Mining Data Stream

In molti contesti di analisi dei dati, si assume di lavorare con un database statico, in cui tutte le informazioni sono disponibili e accessibili in qualsiasi momento. Tuttavia, il mining di data streams introduce un paradigma diverso: i dati arrivano in flusso (stream) continuo e, se non vengono elaborati immediatamente o archiviati, vanno persi per sempre. Inoltre, il volume di dati può essere talmente elevato da rendere impossibile memorizzarli interamente in un database tradizionale.

Le tecniche per il mining di data streams si concentrano sulla sintetizzazione del flusso per ricavare informazioni utili utilizzando risorse limitate. Gli approcci principali includono:

1. **Campionamento e filtraggio (sampling):** estrarre campioni significativi dal flusso e rimuovere elementi indesiderati.
2. **Stima delle cardinalità:** calcolare approssimativamente il numero di elementi distinti nel flusso, utilizzando una quantità di memoria molto inferiore rispetto a quella necessaria per archiviare tutti i dati.
3. **Finestra scorrevole (sliding windows):** analizzare solo gli ultimi n elementi del flusso (la cosiddetta finestra), come se fossero una relazione in un database. Anche in questo caso, se il numero di flussi o la dimensione della finestra sono troppo grandi, è necessario riassumere i dati della finestra stessa.

In molti scenari di **Data Mining**, non è possibile conoscere in anticipo l'intero insieme di dati. La gestione dei data streams diventa quindi cruciale, specialmente quando il tasso di ingresso dei dati è determinato da fonti esterne, come:

- Query dei motori di ricerca.
- Aggiornamenti di stato su piattaforme come Twitter o Facebook.

Questi flussi di dati possono essere considerati **infiniti** e **non stazionari**, poiché la loro distribuzione cambia nel tempo. Gli elementi (o tuple) arrivano rapidamente attraverso uno o più flussi di ingresso, rendendo impossibile memorizzare l'intero flusso in modo accessibile.
Di conseguenza, il sistema può mantenere solo sintesi ridotte (short **sketches**) del flusso, che devono essere costantemente aggiornate.

Descriviamo quindi, un elemento dello steam come una tupla:

$$X = \langle x_1, x_2, x_3, \ldots, x_k \rangle$$

Di queste tuple, nella maggior parte dei casi consideriamo solo alcuni campi, detti **campi chiave**, come per esempio di un pacchetto IP, consideriamo per esempio gli indirizzi del mittente e di destinazione.

La sfida principale diventa quindi: **Come effettuare calcoli critici sui dati di un flusso utilizzando una quantità limitata di memoria secondaria?**

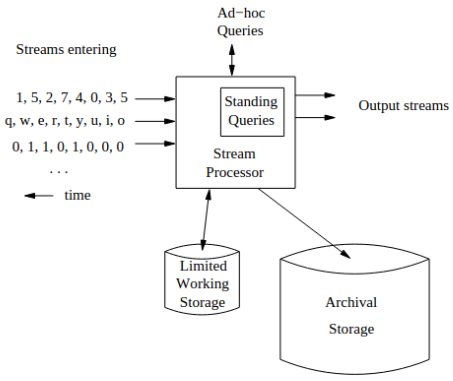


Figure 4.1: A data-stream-management system

I flussi di dati possono essere archiviati in un archivio di grandi dimensioni per conservarli nel lungo termine. Tuttavia, non è possibile utilizzare tale archivio per rispondere a query in tempo reale, poiché il recupero dei dati richiederebbe processi lenti e specifici.

Per l'elaborazione operativa delle query, si utilizza invece una memoria di lavoro (cache), che può contenere sintesi o porzioni di dati del flusso, ovvero i nostri sketch e sample.

Pattern Matching

Un primo problema fondamentale di statistica sui flussi di dati è il **pattern matching**. Supponiamo che gli elementi del flusso x_i appartengano a un alfabeto Σ, rendendo il flusso una stringa di lunghezza m su Σ.

Dati un pattern $y = y_1 y_2 \dots y_n \in \Sigma^n$ e un flusso x, l'obiettivo è determinare quante volte y compare come sottostringa all'interno del flusso x, ovvero trovare tutte le occorrenze in cui:

$$y = y_1 y_2 \dots y_n = x_i x_{i+1} \dots x_{i+n-1}$$

Cos'è uno sketch?

Una soluzione banale a questo problema, è di confrontare uno ad uno ciascun elemento $x_i x_{i+1} \dots x_{i+n-1}$ con $y_1 y_2 \dots y_n$. Osserviamo però che questo richiede tempo $O(n)$, che in questo caso non va bene.

Bisogna dunque definire uno sketch per risolvere in modo efficiente il problema del pattern matching.

