Métodos y algoritmos para el procesado de datos en *streaming*

PID_00276695

Francesc Julbe López

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Francesc Julbe López

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Albert Solé

Primera edición: marzo 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Francesc Julbe López
Producción: FUOC

Índice

Introducción

Más allá de los aspectos arquitecturales para el procesado en *streaming*, que hemos visto en módulos anteriores, en este módulo nos centraremos en la capa algorítmica y metodológica necesaria para tratar dichos datos, o lo que podríamos llamar *stream analytics*.

Vamos a iniciar el módulo caracterizando, desde un punto de vista temporal, un *stream* de datos y cómo el **tiempo** se convierte en el factor clave para su tratamiento, lo que hace imprescindible la introducción del concepto de *ventana* (*window*) como unidad mínima de volumen de datos de trabajo. Así, también presentamos los mecanismos necesarios para entender cómo el tiempo perfila el comportamiento del sistema (*watermarks*) y cómo el sistema debe responder ante ello (*windowing* y *triggers*).

Asimismo, la explotación de un *stream* de datos pasa claramente por permitir que se le formulen consultas. Dichas consultas pueden ser a medida (sobre algún tipo de valor o métrica puntual del *stream*) o continuas, típicas de sistemas de monitoreo, con la finalidad de lanzar alarmas o proveer de estadísticas varias casi en tiempo real. En esta línea, presentaremos el comportamiento dual entre el *stream* de datos y la tabla que nos permitirá expresar consultas complejas sobre *streams* de datos e incluso cruzar (combinar información) *streams* de forma similar a como lo hacemos con bases de datos relacionales estándar. Con tal de facilitar los mecanismos de consulta, los principales *frameworks* del mercado han incorporado funcionalidades para formular este tipo de consultas mediante un lenguaje similar a SQL.

Realizar análisis sobre datos en *streaming* tiene algunas particularidades que deben considerarse, debido a la naturaleza de los datos. Si uno de los criterios principales de la capa *batch* es la escalabilidad horizontal y cómo esta característica tiene impacto en la algorítmica asociada (distribución y paralelismo), el procesado de datos en *streaming* también tiene unas características específicas y distintas en cuanto a las técnicas y algoritmos.

En muchos sistemas es necesario comprender las características de un *stream* de datos a medida que va recibiéndose sin la posibilidad de procesar todos y cada uno de sus valores. Así, introduciremos las técnicas de síntesis, que permiten realizar estimaciones sobre algunas métricas útiles para caracterizar el contenido de un *stream*.

1. Características de un *stream* de datos

Un *stream* de datos fluye de forma continua, pero, como ya hemos visto en módulos anteriores, no se acostumbra a almacenar para su procesado, sino que se procesa a medida que se reciben los datos. Por lo tanto, con el fin de entender los algoritmos necesarios para tratar un *stream* de datos, deben considerarse las siguientes características y aspectos limitadores:

- **Solo una pasada**: por la propia naturaleza de los datos en *streaming* (cantidad y frecuencia de generación), debemos pensar que estos no deben almacenarse ni reprocesarse[1] (aunque por la arquitectura que se despliegue pueda trabajarse con *microbatches* y permitirse el reprocesado, pero de forma limitada). Así, sea cual sea el procesado, solo podrá realizarse en una sola pasada de datos. Nótese que aquí nos referimos a la parte del sistema orientada al procesado en *streaming*. Como hemos visto en las arquitecturas analizadas en módulos anteriores, almacenar los datos es una parte fundamental, pero una vez almacenados se procesan con técnicas *batch*.

[(1)]Algunas herramientas permiten el reprocesado por el almacenaje de la información en *streaming*.

- ***Concept drift* (cambio de concepto)**: no se dispone de un conjunto de datos estáticos históricos, sino que los datos y sus características pueden cambiar a medida que llegan, de modo que algunos análisis predictivos pueden verse afectados porque los datos entrantes son muy cambiantes. Llegados a este punto, debemos cambiar de concepto para empezar a entender que la información que proveen los datos variará en función del tiempo. Esto implica un proceso continuo y en línea de la información. Todas las técnicas que veremos están orientadas en esta dirección.

Bibliografía recomendada

Para más información sobre la disciplina del *stream data management*, podéis consultar: **N. Chaudhry; K. Shaw; M. Abdelguerfi** (eds.) (2005). *Stream Data Management*. Nueva York: Springer US.

- **Limitación del flujo de datos**: es habitual no tener el control sobre el emisor de los datos, por lo que es posible que se produzcan situaciones en las que haya picos de datos. Nuestro sistema de procesado de datos debe ser capaz de gestionar estas situaciónes, descartando registros que no pueda procesar. Hay dos estrategias que nos permiten llevar a cabo este descarte: la aleatoria y la semántica.

- **Limitaciones de dominio**: aunque las anteriores limitaciones son habituales en cualquier sistema de *streaming*, hay limitaciones asociadas a cada problema y flujo de datos concreto. Por ejemplo, si queremos almacenar un *stream* con millones de registros por minuto, hay limitaciones obvias que deben solucionarse a medida que se presentan. En este caso, la capacidad para poder ingerir estos datos y la velocidad en su procesamiento pasan a ser críticas. Así, es habitual desarrollar mecanismos para elaborar sumarios o sinopsis sobre *streams* de datos, antes de llevar a cabo análisis más profundos que seguramente requerirán de métodos más complejos (y lentos). Existen técnicas para el desarrollo de sinopsis (o síntesis) para

streams de datos en la disciplina llamada *stream data management*. Dichas técnicas se analizarán en apartados posteriores.

2. Técnicas fundamentales para el análisis de datos en flujo

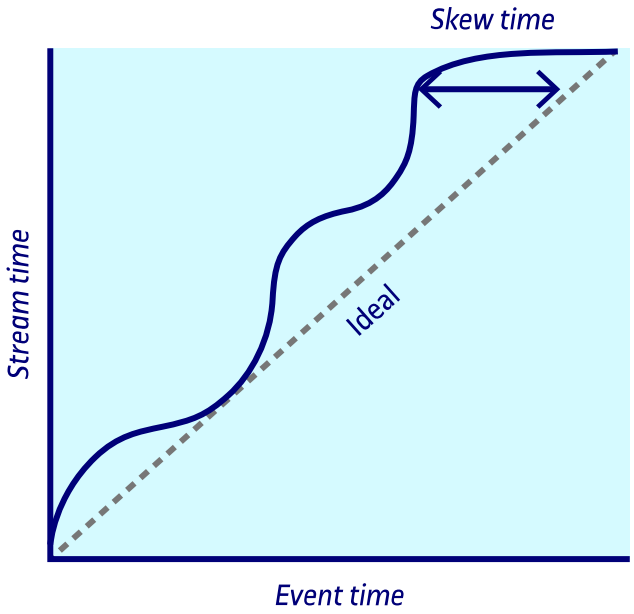
En este apartado veremos los diferentes métodos para el tratamiento de datos en *streaming*, que son genéricos y pueden aplicarse a la mayor parte de problemas. Nos centraremos en tres de ellos, que con seguridad abarcan prácticamente todos los casos de uso que podamos considerar: ventanas de tiempo, sesiones y *triggers*. Cada uno de estos métodos tiene sus características definitorias y, como veremos, son claramente diferenciables, aunque muchas veces sean intercambiables en la práctica.

Prácticamente todos se basan en la evolución del flujo en función del tiempo, ya sea considerando el tiempo de manera explícita o considerando la cantidad de eventos que van llegando. Así, es fundamental empezar definiendo varias medidas de importancia que dependen del tiempo.

2.1. Definición del tiempo para el análisis de datos en flujo

Seguramente, el factor principal que definirá y modelará todos los métodos para el procesado de datos en *streaming* es cómo gestionan el paso del tiempo. Como ya parece evidente, el procesado de datos recibidos en forma de flujo (forma continua) nos obliga a llevar a cabo un análisis temporal de sus características y considerar diferentes visiones del tiempo según la perspectiva que tomemos. En un sistema en *streaming*, el momento en el que se genera el dato, que llamaremos ***event time***, no es el mismo que el momento en el que el sistema lo consume, el ***stream time***, ya que puede haber latencias, debido a la recepción del dato desde el emisor hasta el receptor. Estas latencias pueden deberse a infinidad de razones: retrasos en la red, velocidad de preprocesamiento de los datos, uso de sistemas de mensajería o colas (como ya vimos), etc. Así, la diferencia entre ambas medidas es el llamado ***skew time***. Dependiendo de cómo esté implementado nuestro sistema y sus requisitos de negocio, este tiempo puede no ser negligible y debe considerarse.

Figura 1. Gráfico descriptivo de la relación entre *event time*, *stream time* y *skew time*



2.2. Ventanas de tiempo

Las ventanas de tiempo son el mecanismo más común mediante el cual procesaremos datos en *streaming*. En este contexto, definimos una ventana en función del tiempo mediante dos valores: el tiempo de inicio y el tiempo de finalización.

Por ejemplo, podemos definir una ventana de tiempo que empiece el 2020-1-5 T 0:00 UTC y termine el 2020-1-6 T 0:00 UTC. Esta ventana tendrá una longitud de veinticuatro horas. Como nota al lector con poca experiencia en el tratamiento de datos, cabe saber que el tratamiento de fechas y calendarios es un tema complejo en la programación y hay que ser riguroso para no cometer errores.

Una vez definida la ventana de interés, que vendrá condicionada por el problema que solucionar, los análisis y algoritmos que implementemos considerarán solo los datos que han entrado en flujo dentro del periodo definido por la ventana. Dentro de esta ventana, como veremos, los datos se procesarán de forma muy similar al procesado *batch*.

