IIC2440 – Procesamiento de Datos Masivos

Streams

1. Un primer problema

Supongamos que tenemos un servicio online, donde nos llegan requests de miles de usuarios. En este contexto, nos interesa manejar algunas estadísticas simples sobre esos requests: Cuál es el tipo de request más usada? Hay patrones en el uso de requests? En un entorno de "streams", lo que asumimos es que no tenemos memoria (o no queremos gastarla) para almacenar todos los requests, pero si para un pequeño porcentaje, digamos 10 %.

Para implementar eso, podemos usar por ejemplo una función de hash que mapee todo el espacio de request a $\{0, \ldots, 9\}$: cada request q que llega es hasheado con una función h a uno de 10 baldes. Si $h(q)$ es 0, entonces guardamos este request.

El problema es que hay algunas estadísticas que simplemente no podemos responder fielmente con este sample. Por ejemplo, no nos va a dar una buena respuesta si queremos ver si acaso un request q es generalmente seguido de un request q'.

El problema es que las estadísticas que consigamos, y el sample, va a variar dependiendo de la query que hagamos. Entonces, si queremos ver si acaso q precede a q', es mejor tomar un 10 % de los usuarios (por ejemplo, usando la misma función de hash h), y almacenar todas las queries para estos usuarios.

En general, en el entorno de stream vamos a asumir que no hay memoria para almacenar toda la información que nos va llegando. A veces, como en el caso de arriba, podemos trabajar con samples (aunque cómo samplear no siempre está claro). Otras veces, va a ser imposible, o extremadamente subóptimo, trabajar con samples (esto ocurriría, por ejemplo, si queremos contar la cantidad de request distintas que nos llegan).

2. Filtrado en streams

Otro problema importante en la práctica consiste en filtrar datos dentro de un stream, ya sea por que tenemos un firewall que solo deja pasar ciertos datos, o por que para algunos de estos datos vamos a iniciar un cierto proceso, pero para la mayoría no.

2.1. Filtros de Bloom

Consideremos, por ejemplo, un sistema con nombres de usuario (usernames) y contraseñas. Como somos populares, nos llegan cientos de intentos de logins por minuto. Los usuarios y sus passwords están almacenados en una tabla (que debería estar enriptada, pero para efectos de esta clase eso no importa). Cada request de login que nos llega, tenemos que ir a esa tabla a verificar si el usar corresponde con el password. Pero, el solo hecho de ver si un usuario está en nuestra base de datos nos toma tiempo!

Para poner números, supongamos que tenemos N igual a mil millones de usuarios. Si nuestra tabla está bien indexada, necesitamos alrededor de $\log(N)$ operaciones, o unas 30, para comprobar si un username está o no en la tabla. Realizar 30 operaciones está ok en muchos contextos, pero es demasiado para un contexto donde llegan cientos de logins al segundo.

Una buena solución es usar un filtro de Bloom. Usando un espacio de 1 gb, o alrededor de 8 mil millones de bits, hasheamos cada uno de los N usuarios usando una función de hash h que nos lleve cada string a un número i entre 0 y 8 mil millones. Luego construimos un arreglo donde la i-esima posición tiene un 1 si y solo si ese i resulta ser la imagen $h(s)$ de un hash de uno de nuestros usuarios, un 0 en las otras posiciones.

Cuando llega un usuario tentativo, digamos $u =$ `jreutter@ing.puc.cl`, realizamos lo siguiente. Tomamos $h(u)$, y eso nos da un i entre 0 y 8 mil millones.

- Si a ese número le fue asignado un 0 en nuestro arreglo, entonces sabemos que ese usuario no está en nuestra base de datos.
- Si a ese número le fue asignado un 1, lo tenemos que ir a buscar a nuestra base de datos, por que podría estar.

¿Qué hemos ganado con todo esto? De forma aproximada, si `jreutter@ing.puc.cl` no era parte de nuestros N usuarios, vamos a tener que ir a buscarlo igual con una probabilidad de $1/8$: son 8 mil millones de entradas, y solo mil millones de usuarios, por lo que hay a lo mas mil millones de unos en nuestro arreglo. Eso significa que con una pura operación de hash ya hemos descartado $7/8$ de los usuarios que no están!

Podemos hacer más, pero veamos primero un mejor análisis de la probabilidad de que un usuario que no está sea asignado un 1. No es exactamente $1/8$, pues los hashes pueden colisionar: dos usuarios en nuestro grupo N podrían ir a parar a la misma posición del arreglo.

Ahora, si tenemos B baldes en nuestro arreglo, y N usuarios, la posibilidad que uno de los B baldes sea la imagen del hash de un usuario es $1/B$, y que no sea la imagen de un usuario es $1 - 1/B$. Como son todos independientes, la probabilidad que un balde no sea la imagen de alguno de los N usuarios es $(1 - 1/B)^N = (1 - 1/B)^{\frac{B}{B}\frac{N}{B}}$. Usando que $(1 - 1/\epsilon)^\epsilon = 1/e$, cuando ϵ es pequeño, la probabilidad nos queda como $e^{-\frac{N}{B}}$.

En nuestro caso, la probabilidad que alguno de los bits si sea seteado a 1 es $1 - e^{-\frac{N}{8N}} = 1 - e^{-\frac{1}{8}} \approx 0{,}118$

2.2. Filtros de Bloom anidados

A pesar de nuestro avance, $1/8$ o 0,118 significa que para más del 10 % de usuarios que no están en N vamos a tener que ir a buscarlos a la base de datos igual, lo que todavía puede ser costoso. Pero podemos reducir esa priobabilidad usando varios filtros de Bloom!

> Un filtro de bloom para N strings con H hashes y B baldes funciona de la siguiente forma. Para cada string s en N, y para cada h en H, setear en 1 el balde correspondiente a $h(s)$, es decir, hacer $B[h(s)] = 1$.
>
> Cuando llega un nuevo string, el filtro lo marca como un potencial elemento en N cuando para cada h en H se tiene que $B[h(s)]$ es 1.

Volviendo al ejemplo anterior, si ahora tenemos dos funciones de hash mapeando al mismo conjunto de B baldes, la probabilidad que un balde no tenga un 1 ahora es $e^{-\frac{2N}{B}}$, simplemente por que la segunda función de hash es independiente de la primera, y entonces para ver que baldes tienen un 1, equivalente a tener el doble de usuarios.

Pero ahora cuando nos llega un usuario nuevo, vamos a ir a mirarlo si y solo si *ambas* funciones de hash nos mandan a un 1! En este caso, la probabilidad de que ambas funciones de hash entreguen un uno para un nuevo string corresponde a la probabilidad que ambos baldes a los que se hashean esos strings entreguen un 1. Esto se aproxima como $(1 - e^{-\frac{2N}{B}})^2$, o $(1 - e^{-\frac{1}{4}})^2$ en nuestro caso, lo que es aproximadamente 0,05.