> **Info**
>
> **Definizione:** Sia $x \in \Sigma^*$, allora un sketch di dati è una funzione random $f : x \in \Sigma* \Rightarrow \{0,1\}^k$ aventi le seguenti proprietà:
>
> 1. $k << |x|$, ovvero lo sketch deve essere molto più piccolo dell'input x.
> 2. f deve essere tale da poter essere **combinata** è **aggiornata** in tempo rapido, ovvero $f(xa) = F(f(x), f(a))$.
> 3. f deve mantenere le proprietà chiavi di x con alta probabilità.

La proprietà chiave da mantenere nel problema del patter matching è: date $x, y \in \Sigma^*$, vogliamo la nostra funzione f tale che:

$$x = y \Leftrightarrow f(x) = f(y) \text{ con alta probabilità}$$
$$x \neq y \Leftrightarrow f(x) \neq f(y) \text{ con alta probabilità}$$

Funzione Hash di Rabin

Studiamo adesso una funzione hash, che ci permette creare uno sketch dei dati, con le proprietà menzionate sopra.

Sia $\Sigma = [s]$, ovvero $\Sigma = \{1, 2, 3, \dots, s\}$ e $s \in \mathbb{N}^+$. Si fissa ora un numero primo $q > s$ tale che $q = \theta(s)$ e si sceglie un numero $z \in \mathbb{Z}_q = [q]$.

Rappresentiamo il nostro flusso x e il pattern da matchre y in questo modo:

$$x = \langle x[0], x[1], ..., x[n-1] \rangle \in [s]^n$$
$$y = \langle y[0], y[1], ..., y[n-1] \rangle \in [s]^n$$

Allora la funzione hash che definiamo è:

$$K_{q,z}(x) = \left(x[0]z^{n-1} + x[1]z^{n-2} + \cdots + x[n-1]z^0\right) \mod q = \left(\sum_{i=0}^{n-1} x[n-i-1]z^i\right) \mod q$$

Osserviamo che $K_{q,z}$ è un polinomio, ma una volta scelto random z, $K_{q,z}$ diventa un punto di questo polinomio in $\mathbb{Z}_q$ e non piu un polinomio. Questo rappresenta un efficienza in termini di spazio in quanto per rappresentare un numero in $\mathbb{Z}_q$ servono $log(q)$ bit che nei sistemi a registri è considerato costante. Quindi siamo riusciti a rappresentare una stringa di n caratteri con $log(q)$ bit (prima proprietà delle funzioni di sketch).

Success

Lemma: Sia $c \in \Sigma = [s]$, allora

$$K_{q,z}(xc) = (K_{q,z}(x)z + c) \mod q$$

Dimostrazione: Supponiamo che la nuova stringa di cui dobbiamo fare lo sketch è la seguente:

$$x = \langle x[0], x[1], ..., x[n-1], c\rangle \in [s]^{n+1}$$

Allora

$$K_{q,z}(xc) = \left[\left(x[0]z^n + x[1]z^{n-1} + \cdots + x[n-1]z\right) + cz^0\right] \mod q$$

Raccogliendo z, otteniamo

$$K_{q,z}(xc) = (K_{q,z}(x)z + c) \mod q$$

Osserviamo però che $K_{q,z}(x)$ è lo sketch precedentemente computato, perciò queste 3 operzioni richiedono tempo $\theta(1)$, quindi l'aggornamento dello sketch è rapido (seconda proprietà).

Dati adesso due sketch, per x e y, proviamo a calcolare $K_{z,q}(xy)$

$$K_{q,z}(xy) = (K_{q,z}(x)z^n + K_{q,z}(y)) \mod q$$

L'unico problema in questa formula che richiede tempo lineare è il calcolo di z^n, che può essere precomputato in tempo $O(log(n)log(q))$ usando la tecnica di esponenziazione rapida.

Note

Calcolare direttamente a^b può diventare inefficiente per b molto grande, poiché il risultato cresce esponenzialmente. Invece, si utilizza un approccio chiamato **esponenziazione rapida** (o "exponentiation by squaring"), che riduce la complessità a $O(log(b))$.
L'idea chiave è utilizzare la decomposizione binaria dell'esponente b:

- Scriviamo b in binario.
- Applichiamo una serie di moltiplicazioni e riduzioni modulo qq per calcolare solo i termini necessari.
- Passaggi dell'algoritmo
  i. Caso base:
    - Se $b = 0$, allora $a^b \mod q = 1$.
    - Se $b = 1$, allora $a^b \mod q = a \mod q$.
  ii. Esponente pari:
    - Riduciamo il problema sfruttando la proprietà:

$$a^b \mod q = (a^{\frac{b}{2}} \mod q)^2 \mod q$$

    - Eseguiamo il calcolo in modo ricorsivo per $\frac{b}{2}$.
  iii. Esponente dispari:

$$a^b \mod q = (aa^{b-1} \mod q) \mod q$$

iv. Ripeti finché b non si riduce a 0.