Aunque en este apartado nos hemos centrado en definir qué es una ventana de tiempo, lo habitual no es considerar una sola ventana, sino que se usan para segmentar de manera continua el flujo de datos en función del tiempo. En este módulo nos centraremos en los dos tipos de ventana más comunes: *sliding window* y *tumbling window*.

2.3. Ventana deslizante (*sliding window*)

La *sliding window* se basa en dos parámetros básicos: longitud y desplazamiento (ver figura 2).

1) La longitud de la ventana es la cantidad de información que vamos a considerar. Es decir, va a dar contexto a nuestro análisis. Vamos a definir esta longitud como la diferencia de fecha entre el inicio de la ventana y el final.

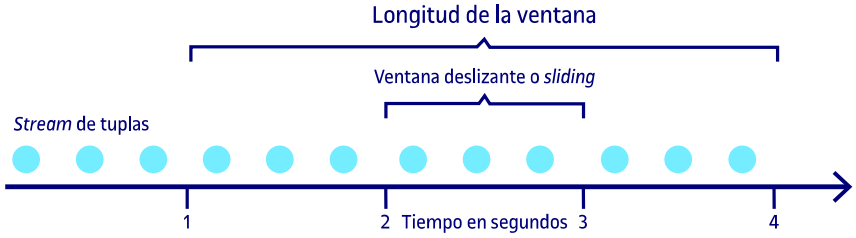
En el ejemplo anterior, esta longitud correspondía a veinticuatro horas.

Debemos considerar que las ventanas largas ofrecen mucho contexto al análisis, pero enlentecen nuestro procesamiento de las estadísticas ya que los datos a procesar son mayores. Las ventanas cortas ofrecen ventajas en este aspecto, pero al no tener contexto, dificultan la validez de las estadísticas (son muy susceptibles al ruido que puede haber en el flujo de datos generado). Escoger un tamaño de ventana ideal no es fácil y debe considerarse tanto el problema en cuestión, el ruido en los datos a procesar y la inmediatez requerida.

2) El desplazamiento se define como el valor (en unidades de tiempo) que nos indica cada cuánto (en unidades de tiempo) se desplaza la ventana para considerar el siguiente conjunto de datos (de la longitud escogida en el punto anterior). Vamos a verlo con el ejemplo de la figura 2.

La figura 2 muestra un sistema configurado para procesar datos con una *sliding window* de longitud igual a 3 y desplazamiento igual a 1. Es decir, nuestra ventana siempre contendrá todos los elementos generados entre el tiempo t y $t + 3$ segundos, por lo que vamos a llamarla $v_t = [t, t + 3]$. El desplazamiento, d, indica cómo a partir de la ventana v_t vamos a generar la siguiente, en este caso desplazando tanto el inicio de la ventana como el final en d unidades de tiempo. Es decir, $v_{t+1} = [t + d, t + 3 + d]$ y $v_{t+2} = [t + 2d, t + 3 + 2d]$ y así consecutivamente. Por ejemplo, $v_0 = [0,3]$, $v_1 = [1,4]$, $v_2 = [2,5]$, y así para todo el *stream* de datos. Debemos notar que varias ventanas comparten subconjuntos de datos, en cuyo caso los mismos datos se procesarán en tres ventanas contiguas.

Figura 2. Ventana deslizante de un segundo sobre una ventana de longitud tres segundos



De los *frameworks* ya analizados en otros módulos, Apache Spark es compatible con la *sliding window*, lo que es lógico si se tiene en cuenta su funcionamiento de base mediante *micro-batches*. Vemos que el *micro-batch*, como comentábamos anteriormente, se procesará de forma muy similar a un conjunto de datos *batch* fijos. Sin embargo, dicha compatibilidad se basa en el *stream time*, aunque puede programarse para trabajar sobre el *event time* si la latencia asociada al *skew time* es significativa y relevante para nuestro sistema.

Apache Storm, la siguiente tecnología que analizaremos, trabaja con registros individuales, generando un DAG de tareas sobre ellas a escala individual, lo que no es compatible de forma nativa con la *sliding window*. Una vez más, sin embargo, es posible dotar a un sistema Storm de compatibilidad con una *sliding window* de forma programática.

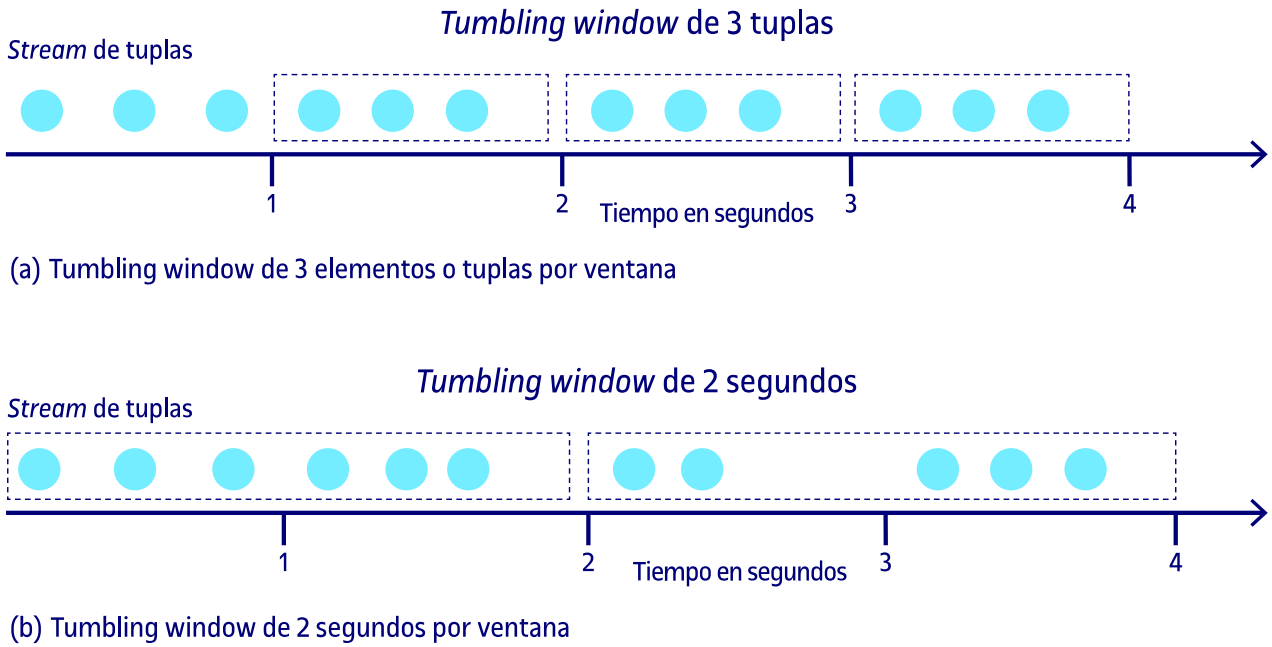
2.4. Ventana de saltos de tamaño constante (*fixed size tumbling window*)

La política de la *tumbling window* ofrece una aproximación diferente a las *sliding windows*. En este caso, las ventanas no se solapan nunca y, por lo tanto, cada subconjunto de datos dentro de la ventana solo se procesa una vez. Podemos diferenciar entre dos tipos, según cómo segmentemos el flujo de datos:

1) Ventana basada en registros. El procesado de los datos dentro de la ventana se ejecuta una vez esta se llena con un número concreto de elementos. La figura 3(a) muestra este tipo de ventana. En la figura, este valor corresponde a 3, es decir, cada ventana estará compuesta por tres elementos y no se va a procesar ninguna otra hasta que no esté llena, independientemente de cuánto tarden estos elementos en ser generados.

2) Ventana basada en rangos de tiempo. La implementación del procesado de registros por intervalos de tiempo concretos es equivalente al intervalo y longitud definidos para la *sliding window*. De forma equivalente a la *sliding window*, esta longitud es independiente del número de elementos a procesar en dicha ventana. La figura 3(b) muestra este tipo de ventana, y vemos, en este caso, que la *tumbling window* es equivalente a una *sliding window* en que la longitud y el desplazamiento son iguales.

Figura 3. Variantes del método *tumbling window*



Un ejemplo de aplicación sería el cálculo de la media móvil correspondiente a un *stream* de datos que se va recibiendo. Podríamos considerar que una red de sensores nos envía la medida de temperatura de una serie de componentes. Si queremos tener la media móvil de la temperatura de los últimos treinta segundos, independientemente del número de lecturas que dispongamos, vamos a utilizar una *tumbling window* de treinta segundos, y entonces se realizará el cálculo, como muestra la figura 3(a).

La *tumbling window* basada en el número de registros haría la media de las últimas treinta mediciones recibidas, por ejemplo, independientemente del momento en que se reciban, como podemos ver en la figura 3(b).

Apache Spark y Apache Storm no dan soporte nativo a esta técnica y requieren de una implementación para llevarla a cabo. Con Apache Storm puede resultar más sencillo debido al mecanismo de procesado de registros individuales que ofrece.

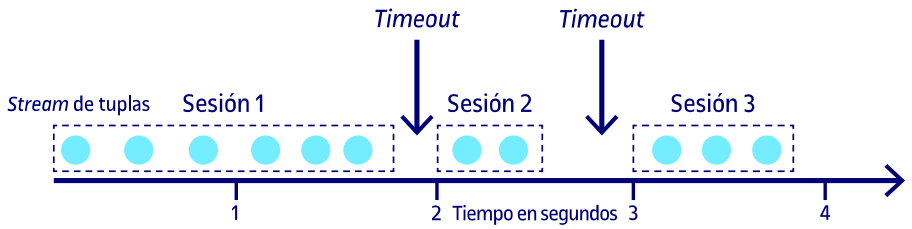
2.5. Sesiones

Otra tipología de segmentación de los *streams* de datos se hace mediante sesiones. Las sesiones se definen mediante periodos de interrupción del *stream* de datos, en los que no llegan datos porque la fuente no los genera.

