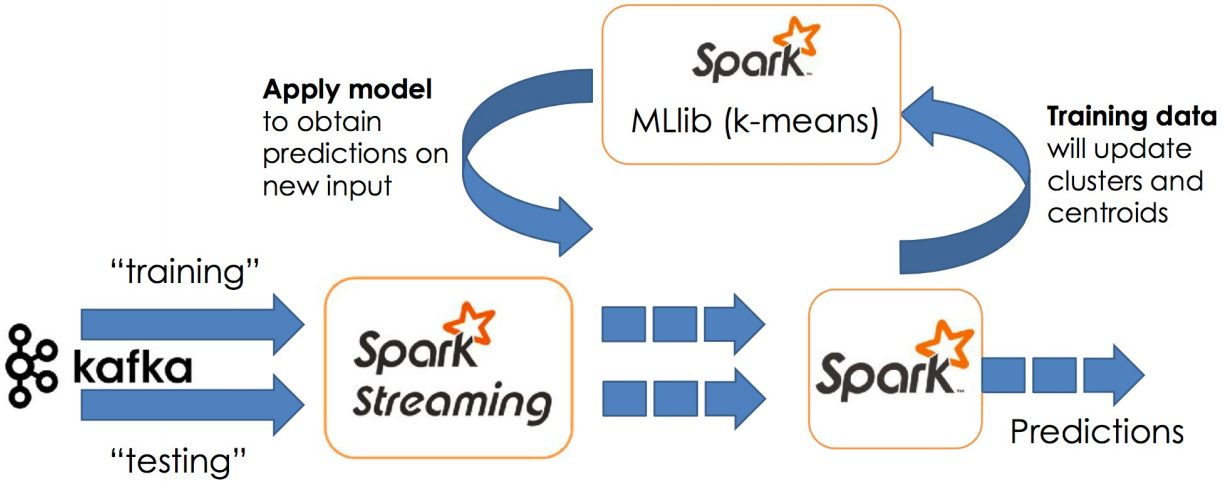

Figura 10. Entrenamiento de modelos de machine learning en streaming. Fuente: Felipe Ortega, https://www.linkedin.com/in/felipeortega/, https://www.meetup.com/es-ES/Big-Data-Developers-in-Madrid/events/238739273/

Es posible implementar esta arquitectura dado que, como hemos dicho, Spark MLlib incluye una implementación particular de k-means para *streaming* y podemos entrenar el modelo de clasificación de manera *online*. Si deseáramos usar otros algoritmos de implementación no compatibles con *streaming* nos veríamos obligados a usar una arquitectura Lambda en la que el entrenamiento del modelo se realiza de manera *offline* en la capa *batch* y la predicción con el modelo entrenado se hace en la *speed layer*.

[11]Guía MLlib (en inglés). [En línea] URL disponible en: https://spark.apache.org/docs/1.6.1/mllib-clustering.html#streaming-k-means

[12]Guía MLlib (en inglés). [En línea] URL disponible en: http://spark.apache.org/docs/2.2.0/mllib-linear-methods.html#streaming-linear-regression

VII. Algoritmos para procesamiento de *Streams*

Uno de los problemas más habituales con grandes volúmenes de datos es contar elementos únicos o calcular estadísticas sobre ellos.

Podemos necesitar contar el número de dispositivos activos o visitantes únicos de una página en un momento dado para medición de audiencias. Ese cálculo lo podemos hacer sobre datos consolidados o sobre eventos en tránsito.

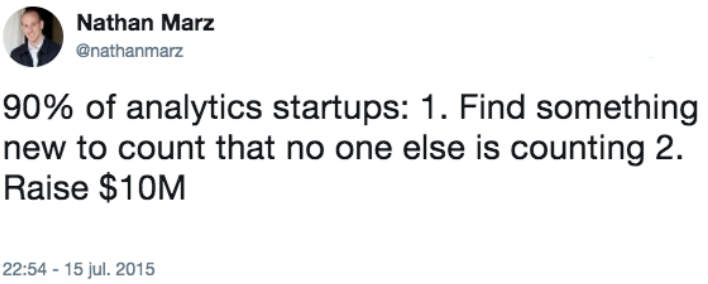

Figura 11. Nathan Marz (creador de Apache Storm) ironiza en este tuit sobre la génesis de startups y problemas de datos. Fuente: https://twitter.com/nathanmarz/status/621422681968259073

En este apartado introduciremos algunos algoritmos[13] básicos de cálculo especialmente diseñados para aplicarse a *streams* de muchos datos.

[13]Wikipedia: "Streaming algorithm" (en inglés). [En línea] URL disponible en: https://en.wikipedia.org/wiki/Streaming_algorithm

7.1. Contar elementos distintos – HyperLogLog

Imaginemos un *stream* de visitantes únicos de una web o un *stream* de IPs de paquetes que pasan a través de un *router.* Deseamos contar el número de visitantes o el número de IPs distintas. Nos estamos enfrentando a un problema de contar valores únicos[14].

Contar elementos únicos no es una simple agregación de todos los elementos, sino que se trata de medir la cardinalidad de un conjunto. En una aproximación simple necesitamos mantener una estructura de datos en la que para cada elemento tenemos asociado un contador. Esta aproximación no es viable para grandes cardinalidades, por lo que se utilizan aproximaciones.

HyperLogLog[15] es un algoritmo aproximado capaz de estimar cardinalidades de elementos superiores a 10^9 con un margen de error del 2 % minimizando el uso de memoria.