L'algoritmo è molto più veloce di calcolare a^b direttamente, perché riduce il numero di moltiplicazioni da b a $log_2(b)$. Questo è cruciale in algoritmi come l'hash di Rabin, dove b (es. z^i) può essere grande.

Dimostriamo ora che la funzione hash di Rabin può essere usata per creare un buon sketch per verificare l'identità di due stringhe.

Success

Lemma: Siano $x = \langle x_1, x_2, \ldots, x_n \rangle$ e $y = \langle y_1, y_2, \ldots, y_n \rangle$ due stringhe tali che $x \neq y$ e $|x| = |y| = n$ e sia $z \in \mathbb{Z}_q$, allora

$$Pr\left[K_{q,z}(x) = K_{q,z}(y) \mod q\right] \leq \frac{n}{q}$$

Dimostrazione: Abbiamo che

$$Pr\left[K_{q,z}(x) = K_{q,z}(y) \mod q\right] = Pr\left[K_{q,z}(x) - K_{q,z}(y) \equiv 0 \mod q\right] \Leftrightarrow K_{q,z}(x-y) = 0 \mod q$$

- La differenza $K_{q,z}(x) - K_{q,z}(y)$ è un polinomio in z, indicato come $K_{q,z}(x-y)$.
- Definiamo $x - y$ come il vettore differenza tra x e y, cioè: $x - y = \langle x_{n-1} - y_{n-1}, x_{n-2} - y_{n-2}, \ldots, x_0 - y_0 \rangle$.
- Il grado di $K_{q,z}(x-y)$ è al massimo n, dato che il vettore $x - y$ ha lunghezza n.
- Un polinomio di grado al massimo n ha al più n radici modulo q.
  Adesso ci sono due possibilità che $K_{q,z}(x-y) = 0 \mod q$:

1. Se è il polinomio identicamente nullo, ma questo è impossibile perché per ipotesi $x \neq y$, quindi esiste almeno un $j = 0, 1, 2, \ldots, n$ tale che $x_j \neq y_j$.
2. Che z sia una radice di questo polinomio. Poiché z è scelto uniformemente in Z_q, la probabilità che $K_{q,z}(x-y) \equiv 0 \mod q$ è al più $\frac{n}{q}$.
   Se scegliamo $q = \theta(n^b)$, allora la probabilità è $\leq \frac{1}{n^b}$ che ci va bene in quanto $\log(q) = \log(n^b) = b\log(n) = O(\log(n))$ spazio.

Algoritmo di Karp-Rabin

Supponiamo di aver elaborato una sequenza di caratteri fino a $x[1], x[2], \ldots, x[i]$ (dove $i \geq n$) e di conoscere due valori di hash:

- L'hash degli ultimi n caratteri nella sequenza: $K_{q,z}(\langle x[i-n+1], x[i-n+2], x[i] \rangle)$.
- L'hash del pattern che stiamo cercando: $K_{q,z}(y)$.

Confrontando questi hash, possiamo verificare se il pattern y appare negli ultimi n caratteri della sequenza. Se appare alloara aggiorniamo un contatore.

Il passo cruciale è aggiornare l'hash quando un nuovo carattere $x[i+1]$ arriva nel flusso di dati. Invece di ricalcolare l'hash per tutta la sottostringa di n caratteri, lo aggiorniamo:

- Rimuovendo il contributo del carattere più vecchio $(x[i-n+1])$.
- Aggiungendo il contributo del nuovo carattere $(x[i+1])$.

Questo si ottiene con la formula:

$$K_{q,z}(\langle x[i-n+2], x[i-n+3], \ldots, x[i+1] \rangle) = (K_{q,z}(\langle x[i-n+1], x[i-n+2], \ldots x[i] \rangle) - x[i-n+1] \times z^{n-1}) \times z + x[i+1] \mod$$

Dove:

- $z^{n-1} \mod q$ viene precomputato per risparmiare tempo.
- La sottrazione rimuove il contributo del carattere più vecchio.
- L'addizione incorpora il nuovo carattere.

Questo metodo opera in tempo costante per gli aggiornamenti, ma l'algoritmo deve memorizzare gli ultimi n caratteri del flusso per accedere a $x[i-n+1]$. Di conseguenza, richiede uno spazio $O(n)$.

Sampling in un datastream

Poiché non possiamo mantenere in memoria l'intero flusso di dati, un approccio naturale consiste nel memorizzare solo un campione del flusso, su cui poi eseguire le nostre query. Esistono due approcci comuni:

1. Eseguire il campionamento di una **proporzione fissa** di elementi dallo stream, ad esempio, un decimo dei dati.
2. Mantenere un campione casuale di **dimensione fissa**, anche su flussi potenzialmente infiniti.

L'obiettivo che vogliamo perseguire sono i seguenti:

1. Ad ogni istante di tempo t, vogliamo che il campione s contenga solo elementi che abbiano la stessa probabilità di essere selezionati dallo stream, indipendentemente dal momento in cui sono stati osservati.
2. Efficienza, aggiornameto veloce e poco spazio di memoria.