Por ejemplo, podríamos considerar un sensor que cuenta coches que circulan por una carretera. Habrá momentos en los que no pasen coches y, por lo tanto, durante estos intervalos el sensor no generará datos.

Cuando el flujo entrante de registros se interrumpe, pasado cierto umbral de tiempo (*timeout*) definido por nosotros, podemos asumir que se ha cerrado una **sesión**, con lo que se genera una ventana. Vemos un ejemplo de ello en la figura 4. Este tipo de ventanas está muy asociado a comportamientos no *deterministas*, a veces asociados al comportamiento humano. No es posible conocer a priori ni el tamaño de la sesión, ni cuándo se va a iniciar la siguiente, ni cuál va a ser su duración. Como podemos ver, son ventanas bastante inciertas. Una vez definida la ventana, el procesado es equivalente al de los tipos de ventana anteriores; los datos dentro de cada una de ellas se procesan de forma independiente.

Figura 4. *Streams* de datos con ventanas por sesiones



Una vez alcanzado el *timeout* sin recibir datos, finaliza la sesión actual y empieza la siguiente.

2.6. *Triggers*

Las ventanas nos permiten seleccionar de manera iterativa fracciones de datos por procesar, aunque estos se procesan de forma muy parecida a los procesos *batch*, pero en *micro-batches*. Los *triggers* ofrecen un mecanismo mucho más próximo al procesado en *streaming* y permiten actualizar estadísticas ya calculadas a medida que el *stream* cambia, como cuando llegan eventos con retraso.

Hay una gran variedad de *triggers*, pero todos están enfocados en indicar en qué momento es necesario actualizar el cálculo requerido, a medida que van llegando los eventos por el *stream* de datos. Aun así, conceptualmente podemos clasificarlos en dos tipos, y sus aplicaciones prácticas casi siempre se reducen usando uno de los dos o una combinación de ambos.

2.6.1. *Triggers* recurrentes de actualización

Éstos indican cuándo procesar la información para actualizar las estadísticas a medida que los contenidos de los *streams* evolucionan. Estas actualizaciones pueden hacerse para cada nuevo registro del *stream*, cada varios registros o tras un periodo de tiempo.

Ejemplo de actualización por registro

Ver la figura en el siguiente enlace: <http://streamingsystems.net/fig/2-6>. En esta figura se describe un sistema con una ventana de tiempo de dos segundos en la cual agregamos los datos a medida que van entrando. El *trigger* está configurado para actualizar los datos con cada nuevo registro dentro de la ventana. Mediante la API de Apache Beam (<https://beam.apache.org/>) podríamos programar dicho *trigger* como sigue:

```
Window.into(FixedWindows.of(TWO_MINUTES)).triggering(Repeatedly(AfterCount(1))
```

Es interesante ver que el *trigger* se dispara cada vez que llega un evento y actualiza los datos de cada ventana con la información retrasada. Vemos, por ejemplo, que el evento en el minuto 12:01:30 llega tarde, en el minuto 12:08:30. Es entonces cuando se procesa y se incluye en la ventana de dos minutos que va desde 12:00 a 12:02.

Los requerimientos del sistema dictan qué tipo de *trigger* se utilizará y cada cuándo se va a disparar, una decisión que suele estar sujeta al equilibrio entre latencia y coste. Éstos *triggers* son los más habituales en sistemas streaming, ya que son sencillos de implementar y comprender. En la parte práctica del módulo vamos a ver cómo se implementan los *triggers* con Spark Structured Streaming.

Como puede deducirse, este tipo de *trigger* es muy útil para sistemas que requieran datos lo más actualizados posible. Sin embargo, en sistemas como Spark Streaming (nótese la diferencia con Spark Structured Streaming, del cual hablamos más tarde), donde la unidad de procesado son *micro-batches*, es necesario utilizar otro tipo de *trigger* que pueda tratar con datos que lleguen con cierto retraso pero que no requieran tanto coste computacional. Aquí podemos considerar dos tipos de *triggers* con retraso constante (`aligned delay`) y no uniforme (`unaligned delay`) y, como veremos en la parte práctica, la forma de instanciar este tipo de *triggers* también es sencilla. Veamos un par de ejemplos para *triggers* con retraso constante y *triggers* con retraso no uniforme:

```
Window.into(FixedWindows.of(TWO_MINUTES)).triggering(Repeatedly(AlignedDelay(TWO_MINUTES)))


Window.into(FixedWindows.of(TWO_MINUTES)).triggering(Repeatedly(UnalignedDelay(TWO_MINUTES)))
```

A continuación, se muestran dos ejemplos del funcionamiento de este tipo de *triggers*.

Ejemplo de *trigger* con retraso constante

Véase figura: <http://streamingsystems.net/fig/2-7>. En este ejemplo tenemos un sistema configurado con un *trigger* que dispara cada dos minutos. Vemos como, de hecho, cada dos minutos se van actualizando las estadísticas por ventana para considerar los elementos con retraso. El *trigger* actualiza las estadísticas de cada ventana siempre que se haya recibido al menos un evento. Además, vemos como la tercera ventana temporal no se actualiza ya que no hay información nueva para incorporar.

El mayor problema del *trigger* con retraso constante es que puede implicar picos de procesado, ya que se ejecutan todos los *triggers* de forma sincrónica en todo el sistema. Si tenemos frecuentes períodos de interrupción en la llegada de datos (debido a la latencia de la red o a problemas en la configuración del sistema), seguramente, cuando se restablezca la llegada, van a entrar gran cantidad de datos, muchos con retraso, y se van a tener que actualizar las estadísticas para muchas ventanas a la vez.

Veamos ahora cómo opera el *trigger* con retraso no uniforme.

Ejemplo de *trigger* con retraso no uniforme

Véase figura: <http://streamingsystems.net/fig/2-8>. En este ejemplo tenemos un sistema configurado con un *trigger* que dispara cada dos minutos en modo de retraso no uniforme. En este caso, el *trigger* no dispara en un punto fijo en el tiempo; vemos que empieza a contar los dos minutos individualmente por ventana una vez se ha recibido un elemento. De la misma forma que con el retraso constante, si no hay nuevos elementos, el *trigger* no empieza a contar y no se ejecuta, ya que no hay datos por actualizar.

Mediante el ejemplo podemos ver que los *triggers* no uniformes distribuyen mejor la ejecución de estos y, por lo tanto, también el uso de la CPU.

Dentro de este tipo de *triggers*, también existe la modalidad que se ejecuta bajo ciertas condiciones (`data-driven trigger`). Así, el *trigger* se activa después de que se hayan recopilado al menos N elementos dentro de la ventana. Esto permite que una ventana emita resultados parciales (antes de que se hayan acumulado todos los datos), lo que puede ser particularmente útil si se está utilizando una única ventana global o de gran tamaño.

Ejemplo

Un sistema en *streaming* ejecuta el *trigger* cada cincuenta registros recibidos. Es importante imponer condiciones de cierre o múltiples condiciones ya que, si al finalizar el *stream* no se ha cumplido la condición, los últimos elementos no se procesarán.

2.6.2. *Triggers* de completitud

Los *triggers* de completitud generan resultados acumulados para una ventana una vez se han satisfecho ciertos requerimientos, generalmente cuando todos los datos dentro de una ventana se han recibido. De forma similar al *batch processing*, se lleva a cabo esta materialización de los datos pero, en este caso, el tamaño del *batch* son los datos contenidos en una ventana. Son menos

frecuentes, pero al tener un rango de procesado más amplio, permiten tratar aspectos como registros perdidos o con retraso sin sobrecargar el sistema. En definitiva, solo se ejecuta una vez cuando se han recibido todos los datos.

En relación con los *triggers* recurrentes, los triggers de completitud permiten controlar el flujo de datos y equilibrar los diferentes factores según su uso:

- **Integridad:** requerimiento de disponer de todos los datos (o gran parte de ellos) antes de calcular su resultado.
- **Latencia:** tiempo que es aceptable esperar para disponer de todos los resultados para mostrarlos.
- **Coste:** capacidad de computación para realizar este procesado.

Aunque estos *triggers* parecen ideales, ya que en gran parte mejoran todas las características de los *triggers* recurrentes, tienen un gran problema nada despreciable: definir e implementar el concepto de *completitud*. En este punto entran las *watermarks*, que definen este concepto en función del *skew time*.

2.7. *Watermarks*

Tal y como hemos introducido anteriormente, en cualquier sistema de procesado de datos en *streaming* hay una latencia entre el tiempo en el que un evento se produce (*event time*) y el tiempo de procesado (*stream time*): el llamado *skew time*, como hemos visto en la figura 1. Además, no existen garantías de que los eventos de datos aparezcan en el mismo orden en el que se generaron. Así, podríamos decir que la *watermark* es la línea curva que permite conocer la relación entre el *stream time* (el tiempo de procesado real del evento en el sistema) y el *event time*. Nos permite definir (o estimar) el retraso con el que llegarán los eventos retardados y, por ende, cuándo ejecutar los *triggers* de completitud.

En sistemas en los que se conoce perfectamente el comportamiento de los datos de entrada podemos caracterizar perfectamente la dependencia entre el *stream time* y el *event time*, lo que nos daría una *watermark* perfecta. En este caso, podemos imaginar la *watermark* como una función analítica entre ambos valores.

Sin embargo, hay muchos escenarios en los que la relación entre ambos tiempos no puede determinarse de forma perfecta, ni es práctico tratar de hacerlo, debido a los elevados costes computacionales que esto puede suponer y a la cantidad de factores que intervienen en el retraso (retrasos de red, carga de los sistemas que generan los datos y los que los procesan, retraso en los sistemas de colas que almacenan los mensajes, etc.). En este caso se recurre a las llamadas *watermarks* heurísticas. Así, se usa toda la información disponible en el sistema para hacer estimaciones del progreso de llegada de los datos para estimar la relación entre el *event time* y el *stream time*. Las *watermarks* heurísticas tienen dos problemas básicos:

- Pueden procesar los datos con demasiado retraso debido a la inexactitud de la relación entre ambos tiempos.
- Pueden perderse algunos paquetes, ya que llegan con un poco más de retraso que el estimado.