Spark incluye una implementación de HyperLogLog para contar cardinalidades, lo mismo que la base de datos clave-valor en memoria Redis. Por otra parte, existen múltiples ejemplos menos triviales de uso de Apache Storm implementando HyperLogLog[16].

[14]Wikipedia: "Count-distinct problem" (en inglés). [En línea] URL disponible en: https://en.wikipedia.org/wiki/Count-distinct_problem

[15]Wikipedia: "HyperLogLog" (en inglés). [En línea] URL disponible en: https://en.wikipedia.org/wiki/HyperLogLog

[16]Blog Mawazo: "Counting Unique Mobile APP Users with HyperLogLog". [En línea] URL disponible en: https://pkghosh.wordpress.com/2014/11/16/counting-unique-mobile-app-users-with-hyperloglog/

7.2. Elementos más frecuentes – *Heavy hitters*

La búsqueda de elementos muy frecuentes o *heavy hitters* es un problema recurrente en publicidad en internet (*clickers* frecuentes, *ads* populares, *spam clickers*), noticias (temas populares, tendencias), buscadores (búsquedas frecuentes) o seguridad (detección de ataques).

Existe toda una categoría de algoritmos para este problema: space-saving, lossy counting o count-min sketch.

7.3. Muestreo uniforme

En muchas ocasiones necesitamos extraer muestras de entre enormes volúmenes de datos para hacer cálculos rápidos. Un ejemplo puede ser extraer una muestra de tráfico en la red de un proveedor de telecomunicaciones dado que los datos de tráfico que se generan en su red pueden estar perfectamente en la escala de terabytes.

El problema de muestreo uniforme[17] consiste en escoger una muestra aleatoria de N posibles elementos cuando de antemano no conocemos los N posibles elementos que nos llegan en el *stream* o cuando los N elementos no nos caben en memoria.

Este problema es recurrente y existen múltiples[18] implementaciones[19] del algoritmo.

[17]Wikipedia: "Reservoir sampling" (en inglés). [En línea] URL disponible en: https://en.wikipedia.org/wiki/Reservoir_sampling

[18]Greg Grothaus: "Reservoir Sampling: Sampling from a stream of elements". [En línea] URL disponible en: https://gregable.com/2007/10/reservoir-sampling.html

[19]Blog Balls & Bins, a mix of random thoughts: "Distributed/Parallel Reservoir Sampling". [En línea] URL disponible en: https://ballsandbins.wordpress.com/2014/04/13/distributedparallel-reservoir-sampling/

VIII. Resumen

En el contexto del procesamiento de grandes volúmenes de datos, denominamos *streaming* a la forma de procesar los datos en tránsito. Se suele contraponer al procesamiento *batch,* en el que se procesan datos previamente persistidos.

El punto de partida arquitectónico es el patrón o arquitectura Kappa. Para implementar Kappa necesitamos canalizar el *stream* de datos, por ejemplo mediante Apache Kafka, y procesar los datos en tránsito mediante Spark Streaming y similares.

Las técnicas de procesamiento de *streams* incluyen algoritmos que trabajan sobre aproximaciones y mantienen una precisión muy alta.

Recursos

Enlaces de Interés

- **Jay Kreps: "Questioning the Lambda Architecture".**
  https://www.oreilly.com/ideas/questioning-the-lambda-architecture

- **Github: "A collection of links for streaming algorithms and data structures".**
  https://gist.github.com/debasishg/8172796

- **Google Cloud Platform: "How WePay uses stream analytics for real-time fraud detection using GCP and Apache Kafka".**
  https://cloud.google.com/blog/big-data/2017/08/how-wepay-uses-stream-analytics-for-real-time-fraud-detection-using-gcp-and-apache-kafka

- **Databricks: "Introducing streaming k-means in Apache Spark 1.2".**
  https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html

- **O'Reilly: "Applying the Kappa architecture in the telco industry".**
  https://www.oreilly.com/ideas/applying-the-kappa-architecture-in-the-telco-industry

- **Srinath Perera: "Short Introduction to Realtime Analytics with Big Data: What, Why, How?".**
  http://srinathsview.blogspot.com.es/2014/12/realtime-analytics-with-big-data-what.html

- **Ericsson Research Blog: "Data processing architectures - Lambda and Kappa examples".**
  https://www.ericsson.com/research-blog/data-processing-architectures-lambda-and-kappa-examples/

Glosario.

- **Arquitectura software**: Es la forma de organizar un sistema informático complejo entendido como un conjunto de programas que trabajan de manera coordinada para resolver un cálculo o problema. Las arquitecturas software suelen modelarse en capas que permiten abstraer diferentes niveles de complejidad.

- **Conector**: Pieza o programa software especializado para integración de programas que deben comunicarse entre sí. El conector es el encargado de mantener el canal de comunicación y traducir formatos de datos.

- **Machine learning**: (Aprendizaje automático) Es la disciplina de la inteligencia artificial que se ocupa de desarrollar técnicas que permitan a las computadoras "aprender".

- **Message passing layer.**: Capa de una arquitectura software especializada en solucionar la comunicación entre programas.

- **Serving layer**: Capa de una arquitectura software especializada en el ofrecimiento de información o datos a terceros programas. En el contexto del Big Data, la serving layer es la que almacena resultados agregados para su consulta por parte de terceros programas.

- **Stream**: Flujo continuo de eventos o datos.

- **Stream processing layer**: Capa de una arquitectura software especializada en procesamiento de mensajes que llegan de manera continua en forma de streams.