Sampling di una proporzione fissata

Ci troviamo nel seguente scenario: **flusso di query in un motore di ricerca**
Si vuole selezionare un campione $S \subseteq U$ da un flusso di dati U, mantenendo una proporzione fissa dei dati originali (ad esempio, il 10%).
Il flusso di dati in ingresso (U) è costituito da tuple nella forma: `(userId, query, time)` , dove:

- `userId` identifica un utente.
- `query` è la query effettuata.
- `time` è il timestamp dell'operazione.

Una domanda tipica a cui si vuole rispondere è: *"Quanto spesso un utente effettua la stessa query in un solo giorno?"*. Questo è rilevante perché un motore di ricerca potrebbe voler monitorare il comportamento degli utenti senza archiviare l'intero flusso di query (per esempio, solo il 10% del flusso giornaliero può essere memorizzato per limiti di spazio).

Il problema algoritmico che si vuole risolvere è:
Trovare un sottoinsieme $S \subseteq U$ che approssimi bene, per ciascun utente medio (u) e per ogni query (q), la frazione di occorrenze di q effettuate da u.

- S deve garantire che, in aspettativa, la distribuzione delle query nel campione sia rappresentativa di quella originale.

Infine, definiamo **q-occorrenza** una tupla $t \in U$ che include la query q. Per esempio `(userId, q, time)` è una q-occorrenza se la query è q.

Questo tipo di campionamento è utile in contesti in cui:

- Lo spazio di memoria è limitato.
- È necessario rispondere a domande su statistiche aggregate (ad esempio, la frequenza delle query per utente) senza memorizzare ogni singolo dato.

Algoritmo wrong

Analizziamo ora un **algoritmo ingenuo (Naïve Algorithm)**, per il problema del campionamento a proporzione fissata.

Algoritmo ingenuo (N-Algo)

- Input: Un flusso di tuple U, dove ogni tupla ha la forma `(userId, query, time)` .
- Output: Un sottoinsieme $S \subseteq U$ campionato casualmente.

Si parte con $S = \emptyset$ (un insieme vuoto).
Per ogni tupla nel flusso U:

- Passo 1: Associare casualmente ogni tupla a un numero intero $z \in [10] = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (che rappresenta 10 "buckets"). Per fare questo scegliamo una **funzione hash universale** che mappa l'insieme delle tuple in $[10]$
- Passo 2: Aggiungere la tupla `(userId, query, time)` a S solo se $z = 0$. Altrimenti, la tupla viene scartata.

Alla fine, per ogni query q, calcolare le frazioni di occorrenze basandosi solo sul sottoinsieme S, ignorando il resto del flusso.

Questo algoritmo ha un enorme problema, ovvero **non garantisce un campionamento rappresentativo per ogni utente e query**
L'algoritmo assegna ogni tupla a un bucket casuale indipendentemente dal contesto (ad esempio, lo stesso utente o la stessa query). Ciò significa che le distribuzioni delle query (o degli utenti) non vengono rispettate nel campionamento. Alcuni utenti o query potrebbero essere rappresentati in modo sproporzionato o completamente esclusi.

L'analisi dell'algoritmo ingenuo si basa sul caso "simple homogeneous", dove considera un flusso di dati U generato da un utente medio, che esegue due tipi di query:

- Alcune query (**singleton**) che vengono effettuate una sola volta, in numero m.
- Altre query (**duplicate**) che vengono effettuate due volte, in numero d.

In totale, l'utente ha $m + 2d$ occorrenze di query nel flusso.

L'obiettivo dell'algoritmo di campionamento è produrre un sottoinsieme S che rappresenti in modo accurato le proporzioni di queste query. In particolare, vogliamo che S rispecchi la frazione corretta delle query duplicate rispetto al totale, cioè:

$$\frac{d}{m+d}$$

L'algoritmo campiona casualmente le tuple del flusso U, mantenendo solo il 10% di esse. La probabilità di mantenere una singola tupla è quindi $\frac{1}{10}$. In media, la dimensione del campione S è pari a $\frac{1}{10}\,|U|$, il che significa che S contiene il 10% delle query totali, indipendentemente dal tipo.
Per quanto riguarda le query singleton (quelle emesse una sola volta), il campionamento è abbastanza semplice: il 10% di m viene mantenuto, quindi nel campione ci aspettiamo circa $\frac{m}{10}$ query singleton. Questo è corretto e non crea problemi.

Le difficoltà emergono, invece, quando consideriamo le query duplicate. Queste sono emesse due volte nel flusso e possono comparire in S in due modi:

- Una query duplicata può apparire una sola volta in S, se una delle due occorrenze è selezionata.
- Una query duplicata può apparire due volte in S, se entrambe le occorrenze vengono selezionate.