Así, la *watermark* es la noción del sistema de cuánto se puede esperar a que todos los datos en una determinada ventana hayan llegado al sistema de procesado. Sin embargo, cuando la *watermark* progresa más allá del final de una ventana, cualquier registro adicional que llegue con una watermark de tiempo en esa ventana se considera información retardada. Veámoslo con un ejemplo.

Ejemplo de *watermark* heurística

Supongamos que tenemos un sistema con una ventana de procesado de cinco minutos y una watermark que supone aproximadamente treinta segundos de tiempo de retraso entre el *event time* y el *stream time*. Así, el sistema va a materializar la ventana 0:00 a 5:00 en el tiempo 5:30. Si un registro de datos llega a las 5:34, entenderíamos que dicho registro ha llegado con retraso y este nunca llegaría a procesarse. La *watermark* nos indicaba que los paquetes incurrirán un retraso de como mucho treinta segundos y el nuevo paquete ha tardado 34 segundos, cuatro segundos después de cuando realmente se debería haber recibido. Así, una *watermark* perfecta no hubiera permitido dicho comportamiento.

Ejemplo de la diferencia entre la *watermark* heurística y la perfecta

Veamos el ejemplo en <http://streamingsystems.net/fig/2-10>. En este caso, podemos ver que la *watermark* perfecta espera al evento que llega en el segundo 12:01:30 para procesar la primera ventana. A causa de este retraso, vemos cómo el procesado de las ventanas siguientes debe esperar, en consecuencia retrasando, en este caso, la actualización de la información. Vemos como la *watermark* heurística no tiene este problema pero, en consecuencia, puede perder paquetes por una mala estimación del *skew time*.

Este ejemplo nos indica uno de sus problemas: la *watermark* puede asumir que ha procesado todos los datos demasiado rápido, pero en realidad se ha dejado datos por calcular. De forma similar, puede ocurrir que una *watermark* dé indicaciones incorrectas en cuanto a retraso y sea demasiado lenta, es decir, que haya que esperar demasiado tiempo una vez completado el procesado. De hecho, una *watermark* induce retraso por su propia naturaleza, ya que trata de modelar de la mejor manera posible, introduciendo también una desactualización de datos intrínseca. Una *watermark* conservadora actualiza los datos más lentamente.

Ejemplo de un retraso inducido por la *watermark*

Veamos otra vez el ejemplo en <http://streamingsystems.net/fig/2-10>, donde la línea de puntos indica la *watermark* ideal, donde el *skew time* es cero. Cualquier *watermark* que implique una espera debe estar por encima de esa línea.

Como posiblemente ya imaginamos, los *triggers* y las *watermarks* nos permiten combinar el procesado de los datos de modo que la *watermark* está enfocada a la completitud y el *trigger* nos permite trabajar a nivel de registro si es necesario, refinando la completitud de datos que una *watermark* debe permitir o puede haber omitido por retraso. Así, habitualmente ambas herramientas se combinan para procesar datos con retraso, añadiéndolos a una ventana anterior mediante un *trigger* más refinado si esto se produce.

En la parte práctica de este módulo vamos a ver los conceptos mencionados mediante Spark Structured Streaming.

Bibliografía recomendada

En este apartado solo hemos visto una parte muy introductoria de los tipos de *triggers* y *watermarks* y cómo se trabaja con ellos. El lector puede extender los conceptos teóricos mediante la lectura de los capítulos 2 y 3 de **T. Akidau; S. Chernyak; R. Lax** (2018). *Streaming Systems. The What, Where, When, and How of Large-Scale Data Processing*. O'Reilly Media.

3. Diferencias entre el procesado *micro-batching* y el procesado continuo

Una vez entendidas la tecnologías y técnicas para procesar datos en *streaming* en lo referente a la llegada de dichos datos en el sistema y en cómo enfrentar las particularidades intrínsecas (flujo no uniforme pero continuo e ilimitado de datos), vamos a ver cómo dichos datos son explotados habitualmente desde una perspectiva de negocio. Esto nos conduce a preguntarnos cómo podemos trabajar con estos datos.

Actualmente, las herramientas principales en el mercado del procesado en *streaming* pueden clasificarse en dos tipologías significativamente distintas: el *micro-batching* y el procesado continuo. Aunque pueden parecer muy similares, ya que ambas usan el concepto de *ventana*, su mayor diferencia reside en la interpretación del flujo de datos entrante y sus operaciones.

En el *micro-batching*, la ventana es una parte fundamental del procesado de datos y se usa para definir el conjunto de datos que se va a procesar. Los datos de cada ventana se van a procesar de forma equivalente a un conjunto de datos estáticos, y solo se procesan en función del *stream time*. Una vez procesada la ventana, los datos se descartan y se prosigue con la siguiente. Cabe decir que la mayor parte de herramientas que usan *micro-batching* habilitan maneras de compartir datos entre ventanas (mediante herramientas parecidas a variables globales) pero la semántica de su uso debe ser propia del problema a solucionar. Además, el procesado en *micro-batching* corresponde a tecnología clásica y, en algunos aspectos, anticuada. En la asignatura, dentro de esta tipología, vamos a trabajar con Apache Spark Streaming.

En el caso del procesado en continuo, como ya se ha comentado, el flujo de datos es interpretado como un continuo y estos se analizan a medida que van llegando, sin necesidad de esperar a que se complete ninguna ventana para actualizar las estadísticas necesarias o hacer el cálculo correspondiente. Aunque es configurable en la mayor parte de entornos, podemos pensar que puede llegar a incorporarse en los cálculos pertinentes a la resolución del dato entrante. Aun así, el concepto de *ventana* sigue existiendo, pero se usa principalmente para facilitar las agregaciones y los cálculos pertinentes en función del tiempo. En este caso, las ventanas no forman parte intrínsecamente del método de procesado. Dentro de esta tipología vamos a trabajar con el *framework* que ofrece Apache Structured Streaming.

Como hemos mencionado, el procesado mediante *micro-batching* es muy similar al procesado *batch* y, aparte del concepto de *ventana*, no conlleva mayor cambio de paradigma. Sin embargo, no es así con las herramientas de procesado continuo. A continuación, veremos su funcionamiento más básico, y en la parte práctica entraremos en profundidad en la materia.

4. Habilitando el procesado continuo con Apache Structured Streaming

Apache Structured Streaming y Apache Kafka ofrecen un sistema integral para la captura, el procesamiento y el almacenamiento de datos en *streaming*. Seguramente, llegados a este punto, nos acercamos más a una arquitectura Kappa que a una Lambda. Dos cuestiones importantes que se deben considerar son en qué medida Structured Streaming satisface la semántica *exactly-once* y cómo deben integrarse las herramientas disponibles a este objetivo. Si el lector está interesado en este aspecto, se aconseja buscar información mediante Google, ya que hay diversas opciones para hacerlo y seguramente entraríamos en detalles técnicos demasiado extensos para describir en este módulo. En este apartado nos vamos a centrar en analizar cómo Apache Structured Streaming habilita el cambio de paradigma citado mediante la dualidad *stream*-tabla y qué operaciones pueden surgir de este cambio de paradigma.

Enlace recomendado

Stream-Stream Joins by Confluent: <https://www.youtube.com/watch?v=51yLu5FnPYo>.

4.1. Dualidad *stream*-tabla

Cuando tenemos información estructurada, en muchas aplicaciones es necesario (o aconsejable) convertir *streams* a registros de tablas (similares a las SQL) para que puedan proporcionar información actualizada. Así, si esta información se materializa en una base de datos, se permite la interacción e integración con gran parte de sistemas de explotación. Y, a su vez, una tabla permite reconstruir un *stream* de datos a partir de cada uno de sus registros.

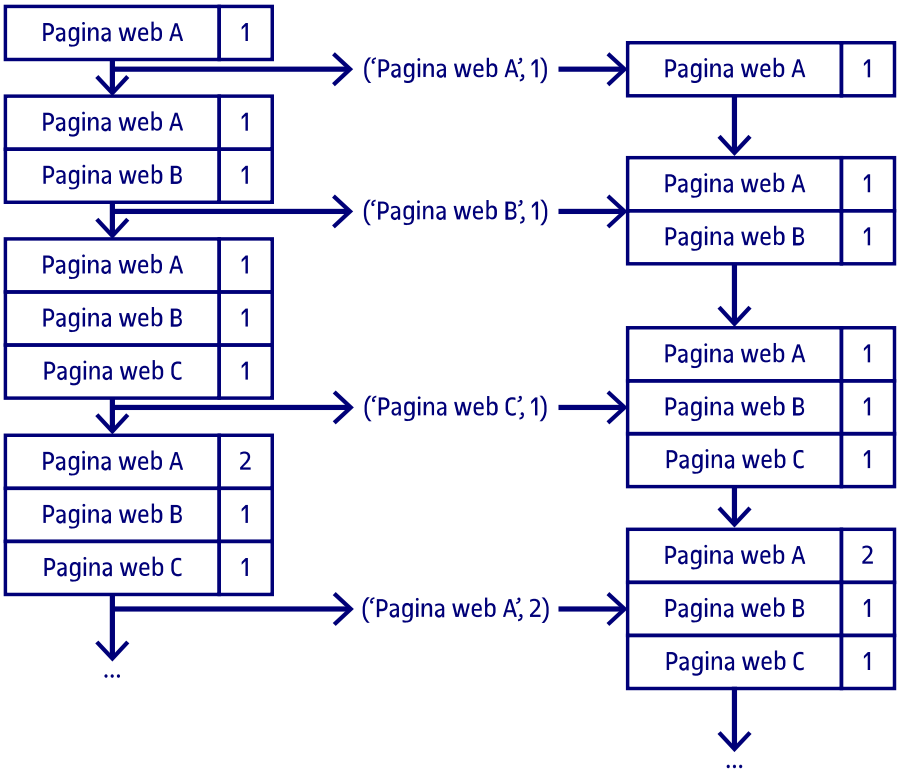
1) *Stream* como tabla. Un *stream* se puede considerar un registro de cambios de una tabla o un *changelog*, donde cada registro de datos en la secuencia captura un cambio de estado de la tabla. Por lo tanto, cuando tenemos datos estructurados, una secuencia es una tabla disfrazada y se puede convertir fácilmente en una tabla real al reproducir el registro de cambios de principio a fin para reconstruirla. Del mismo modo, la agregación de registros de datos en una secuencia devolverá una tabla.