Per ciascuna query duplicata, la probabilità che entrambe le sue occorrenze siano incluse in S è:

$$P[\text{entrambe le occorrenze in } S] = \frac{1}{10} \cdot \frac{1}{10} = \frac{1}{100}$$

Quindi, solo una frazione molto piccola $(\frac{1}{100})$ delle coppie duplicate verrà mantenuta completamente. Questo è un problema significativo, perché nel flusso originale ogni query duplicata compare due volte, e questa caratteristica dovrebbe essere rispettata anche nel campione.

D'altra parte, è più probabile che una query duplicata appaia solo una volta in S. La probabilità che almeno una delle sue due occorrenze sia inclusa è:

$$P[\text{almeno una occorrenza in } S] = 1 - P[\text{nessuna occorrenza in } S] = 1 - (\frac{9}{10} \cdot \frac{9}{10}) = \frac{19}{100}$$

Quindi, in media, il 18% delle query duplicate sarà rappresentato almeno una volta nel campione.

Tuttavia, per riflettere accuratamente la proporzione delle query duplicate rispetto al totale, non basta che queste siano incluse almeno una volta. Dovremmo rappresentare correttamente il loro numero relativo di occorrenze rispetto alle query singleton.

L'N-Algo produce una frazione approssimativa di query duplicate nel campione pari a:

$$\frac{\text{duplicate nel campione}}{\text{totale nel campione}}$$

Questa frazione non corrisponde al valore corretto $(\frac{d}{m+d})$ calcolato sul flusso originale. Invece, si ottiene una frazione distorta a causa della bassa probabilità di includere entrambe le occorrenze delle query duplicate.

$$\frac{\frac{d}{100}}{\frac{m}{10} + \frac{d}{100} + \frac{18d}{100}} = \frac{d}{10m + 19d}$$

Tale valore è molto inferiore rispetto al valore corretto $\frac{d}{m+d}$.

Algoritmo corretto

L'N-Algo campionava ogni tupla nel flusso di dati U in modo indipendente, il che causava una distorsione nei risultati, specialmente per le query duplicate. La soluzione qui proposta evita di campionare singole tuple; invece, si concentra sul campionare interamente gli utenti e tutte le loro query.

L'algoritmo parte dal set di tutti gli utenti I, rappresentati dai loro `userID` . Si sceglie una funzione universale di hash e si mappano tutti gli utenti in 10 buckets. Successivamente, si seleziona un bucket specifico (per esempio il bucket 0) per ogni utente nel bucket selezionato, vengono incluse tutte le sue query (q-occorrenze) nel campione S. Rispetto all'approccio banale, non si campionano le query in modo indipendente, ma si mantiene interamente il comportamento di ciascun utente selezionato. Una volta raccolto S, è possibile calcolare medie e altre statistiche relative alle query q. Poiché il campione preserva la struttura originale dei dati a livello di utente, le statisitche saranno rappresentative.

L'algoritmo presuppone di conoscere l'insieme completo degli `userId` (I) in anticipo. Tuttavia, se I non è noto, è possibile modificare l'approccio:

- Durante la fase di streaming dei dati, si può applicare l'hashing "on-the-fly" agli `userId` man mano che le tuple vengono osservate. Questo consente di costruire i buckets in tempo reale senza avere l'elenco completo degli utenti.

Analizziamo ora tale algoritmo e capiamo come esso riesca a creare un sample rappresentativo del flusso di dati. Iniziamo con alcune definizioni:

1. Q: L'insieme di tutte le query.
2. I: L'insieme di tutti gli ID degli utenti (`userId`).
3. $x(v, q)$: Numero di occorrenze della query q effettuate dall'utente v nel flusso di input (valore deterministico).
4. $X_S(V, q)$: Numero di occorrenze di q per un utente V selezionato casualmente nel campione S. Questo valore è una variabile casuale perché dipende dalla selezione casuale degli utenti in S (variabile random).

L'obiettivo dell'algoritmo è stimare il valore medio delle q-occorrenze per tutti gli utenti del flusso ($AVG_{v \in I}(x(v, q))$) usando solo i dati nel campione S. In altre parole, si vuole che:

$$\text{Media nel campione} \rightarrow \text{Media vera}$$

ovvero vogliamo calcolare la media delle q-occorrenze per tutti gli utenti v nel flusso originale I

$$\text{Media vera} = AVG_{v \in I}(x(v, q))$$

ma, tale media la dobbiamo stimare usando solo gli utenti del campione S

$$\text{Media stimata} = AVG_{V \in S}(X_S(V, q))$$

Ricordiamo che nell'algoritmo, gli ID degli utenti (v) vengono scelti uniformemente random e per ogni utente selezionato si includono tutte le sue q-occorrenze nel campione S.
Poiché ogni utente ha la stessa probabilità di essere incluso in S, il valore medio delle $X_S(V, q)$ (calcolato sul campione S) è un **buon stimatore** del valore medio delle $x(v, q)$ sull'intero flusso.

$$E_{V \in S}(X_S(V, q)) = AVG_{v \in I}(x(v, q))$$

La media stimata usando il campione convergerà alla media vera man mano che il campione diventa sufficientemente grande ($|S| \rightarrow |I|$).

Quindi per il teorema del limite centrale (legge dei grandi numeri):

$$AVG_{V \in S}(X_S(v, q)) \rightarrow AVG_{v \in I}(x(v, q)) \text{ quando } |S| \rightarrow |I|$$

Rimane da mostrare che la dimensione del campione S è il 10% di U.

1. Definiamo una variabile aleatoria C_j nel seguente modo

$$C_j = \begin{cases} 1 & \text{se l'utente } j \text{ è selezionato nel campione } S \\ 0 & \text{altrimenti} \end{cases}$$

2. Sia x_j il numero di query nel flusso U generate dal utente j

Dobbiamo adesso stimare la dimensione di S:

$$E[|S|] = Pr[C_j = 1] \cdot \sum_j x_j = \frac{1}{10} \cdot \sum_j x_j = \frac{|U|}{10}$$

Dunque, la dimensione attesa di S è il 10% di U.

Nonostante la dimensione attesa di S sia ben definita, la varianza può essere molto alta. Questo significa che il numero di elementi effettivamente inclusi in S potrebbe variare notevolmente rispetto al valore atteso, specialmente in situazioni di distribuzione sbilanciata dei dati.
Supponiamo che ci siano solo due utenti nel flusso:

- L'utente j_1 genera il 90% degli elementi.
- L'utente j_2 genera il 10% degli elementi.

Cosa succede con il campionamento casuale?

- Se j_1 viene selezionato (hashato nel bucket campionato), allora il campione S conterrà 90% degli elementi di U.
- Se invece j_2 viene selezionato, il campione S conterrà solo il 10% degli elementi di U.

Questo porta a una varianza molto alta nella dimensione di S. In alcuni casi, S sarà molto grande (se j_1 è selezionato), mentre in altri sarà molto piccolo (se j_2 è selezionato).
Di conseguenza, la rappresentatività del campione può variare significativamente a seconda degli utenti selezionati.

Il problema **generale del campionamento nei flussi** di dati riguarda la selezione di un sottoinsieme delle tuple in base a una chiave. Quindi una tupla t è composta da k campi, di cui uno rappresenta la chiave.

$$t_i = \langle X_{i1}, X_{i2}, \ldots, X_{ik} \rangle$$

La chiave può essere una singola componente (es., "utente") o una combinazione di componenti (es., "utente-query").

Per creare un campione di proporzione a/b:

- Si applica una funzione di hashing alla chiave di ogni tupla che assegna valori in b bucket.
- Si accettano solo le tuple il cui valore hash rientra in una certa soglia ($< a$).

Se la chiave è composta da più di una componente, la funzione di hashing deve combinare i valori di quelle componenti per generare un singolo valore hash.

Il risultato sarà un campione costituito da tutte le tuple con determinate chiavi. Le chiavi selezionate rappresenteranno approssimativamente a/b di tutte le chiavi che appaiono nel flusso.

Sample di dimensione fissa

Vogliamo mantenere un campione S di dimensione fissa s da un flusso di dati potenzialmente infinito. I vincoli da rispettare sono:

- La memoria disponibile è limitata, quindi S deve contenere sempre e solo s elementi.
- Supponendo che al tempo n abbiamo visto n elementi del flusso, ogni elemento visto nel flusso deve avere una **probabilità uniforme** s/n di essere incluso nel campione, indipendentemente da quanto è stato osservato.

La soluzione a questo problema è il **Reservoir Sampling**, un algoritmo per mantenere un campione casuale di dimensione fissa s da un flusso di dati infinito.

1. **Fase iniziale:**
   Quando il flusso di dati inizia, memorizza i primi s elementi nel campione S, tali elementi costiutiscono il reservoir iniziale.
2. **Dopo i primi s elementi ($n > s$):**
   Quando un nuovo elemento n-esimoa arriva dal flusso:
   i. Con probabilità $\frac{s}{n}$, **includi l'elemento n nel campione S**, altrimenti scartalo.
   ii. Se l'elemento s viene incluso, **sostituiscilo con uno degli s elementi già presenti in S**, scelto in modo uniforme random.

Success

Claim

Dopo aver osservato n elementi, ogni elemento visto finora ha una probabilità $\frac{s}{n}$ di far parte del campione S.

Dimostrazione

La dimostrazione avviene per **induzione** su n.

1. Caso base

Quando $n = s$, abbiamo visto solo s elementi. L'algoritmo memorizza tutti i primi s elementi nel campione S. Pertanto:

$$Pr[\text{ogni elemento è in } S] = \frac{s}{s} = 1$$

La proprietà desiderata è quindi soddisfatta: ogni elemento visto finora è nel campione con probabilità 1.