Ejemplo

Podríamos calcular el número total de páginas vistas por usuario a partir de una secuencia de entrada de eventos de páginas vistas y el resultado sería una tabla. La clave de la tabla sería el usuario, y el valor, el recuento de páginas vistas correspondiente.

2) *Tabla-stream*. Una tabla puede considerarse como una captura del último valor para cada clave en un *stream* y se puede convertir en un *stream* iterando sobre cada entrada de valor clave en la tabla.

Figura 5. Ejemplo de dualidad *stream*-tabla



Conversión de vistas a una lista de páginas web de un formato tabular a *stream* y viceversa.

La figura 5 representa una tabla con información de las visitas a un conjunto de páginas web (A, B, C…). Cada cambio en la tabla se convierte en una nueva tupla (el registro con <key, value> que se ve en la columna central), con el valor instantáneo de las visitas para dicha página. Cada cambio puede generar una nueva tupla que, a su vez, podría actualizar una tabla con el cambio de dicho registro (la tabla que se va actualizando a la izquierda). De ahí que la interpretación de la dualidad sea mediante la caracterización de los *streams* como *changelogs* de la tabla (cambios en la tabla).

Cuando tenemos datos estructurados, esta dualidad aporta muchos beneficios, ya que permite trabajar con lo mejor de ambos mundos, a partir de un stream de datos podemos construir una tabla añadiéndole nuevos registros de forma ilimitada. Sin embargo, la naturaleza cambiante del flujo de datos en *streaming* añade complejidad a la forma de trabajar con herramientas tipo SQL sobre ellos.

En la parte práctica del módulo vamos a ver a nivel operativo cómo funciona Apache Structured Streaming. A continuación, introduciremos brevemente los conceptos básicos y los tipos de operaciones que nos ofrece la tecnología. Nos vamos a centrar en las dos operaciones más características: las agregaciones y los *joins*.

4.2. Transformaciones y agregaciones

Apache Structured Streaming nos permite utilizar los ya conocidos *dataframes* y *datasets* y sus transformaciones de forma similar al uso en el procesado *batch*. Pero, además, nos permite realizar operaciones de forma continua (infinita), explotando la dualidad *stream*-tabla, sobre una *dataframe* en la que se añaden o modifican registros de forma (más o menos) instantánea.

Como ya hemos visto, los datos en *streaming* se procesan por ventanas de tiempo (*sliding*, *tumbling* o por sesiones) y este concepto puede extenderse casi de manera trivial usando la dualidad *stream*-tabla. Si pensamos cómo realizaríamos una agregación por tiempo en SQL, seguramente lo primero que se nos ocurre es hacer un «*group by*» por marca temporal. Sin ir más lejos, Structured Streaming ofrece una operación equivalente. Así, el *windowing* en el *structured streaming* es equivalente a un «*group by*», esto es, una operación solo en al ámbito de la ventana, como por ejemplo las agregaciones por ventanas de tiempo. Podemos ver el siguiente ejemplo:

Ejemplo de agregación para contar eventos cada hora y cinco minutos

```
inputDF.groupBy($"action", window($"time", "1 hour", "5 minutes")).count()
```

Vemos que realizamos una agrupación por ventana temporal, y para cada una de ellas hacemos un *count* de la actividad observada.

4.3. *Joins*

De forma equivalente a la tecnología SQL, una de las operaciones más potentes es el cruce de información entre diversas fuentes de datos mediante operaciones *join*. Estas, aunque son conceptualmente equivalentes a los *joins* SQL, toman en cuenta la dimensión temporal de los datos y también su efimeridad. Veamos la flexibilidad ofrecida en estas operaciones:

1) *Join stream*-tabla. Muchas aplicaciones, en diversos casos de uso, requieren de *joins* de *streams* con múltiples bases de datos a gran escala. La dualidad ya comentada nos permite realizar dichos *joins* con muy baja latencia. Por ejemplo, en una plataforma de comercio electrónico, una transacción puede requerir el acceso y la actualización de múltiples bases de datos de cliente, productos y transacciones bancarias. Así, el *join* de *streams* con grandes tablas se produce en entornos en los cuales es necesario acceder a mucha información con muy baja latencia.

2) *Join stream-stream*. El *join* entre *streams* es menos intuitivo. Imaginemos que un *stream* va a contener información que, de algún modo, puede cruzarse con la información de otro *stream*. Si coincide la condición de *join* (*where* o *on*), el cruce nos devuelve un resultado. Sin embargo, ¿qué ocurre cuando pasa el tiempo? Los *streams* crecen de forma indefinida y el *join* crecería igual, lo que, llegado a cierto punto, sobrepasará la memoria del sistema. Así, para el *join* de *streams* debemos trabajar con ventanas de tiempo una vez más. De este

modo, la ventana va a contener registros con una marca temporal que cae dentro de dicha ventana. Asumiendo la limitación de las ventanas de tiempo para realizar *joins*, la dualidad *stream*-tabla nos permite generar dos tablas a partir de dos *streams* que podemos cruzar con la condición requerida. Aquí es donde podemos explotar todas las ventajas de SQL (la mayoría de *frameworks* permiten utilizar lenguaje tipo SQL para dicha algorítmica). Siempre y cuando dos registros pertenezcan a la misma ventana, el *join* entre ambos es posible. En caso contrario, el *join* no devuelve resultados.

Vemos que esta funcionalidad nos permite realizar operaciones (y alertas) muy complejas sobre múltiples *streams* en tiempo real y, cuando la lógica de negocio requiere de sistemas que crucen diferentes fuentes de datos en *streaming,* Structured Streaming tiende a ser la solución a aplicar.

5. Algoritmos en línea para el análisis de datos en flujo

Con todas las técnicas descritas anteriormente, ahora estamos en posición de empezar a sacar valor a las fuentes de datos en *streaming,* es decir, a sacar información relevante sobre estos datos. En este aspecto, vamos a diferenciar dos tipos de algoritmos muy claros: los que requieren de información històrica para calcular el valor dentro de la ventana actual y los que no. Empezaremos por los fáciles, los que no, y vamos a dedicar lo que queda de módulo a los otros.

Consideramos el conjunto de algoritmos con irrelevancia del tiempo como todos los algoritmos que no requieran datos anteriores a los existentes dentro de la ventana actual. Podemos encajar aquí cualquier algoritmo tipo *batch* pero enmarcado dentro de la ventana (*micro-batch*). Podríamos pensar desde ejemplos sencillos como el cálculo del número de eventos que han llegado dentro de la ventana, su media, cualquier operación que cruce estos datos con datos estáticos de nuestro negocio hasta operaciones realmente complejas que incorporan algoritmos de Machine Learning.

Por otro lado, incluimos en los algoritmos con relevancia en el tiempo todos aquellos que requieran datos o estadísticas de ventanas anteriores. Es decir, asumiendo que el algoritmo corresponde a una función f_x, el resultado del algoritmo a ventana x_t requiere información de la ventana anterior x_{t-1} y los datos entrantes D, es decir tenemos que $x_t = f_x(x_{t-1}, D)$. Vamos a decir que en este caso el algoritmo requiere de memoria 1, pero podemos encontrar algoritmos que requieran más memoria de ventanas anteriores. Por ejemplo, el algoritmo $x_t = f_x(x_{t-1}, x_{t-2}, x_{t-3}, D)$ diríamos que requiere memoria 3.

Ejemplo

Tenemos un sistema con una ventana móvil de dos segundos y que desplaza cada segundo, y queremos calcular una media móvil de los valores entrantes. La configuración escogida de ventana nos da suficiente actualización de datos, pero seguramente, al ser tan pequeña, va a mostrar muchas oscilaciones en sus medias por ventana. En este caso, es habitual extender la media a periodos más largos, por ejemplo diez segundos. Por lo tanto, la medida va a considerar información de las cinco ventanas temporales anteriores.

6. Cálculo en línea de funciones matemáticas

En general, para el análisis de datos, muchas veces nos centramos en valores estadísticos o funciones matemáticas. En este caso, cuando los datos van llegando en formato de flujo, el problema llega cuando la función (o estadística) a calcular agrupa varias ventanas temporales y no podemos almacenar (por falta de memoria) o recalcular (porque sería muy costoso) toda la información necesaria para el cálculo.

En estos casos debemos hacer un cambio de paradigma conceptual y entender la información imprescindible a transferir entre ventanas y cómo actualizar el estadístico en vez de recalcularlo considerando todo el conjunto de datos.

Para algunos estadísticos este proceso es relativamente sencillo y basta con guardar muy poca información y ejecutar unas pocas líneas de código. Por ejemplo, para calcular el mínimo o máximo que hemos observado en el stream de datos:

> Para calcular el mínimo, solo es necesario inicializar una variable *min* al máximo valor que podamos tener. Así, cada vez que se recibe un nuevo registro del *stream* de datos, se compara dicho registro con la variable *min*. Si es menor, se actualiza la variable *min* con su valor. Si repetimos esta comparación y posible actualización cada vez que llega un nuevo valor, conseguiremos almacenar en la variable *min* el valor más pequeño que hemos observado en el *stream* en cualquier momento del tiempo.