2. Passo induttivo

Supponiamo che, dopo aver visto n elementi, ogni elemento visto finora abbia probabilità $\frac{s}{n}$ di essere nel campione S. Ora dimostriamo che, quando arriva l'elemento $n+1$, la proprietà rimane valida.

Quando arriva l'elemento $n+1$, l'algoritmo esegue i seguenti passi:

1. Decide di includere $n+1$ in S con probabilità $\frac{s}{n+1}$.
2. Se l'elemento $n+1$ viene incluso, sostituisce uno degli s elementi esistenti nel campione, scelto uniformemente a caso.

Analizziamo la probabilità che un elemento x, già presente nel campione S dopo n passi, rimanga in S dopo il passo $n+1$:

1. **Caso 1: $n+1$ non viene incluso nel campione.**
   In questo caso, il campione non cambia, quindi x rimane in S. La probabilità che ciò accada è:

$$Pr[n+1 \text{ non viene incluso}] = 1 - \frac{s}{n+1}$$

2. **Caso 2: $n+1$ viene incluso nel campione.**
   In questo caso, $n+1$ sostituisce un elemento esistente nel campione, scelto uniformemente a caso.

- La probabilità che x **non venga rimosso** (e quindi rimanga in S) è:

$$Pr[x \text{ non viene rimosso}] = 1 - \frac{1}{s}$$

- La probabilità che $n+1$ venga incluso è:

$$Pr[n+1 \text{ incluso}] = \frac{s}{n+1}$$

- Combinando, la probabilità che x **non venga rimosso** in questo scenario è:

$$\frac{s}{n+1} \cdot \left(1 - \frac{1}{s}\right)$$

Probabilità totale che x rimanga in S

Combinando i due scenari:

$$Pr[x \text{ rimane in } S] = Pr[n+1 \text{ non incluso}] + Pr[n+1 \text{ incluso e } x \text{ non rimosso}]$$

Sostituendo le probabilità:

$$Pr[x \text{ rimane in } S] = \left(1 - \frac{s}{n+1}\right) + \frac{s}{n+1} \cdot \left(1 - \frac{1}{s}\right)$$

Semplificazione

Semplificando:

$$Pr[x \text{ rimane in } S] = \left(1 - \frac{s}{n+1}\right) + \frac{s}{n+1} \cdot \frac{s-1}{s}$$

$$Pr[x \text{ rimane in } S] = \frac{n}{n+1}$$

Pertanto, dopo il passo $n+1$, la probabilità che x rimanga in S è:

$$Pr[x \text{ in } S] = \frac{s}{n} \cdot \frac{n}{n+1} = \frac{s}{n+1}$$

Conclusione

Per induzione, abbiamo dimostrato che, dopo $n+1$ passi, ogni elemento visto finora ha probabilità $\frac{s}{n+1}$ di essere incluso nel campione S. L'algoritmo quindi garantisce un campionamento uniforme con memoria $O(s)$ e tempo $O(1)$ per elemento.

Sliding Window - Contare Bit

Le sliding window rappresentano un modello molto utile nel contesto dell'elaborazione di flussi di dati. L'idea è quella di analizzare i dati in arrivo considerando una "finestra" di lunghezza N, che include solo gli ultimi N elementi ricevuti. In questo modo possiamo concentrarci sui dati più recenti trascurando quelli passati. Un caso particolarmente interessante è quello in cui N è così grande da rendere impossibile memorizzare tutti i dati in memoria o sul disco.

Un'applicazione concreta di questo modello è rappresentata dall'analisi delle transazioni finanziarie di una grande azienda. Ogni elemento del flusso può essere visto come una transazione. Ad esempio, possiamo tenere traccia delle vendite di un determinato prodotto x, rappresentandole come un flusso binario (0/1): il valore 1 indica che il prodotto è stato venduto nella j-esima transazione, mentre il valore 0 indica che non lo è stato.

Un tipico tipo di query in questo contesto potrebbe essere: "Quante volte è stato venduto il prodotto x nelle ultime k transazioni?", dove k è un valore minore o uguale a N. Questo tipo di interrogazione permette di ottenere informazioni immediate e utili sulle performance recenti di un prodotto o sull'andamento delle vendite in una finestra temporale specifica.

Problema

Sia dato un flusso binario infinito $I = b_1, b_2, b_3, \ldots$ dove ogni elemento $b_i \in \{0, 1\}$ rappresenta un bit del flusso in arrivo in sequenza temporale.

Input:

- Una finestra temporale di lunghezza N, che considera solo gli ultimi N bit del flusso I.
- Una variabile k, tale che $1 \leq j \leq N$, che definisce la sotto-finestra di lunghezza k all'interno della finestra principale.