Otros valores estadísticos como la media son igualmente sencillos de calcular. En este caso, solo necesitaremos guardar dos valores: la suma total de los valores vistos en el *stream* hasta el momento, *acc*, junto con una variable auxiliar *counter* que cuente el número de registros que llevamos sumados. Es sencillo ver que podemos obtener la media del *stream S* en cualquier momento haciendo la siguiente división:

$$< S > = \frac{acc}{counter}$$

Aunque este proceso puede parecer aparentemente sencillo, no siempre es posible reducir toda la información de ventanas anteriores a un conjunto reducido de variables. En estos casos, habrá que trabajar el algoritmo como una relación de recurrencia para derivarlo a una expresión matemática que opere de forma incremental a medida que van llegando nuevos datos. Uno de los ejemplos más clásicos de este proceso es el cálculo de la desviación estándar de forma incremental.

Enlace recomendado

Para saber más sobre la relación de recurrencia, podéis consultar el siguiente enlace: <https://en.wikipedia.org/wiki/Recurrence_ relation>.

Ejemplo del método de Welford para calcular la varianza

La desviación estándar es una medida que cuantifica la dispersión de un conjunto de datos en relación con su media. Un conjunto de datos que esté agrupado en torno a un solo valor tendría una pequeña desviación estándar, mientras que un conjunto de

datos que están distribuidos por todo un dominio de datos tendría una gran desviación estándar.

Dada una muestra x_1, x_2,..., x_n, la desviación estándar se define como la raíz cuadrada de la varianza, que se calcula de la siguente manera:

$$s^2 = \frac{\sum_{i=1}^{N}\left(x_i - \overline{x}\right)^2}{N - 1}$$

donde $\overline{x}$ es la media de la muestra. La definición de la desviación estándar se puede convertir directamente en un algoritmo que calcula tanto la media como la desviación estándar, procesando dos veces los datos:

- Calcular la media en una pasada sobre los datos.
- Llevar a cabo una segunda pasada que calcule las diferencias al cuadrado respecto de la media.

Sin embargo, hacer estas dos pasadas no es lo ideal, y puede ser poco práctico en entornos de procesamiento en *streaming*, ya que requiere almacenar la mayor parte de valores de ventanas anteriores. No obstante, existen métodos para solventar estos problemas y hacer actualizaciones sucesivas de la media y desviación en un solo paso. La mayor parte de estos métodos son numéricamente inestables porque requieren la operación números de coma flotante de magnitudes muy distintas. Sin embargo, existe un método que permite actualizar la varianza de una forma muy eficiente, el método Welford. Solo con fines ilustrativos, vamos a revisar cómo se deriva y cómo opera. El objetivo no es entender el método, sino la complejidad del cambio de paradigma que puede suponer el tratamiento de datos en flujo.

El método de Welford es un método que hace una única lectura del flujo de datos y que se puede utilizar para calcular la varianza. Se puede derivar al observar las diferencias entre las sumas de las diferencias al cuadrado para las muestras N y N–1. A continuación, podéis ver cómo se calcula la derivación del método Welford para el cálculo de la varianza. Es realmente sorprendente lo simple que resulta dicha diferencia:

$$\begin{aligned}
&\left(N-1\right)S_N^2 - \left(N-2\right)S_{N-1}^2 \\
&= \sum_{i=1}^{N}\left(x_i - \bar{x}_N\right)^2 - \sum_{i=1}^{N-1}\left(x_i - \bar{x}_{N-1}\right)^2 \\
&= \left(x_N - \bar{x}_N\right)^2 + \sum_{i=1}^{N-1}\left(\left(x_i - \bar{x}_N\right)^2 - \left(x_i - \bar{x}_{N-1}\right)^2\right) \\
&= \left(x_N - \bar{x}_N\right)^2 + \sum_{i=1}^{N-1}\left(x_i - \bar{x}_N + x_i - \bar{x}_{n-1}\right)\left(\bar{x}_{N-1} - \bar{x}_N\right) \\
&= \left(x_N - \bar{x}_N\right)^2 + \left(\bar{x}_N + x_N\right)\left(\bar{x}_{N-1} - \bar{x}_N\right) \\
&= \left(x_N - \bar{x}_N\right)\left(x_N - \bar{x}_N - \bar{x}_{N-1} + \bar{x}_N\right) \\
&= \left(x_N - \bar{x}_N\right)\left(x_N - \bar{x}_{N-1}\right)
\end{aligned}$$

Esto significa que podemos calcular la varianza en una sola pasada usando el siguiente algoritmo:

Figura 6. Código para la implementación del método Welford para el cálculo de la varianza

```
variance(samples):
  M := 0
  S := 0
  for k from 1 to N:
    x := samples[k]
    oldM := M
    M := M + (x–M) / k
    S := S + (x–M) × (x–oldM)
  return S / (N–1)
```

En literatura podemos encontrar un gran abanico de comparaciones de este segundo método con su versión tradicional en dos pasadas. En todos los casos se observa que ambos métodos funcionan exactamente igual y devuelven el mismo resultado.

7. Técnicas de síntesis

Tal y como se ha comentado en el primer apartado, hay circunstancias en las que es necesario aplicar técnicas que nos permitan caracterizar y describir (de manera preliminar o final) un *stream* de datos que no puede procesarse en su totalidad, bien debido a su volumen y ratio de datos por segundo, o debido a las limitaciones de memoria o cálculo del sistema de procesado. Y es que es habitual no tener ningún control sobre el *stream* de datos entrante, ya que es un flujo continuo de registros con fluctuaciones, máximos y mínimos, con lo que disponer de técnicas que nos permitan tener descripciones aproximadas de los datos entrantes.

De hecho, como se ha comentado en módulos anteriores, es posible que la *speed layer* no proporcione respuestas exactas a algunas consultas. Recordemos que, en la arquitectura Lambda, la completitud se obtiene a través de la capa *batch*. Sin embargo, en muchas ocasiones, la inexactitud de los resultados es un problema menor derivado de la no-persistencia de dicha capa en la arquitectura Lambda. Disponer de técnicas y mecanismos que nos permitan analizar las propiedades estadísticas del *stream* de datos entrante puede ser fundamental y suficientemente representativo en muchos escenarios.

En este apartado, vamos a ver una serie de técnicas para extraer información general del flujo de datos fijando los tamaños de memoria a usar en el servidor y garantizando una serie de propiedades estadísticas sobre estas operaciones.

7.1. Muestreo aleatorio (*reservoir sampling*)

La primera técnica que vamos a ver nos va a facilitar la generación de una muestra aleatoria y representativa del *stream* de datos entrante. Recordemos que para que una muestra sobre unos datos generados de manera uniforme no tenga sesgo, debemos garantizar que cada elemento que hayamos observado en el *stream* entrante pueda ser escogido con una probabilidad $1/Q$, siendo Q el número de elementos observados. Así, si hemos visto tres posibles elementos, debemos garantizar que cada uno pueda ser escogido con probabilidad 1/3. En general, cada lenguaje de programación ofrece funcionalidades para este objetivo. Python, por ejemplo, ofrece «numpy.random.choice», que fijando el valor de reemplazo a «Falso», se puede usar a tal efecto. La mayor parte de métodos para realizar este muestreo requieren guardar una colección de todos los elementos que hemos ido observando para, en el momento de hacer el muestreo, poder consultarlos. En sistemas *streaming*, la mayor parte de las veces no es posible guardar todo el conjunto a muestrear por problemas de memoria. Es entonces cuando el *reservoir sampling* muestra su mejor cara.

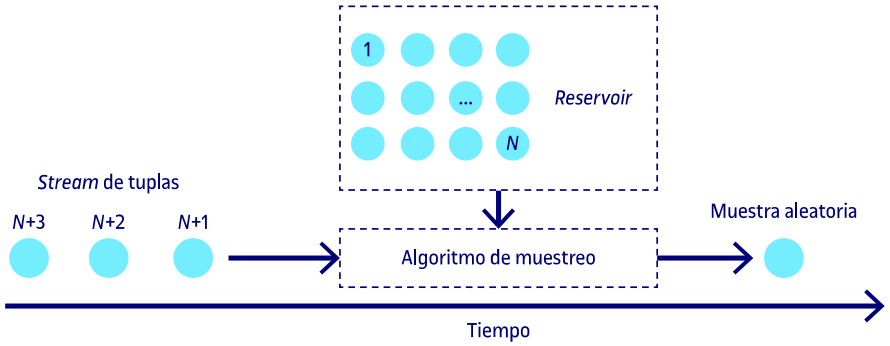
Enlaces recomendados

Podéis ampliar la información sobre el muestreo aleatorio en los siguientes enlaces: <https://www.youtube.com/watch?v=Ybra0uGEkpM>, <https://www.youtube.com/watch?v=3fg708--uT4>, <https://www.youtube.com/watch?v=Ojz_deSQ_JI>.

Para aplicar el *reservoir sampling* debemos conocer el tamaño de la muestra que queremos obtener. Vamos a asumir que es N. Es decir, queremos obtener N elementos sampleados uniformemente de nuestro *stream* de datos. Para ello, vamos a almacenar solo un *reservoir* (nuestra muestra aleatoria, con los N elementos de la muestra, que iremos actualizando a medida que los datos van entrando en nuestro sistema de *streaming*. En este caso el algoritmo garantiza que cada elemento que se ha observado en el sistema tenga una probabilidad de ser muestreado de N/Q, donde Q es, otra vez, el numero de elementos que hemos observado.

Por regla general, cuando el sistema se está iniciando, la primera muestra se genera con los N primeros elementos del *stream*. A partir del elemento $Q + 1$, se activa un algoritmo (el *sampling algorithm*) que decide si dicho elemento debe reemplazar a alguno de los ya existentes en el *reservoir* mediante unas reglas probabilísticas para garantizar la uniformidad de la muestra. En este caso, para el elemento $Q + 1$ se calcula la probabilidad de reemplazo $p = N/Q$. A partir de aquí, se genera un número aleatorio Q y se reemplaza un elemento de la muestra si $q < p$.

Figura 7. Ejemplo de *reservoir sampling*



Pasado un cierto tiempo, la muestra del *reservoir* es suficientemente representativa de nuestro *stream* de datos, se garantiza que cada elemento del *stream* sea muestreado con una probabilidad N/Q y se mantiene acotado al tamaño de datos almacenados para generar la muestra.

7.2. Contando elementos distintos

Otra información de mucha utilidad que debemos conocer de un *stream* de datos es el número de elementos distintos que hemos observado.

Este problema aparece generalmente cuando la tipología de objetos a observar es discreta, por ejemplo, números enteros o ID de usuarios, tipos de objetos, etc. Por ejemplo, cuántos usuarios están accediendo a un sistema en estos momentos. La metodología no se usa en general cuando los valores analizados corresponden a una variable continua, por ejemplo, la temperatura que toma un sensor, la presión atmosférica, el voltaje de una línea, etc. Aun así, siempre podemos discretizar la información si es necesario.

Recordemos que en este tipo de técnicas asumimos que no se puede almacenar la totalidad del *stream* de datos en memoria, así que debemos basarnos en técnicas probabilísticas (y heurísticas) para llevar a cabo dichas operaciones. En este caso, usamos el contaje de elementos distintos, esto es, la cardinalidad del flujo entrante.

Para este objetivo existen básicamente dos tipos de algoritmos:

1) Los basados en la ordenación estadística. Los algoritmos de esta clase se basan en estadísticas relativas a la ordinalidad, como los valores más pequeños que aparecen en una secuencia.

2) Los basado en patrón de bits. Actualmente, los algoritmos de patrón de bits son los más utilizados. Se basan en la observación de patrones de bits que ocurren al comienzo del valor binario de cada elemento de *stream*. Más específicamente, se centran en el número de ceros iniciales o finales de la representación binaria de un hash del elemento observado en el *stream*. Algunos de los algoritmos que se encuentran en esta categoría son LogLog, HyperLogLog e HyperLogLog++.

Función hash

Una función hash es un algoritmo matemático que transforma cualquier bloque arbitrario de datos en una nueva serie de caracteres con una longitud fija. Independientemente de la longitud de los datos de entrada, el valor hash de salida tendrá siempre la misma longitud. Algunas de las técnicas más utilizadas son SHA-1, SHA-2 y MD5.

No existen dos entradas que produzcan el mismo hash, lo que convierte dicha técnica en una potente herramienta para la criptografía.

Enlace recomendado

Podéis ampliar la información sobre HyperLogLog en el siguiente enlace: <https://www.youtube.com/watch?v=eV1haPUt0NU>.

En estos algoritmos se usan técnicas heurísticas (aproximadas) mediante las cuales se nos permite obtener una estimación del número de elementos distintos en el *stream* con una precisión bastante buena, aunque es configurable mediante parámetros. A continuación, vamos a presentar los conceptos generales del algoritmo HyperLogLog++.

Enlaces recomendados

Podéis encontrar los detalles de los algoritmos citados en:

P. Flajolet; É. Fusy; O. Gandouet; F. Meunier (2007). "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". En: *2007 Conference on Analysis of Algorithms, AofA 07*, <http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf>, y **S. Heule; M. Nunkesser; A. Hall** (2013). "HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm". En: *EDBT/ICDT'13*, <http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/40671.pdf>.

Existen implementaciones de dichos algoritmos en casi cualquier lenguaje de programación como por ejemplo: <https://pypi.org/project/hyperloglog/>, en Python, o el módulo streaming en Java de Spring: <https://www.javadoc.io/doc/com.clearspring.analytics/stream/latest/index.html>.

Para entender el algoritmo HyperLogLog++, debemos asumir que los bits de la función hash que queremos usar y que codifican nuestro elemento están más o menos distribuidos aleatoriamente. Es decir, en cada una de las posiciones

en codificación binaria (1 o 0) de nuestro hash, la probabilidad de encontrar un 1 o un 0 son prácticamente idénticas e iguales a 0.5. De esta forma, la probabilidad de encontrar un hash con valor cero en el último bit es 0.5, la de encontrar dos ceros $0.5 \cdot 0.5$ y así, la de encontrar N ceros al final del hash es de $p = 1/(2^N)$. Por lo tanto, la pregunta fundamental es, si he observado (en el *stream* de datos) una codificación con N ceros al final, cuántos elementos distintos, en media y asumiendo que pueden estar repetidos, he muestreado. En este caso, la HyperLogLog usa una aproximación al valor. Mediante la teoría de la probabilidad, se puede demostrar que el número de veces a muestrear una variable aleatoria antes de tener un suceso con probabilidad p es $1/p$. Recalcando que se asume como una aproximación al valor real de elementos distintos vistos, HyperLogLog considera que, si se han encontrado N ceros, se ha muestreado 2^N veces y que, de este modo, corresponde al número de valores distintos vistos. Esta aproximación, que funciona bastante mal para valores de N pequeños, consigue un error de solo el 2 % cuando el valor de elementos utilizado está por encima de 10^9.

7.3. Frecuencia

Enlaces recomendados

En los siguientes enlaces podéis ver ejemplos de *count-min sketch*: <https://www.youtube.com/watch?v=ibxXO-b14j4>, <https://www.youtube.com/watch?v=mPxslXpg8wA>.

La siguiente técnica que vamos a analizar, que completa la anterior, nos permite conocer con qué frecuencia ocurre cada uno de los valores distintos de un *stream*. De la misma forma que antes debemos entender que este problema no es complicado cuando todos los datos se pueden almacenar en una base de datos. En este caso, solo deberíamos agrupar los valores mediante un «*group by*» en SQL. De forma similar al problema anterior, este surge cuando los datos son tan abundantes que no podemos almacenarlos en memoria y ni siquiera podemos guardar todos los valores necesarios para realizar la estadística de forma exacta. Dadas estas circunstancias, el algoritmo más popular que nos permite obtener una aproximación a este valor es el *count-min sketch*. De forma similar a los otros algoritmos que ya hemos visto, el objetivo del algoritmo no es ofrecer un valor exacto, sino una aproximación, en este caso una cota superior. Es decir, vamos a obtener el número máximo de veces que **puede** haber aparecido el valor, pero sin pasarse. Hay que entender que muchas veces habremos visto el valor un número menor de veces.

El *count-min sketch*, como es habitual en las técnicas de síntesis aproximadas, se basa en una estructura de datos probabilística que nos sirve como tabla de frecuencia de los elementos en un *stream* de datos. En este caso, *count-min sketch* se basa en una matriz de tamaño predefinido ($N \times M$) donde operaremos para ir registrando la información de los elementos observados en el *stream*. Vamos a llamar a esta matriz, **matriz Sketch**.

La idea general del algoritmo es bastante básica y se basa en incrementar para cada nuevo elemento observado en el *stream* su posición asignada en la matriz Sketch. Claramente, debido a las restricciones de memoria implícitas en el problema, varios elementos deberán estar asignados a posiciones idénticas

dentro la matriz. Por este motivo el algoritmo es inexacto. En la citada matriz, las filas (N) van a corresponder a N funciones hash diferentes y M (el número de columnas) corresponde a un número arbitrario que nos permite controlar el tamaño de la estructura (cuánta memoria queremos usar a tal efecto). Como veremos, el valor de M nos permite controlar las colisiones que podamos encontrar en el cálculo de las funciones hash o, equivalentemente, el numero de elementos asignados a una misma celda de la matriz.

Vamos a verlo con unos ejemplos. Supongamos que $N = 1$ (una función de hash). Entonces, para cada elemento nuevo observado, vamos a calcular su hash (que nos indicará la fila que le corresponde al elemento) y para decidir la columna vamos a aplicar sobre el hash la operación «módulo M». La operación módulo nos va a devolver un número entre 0 y $M - 1$, que va a indicar la columna asignada a este dato (elemento). Como vemos, usamos la operación «módulo M» para restringir el rango de la función hash al tamaño de nuestra matriz Sketch. Claramente, el problema principal de la estructura de datos son las colisiones de la función hash debido a la operación módulo. A varios elementos distintos les puede llegar a corresponder la misma posición en la tabla. Esto deriva en un degeneración de la exactitud de la estructura en función del número de colisiones. Recordemos, en este punto, que pretendemos obtener una aproximación al valor, no el valor exacto. Para minimizar las colisiones, lo que podemos hacer es aumentar M. Para valores de $N > 1$, las operaciones a realizar son idénticas a las explicadas pero llevadas a cabo para cada fila de la matriz Sketch de manera independiente. Incrementar el valor de M es incluso más fácil, y solo hace falta aplicar la operación módulo con el valor correspondiente. Finalmente, para conocer la cota superior a la aparición de un elemento concreto (el número máximo de veces que ha aparecido un elemento), no hace falta más que mirar el número mínimo correspondiente a las distintas celdas correspondientes al elemento consultando. Veamos un ejemplo concreto:

Supongamos un *stream* de datos donde los valores van a llegar en el orden siguiente: [A, C, A, B, B, D, C, A, B, C, D] y una matriz Sketch compuesta por $N = 4$ funciones hash diferentes ($H1$…$H4$) y $M = 4$. Para evitar complicaciones en los cálculos (hash y operaciones módulo) de las celdas correspondientes a los diferentes elementos dentro de nuestro dominio de datos (A, B, C y D) vamos a asumir que podemos obtener estas celdas mediante la siguiente tabla:

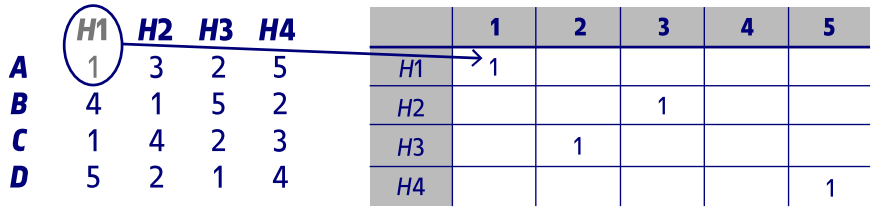
Figura 8. Matriz de valores de la función hash para cada elemento del *stream* de ejemplo

	H1	**H2**	**H3**	**H4**
A	1	3	2	5
B	4	1	5	2
C	1	4	2	3
D	5	2	1	4

Es decir, para la letra A (primera fila), la aplicación del hash $H1$ y la subsiguiente operación módulo nos indica la columna 1 de la matriz de frecuencias, y obtenemos la posición: fila 1, columna 1. Para el hash $H2$ la columna 3 y así para todas las funciones hash. Si

procesamos solo el primer elemento que observamos (letra *A*), vamos a obtener la matriz Sketch equivalente a:

Figura 9. Correspondencia en la matriz Sketch del Registro *A* del *stream* por las cuatro funciones hash



Esto es, para cada función hash (*H*1, …, *H*4) nos situamos en la celda correspondiente a *A* y le incrementamos su valor. El siguiente elemento que observamos del *stream* es *C*, con lo que incrementaremos las celdas correspondientes a *C* para las distintas funciones hash. Para *C*, estas son las columnas 1, 4, 2 y 3. Una vez hemos procesado todos los elementos del *stream*, la matriz resultante es la siguiente:

Figura 10. Correspondencia en la matriz Sketch completa del *stream* de ejemplo

	1	2	3	4	5
*H*1	6			3	2
*H*2	3	2	3	3	
*H*3	2	6			3
*H*4		3	3	2	3

Una vez calculada la matriz Sketch, si queremos saber cuántas veces ha aparecido el elemento *A*, consultamos aquellas celdas correspondientes al elemento *A*. Podemos revisar la figura 8.

Así, recuperamos el número correspondiente de cada celda correspondiente a *A*: {(*H*1,1), (*H*2,3), (*H*3,2), (*H*4,5)}, obtenemos los valores: {6, 3, 6, 3}. Si tomamos el valor mínimo de la lista, vemos que dicho valor es el 3 que es, efectivamente, el número de veces que aparece el elemento *A* en el *stream* analizado. No obstante, como hemos comentado, es posible que se produzcan duplicidades debido a la conversión binaria de las diferentes funciones hash y operaciones módulo. En nuestro ejemplo, esto sucede con los elementos *A* y *C* para el hash *H*1 y *H*2. Así, un mayor número de funciones hash y un mayor valor de *M* nos permite tener mayor dispersión y menos probabilidades de colisión.

7.4. *Membership*

Otra consulta con bastante utilidad sobre un *stream* de datos es preguntarse si un elemento ha aparecido con anterioridad en nuestro *stream*. En este caso la operación de *membership* nos permite responder si un elemento está presente en un conjunto, lo que aplicado al *streaming* indica si el elemento ha aparecido previamente en el flujo de datos. De la misma forma que en los métodos vistos anteriormente, esta operación es fácilmente implementable en SQL si podemos almacenar todos los datos. Una simple consulta «*select*» podría servir a tal efecto. Como hemos mencionado anteriormente, el problema aparece, o bien cuando no es viable almacenar toda la información de los distintos elementos que han aparecido en el flujo, o bien cuando es costoso mantener la consistencia de estos datos en un sistema distribuido. En estos casos, es útil entender cómo funciona el Bloom Filter.

7.4.1. *Bloom filter*

El *bloom filter* es una estructura de datos diseñada para responder, de forma rápida y eficiente desde un punto de vista de consumo de memoria, si un elemento está presente en un conjunto. El coste que pagamos por su eficiencia es la exactitud del método ya que, de forma similar al *count-min sketch*, no nos informa de manera exacta sobre si el elemento ha aparecido o no, sino que solo nos dice si el elemento definitivamente no está en el conjunto o si puede estar en él. Dicho de otro modo, el *bloom filter* nos permite conocer con total certeza si un elemento no está en el conjunto, pero un falso positivo es posible, es decir, puede no estar y aun así devolver una respuesta positiva. Dicha fiabilidad es configurable.

Enlaces recomendados

En los siguientes enlaces podéis ver videos de creación de *bloom filter* y ejemplos:

<https://www.youtube.com/watch?v=gBygn3cVP80>, <https://www.youtube.com/watch?v=RSwjdlTp108>.

Una vez más, *bloom* va a trabajar con diferentes funciones hash y, esta vez, con un solo *array* de bits a diferencia del *count-min sketch*. Cada una de las salidas de las diferentes funciones hash va a indicarnos una posición del array para cada uno de los elementos del *stream* de datos. Como ya hemos visto, la posibilidad de sobrescribir elementos en el *array* nos conduce sin duda a colisiones y a una sobreestimación.

De forma similar al *count-min sketch*, para saber si un elemento ha aparecido antes, es necesario verificar el valor de la celda correspondiente al elemento consultado obtenida mediante función hash de dicho elemento. Si en dicha posición encontramos un cero, sabemos seguro que el elemento no ha aparecido, pero si el valor es uno, no podemos tener certeza de que no haya sido debido a algún otro elemento cuya función hash devuelve el mismo valor.

Como ejemplo, recuperemos los valores de las diferentes funciones hash de la figura 8, pero vamos a cambiar la función $H1$ del elemento D a 6.

Figura 11. Correspondencia en la matriz Sketch completa del *stream* de ejemplo de la figura 8 con el cambio de una función

	H1	H2	H3	H4
A	1	3	2	5
B	4	1	5	2
C	1	4	2	3
D	6	2	1	4

Consideremos en esta ocasión el *stream* de datos siguiente: A, C, C, B, A.

Recordemos que, para calcular la frecuencia de un elemento en un *stream*, *bloom filter* utiliza un solo *array* sobre el cual superpone todos los valores de las distintas funciones hash. Para el elemento A, el *array* es como sigue:

Figura 12

1	2	3	4	5	6
1	1	1	0	1	0

Como vemos, cada nombre de columna corresponde al resultado específico de cada una de las funciones hash. El elemento A da como resultado 1, 3, 2 y 5 para las cuatro funciones hash, marcando como 1 dichas posiciones en cada columna.

Así, para los cinco elementos del *stream*, después de incrementar los contadores para cada posición, el *array* resultante es la siguiente:

Figura 13. Superposición de todos los valores hash para cada elemento del *stream*

1	2	3	4	5	6
5	5	4	3	3	0

Si dos elementos coinciden, se incrementa el contador (+1).

Si queremos saber si el elemento *C* se encuentra en nuestro *stream*, verificamos que los elementos en las posiciones 1, 4, 2 y 3 no estén vacíos (obsérvese en la figura 11 el resultado de cada una de las funciones hash para el elemento *C*). Y efectivamente así es, ninguna de las posiciones correspondientes a dichos valores está vacía, luego el elemento *C*, puede haber aparecido. ¿Y si queremos consultar el elemento *D*? Para saber si ha aparecido, revisamos si las posiciones 6, 2, 1 y 4 están vacías (son el resultado que obtenemos de cada una de las funciones hash sobre el registro *D*, como muestra la figura 11). La posición 6 está vacía, con lo que podemos asegurar que el elemento *D* no ha aparecido.

Si la función *H*1 para *D* no diera 6 como resultado sino 5, como en el ejemplo de la figura 10, ocurriría que todas las posiciones equivalentes del *array* para *D* contendrían elementos, sin embargo, *D* nunca ha aparecido. Por tanto, el algoritmo nos daría un **falso positivo**. Y es que el resultado acumulado de las funciones hash sobre los elementos observados distintos a *D* han generado valores coincidentes a las funciones hash del registro *D*.

Este es un claro ejemplo del tipo de las limitaciones que nos podemos encontrar en sistemas *big data* consumiendo datos en *streaming*. La naturaleza del consumo de los datos induce al uso de técnicas probabilísticas y rangos de confianza para los resultados de nuestros análisis. Aun así, estos algoritmos nos permiten conocer información muy relevante de un *stream*, especialmente necesaria para aplicaciones relacionadas con la seguridad, la privacidad y el control de sistemas críticos.

Bibliografía

Akidau, T.; Chernyak, S.; Lax, R. (2018). *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*. O'Reilly Media.

Cormode, G.; Muthukrishnan, S. (2004). «An Improved Data Stream Summary: The Count-Min Sketch and Its Applications». *Journal of Algorithms*. <http://dimacs.rutgers.edu/~graham/pubs/papers/cm-full.pdf>

Flajolet, P.; Fusy, É.; Gandouet, O.; Meunier, F. (2007). «Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm». En: *2007 Conference on Analysis of Algorithms, AofA 07*.

Julbe, F.; Conesa, J.; Casas, J.; Nin, J.; Rodríguez, M. E. (2019). *Captura, pre-processament i emmagatzematge de dades massives*. Universitat Oberta de Catalunya.

Marz, N.; Warren, J. (2015). *Big Data. Principles and best practices of scalable real-time data systems*. Manning Publications.

Vitter, J. S. (1985). «Random Sampling with a Reservoir». *ACM Transactions on Mathematical Software*. <www.cs.umd.edu/~samir/498/vitter.pdf>