Funzione da calcolare:

La funzione $\#1(I, N, k)$ è definita come il numero di bit uguali a 1 presenti negli ultimi k bit della finestra di lunghezza N. Formalmente:

$$\#1(I, N, k) = \sum_{i=N-k+1}^{N} b_i$$

dove b_i è il valore del bit i-esimo nella finestra corrente.

Obiettivo:

Calcolare $\#1(I, N, k)$ per un valore qualsiasi di k, con $k \leq N$, in modo efficiente, considerando i seguenti vincoli:

1. **Flusso infinito:** Non è possibile memorizzare tutti i bit del flusso I.
2. **Efficienza temporale:** La funzione $\#1(I, N, k)$ deve essere calcolata e aggiornata in tempo costante o sublineare rispetto a N.
3. **Efficienza spaziale:** L'algoritmo deve utilizzare uno spazio di memoria significativamente inferiore rispetto alla memorizzazione esplicita di tutti i bit della finestra N.

Soluzione banale:

Una soluzione semplice consiste nel memorizzare esplicitamente gli ultimi N bit del flusso I. Ogni volta che un nuovo bit arriva, il bit più vecchio viene scartato, e il conteggio $\#1$ viene aggiornato sulla base del nuovo contenuto della finestra. Questa soluzione richiede $O(N)$ spazio di memoria e $O(k)$ tempo per calcolare $\#1$, ma non è scalabile per valori molto grandi di N.

✓ **Success**

Claim

Per qualsiasi distribuzione arbitraria di input, calcolare esattamente la funzione $\#1(\mathcal{I}, N, k)$ richiede uno **spazio di memoria** pari almeno a N. Questo implica che non è possibile ottenere una rappresentazione compatta ($|R(N,k)| < N$) dei dati che sia in grado di fornire una soluzione esatta e **deterministica** per tutti i possibili flussi di input.

Dimostrazione

Supponiamo che esista uno sketch $R(N,k)$ che codifica il contenuto di una finestra di lunghezza N in uno spazio minore di N, cioè $|R(N,k)| < N$.

1. **Esistenza di finestre indistinguibili**:
   Se $|R(N,k)| < N$, allora esistono almeno due finestre x e w, entrambe di lunghezza N, che **condividono la stessa rappresentazione** R, ma sono diverse tra loro.
   Quindi:

$$x = \ldots, 1, x_{k-1}, x_{k-2}, \ldots, x_1$$

$$w = \ldots, 0, w_{k-1}, w_{k-2}, \ldots, w_1$$

   Dove esiste un indice k tale che $x_k \neq w_k$ (ad esempio, $x_k = 1$ e $w_k = 0$) e tutti gli altri bit coincidono.
2. **Impossibilità di distinguere tra x e w**:
   Poiché x e w condividono la stessa rappresentazione R, qualsiasi algoritmo basato su R non può distinguere tra le due finestre. Di conseguenza, l'algoritmo **fallisce** nel calcolo di $\#1(I, N, k)$, dato che:

$$\sum_{i=1}^{k} x_i \neq \sum_{i=1}^{k} w_i$$

   Infatti, il numero di bit pari a 1 differisce tra x e w.
3. **Conclusione**:
   Questa contraddizione dimostra che una rappresentazione $R(N,k)$ di dimensione $|R| < N$ non è sufficiente per calcolare $\#1(I, N, k)$ in modo deterministico ed esatto.
   Pertanto, è necessario uno spazio di memoria **almeno pari a** N per garantire una soluzione esatta.

Soluzione Approssimata

Una semplice soluzione che non risolve il nostro problema si basa sull'assunzione che lo stream di bit in input abbia distribuzione uniforme, ovvero che i bit pari a 1 e quelli pari a 0 siano distribuiti uniformemente nella finestra di lunghezza N.

Siano:

- S: rappresenta il numero di bit pari a 1 presenti nella finestra di lunghezza N. Questo contatore richiede $O(logN)$ spazio in quanto dobbiamo memorizzare un intero modulo N.
- Z: rappresenta il numero di bit pari a 0 nella finestra di lunghezza Z. Tuttavia, non è necessario mantenere un contatore separato per Z, perché $S + Z = N$, il che consente di calcolarlo implicitamente.

La funzione $\#1(I, N, k)$, ovvero il numero di bit pari a 1 nei più recenti k bit, viene stimata come:

$$\#1(I, N, k) \approx k \cdot \frac{S}{S + Z}$$

La soluzione è corretta solo quando $k = N$, perché l'assunzione di uniformità si applica all'intera finestra N, ma potrebbe non essere valida per sotto-finestre di lunghezza $k < N$.

L'algoritmo di Datar-Gionis-Indyk-Motwani

L'algoritmo **DGIM** utilizza $O(log^2(N))$ bit per rappresentare una finestra di N bit e consente di stimare il numero di 1 nella finestra con un errore che non supera il 50%.

Ad ogni bit del flusso viene associato an un **timestamp**, ovvero la posizione in cui arriva. Il primo bit avrà timestamp 1, il secondo timestamp 2, e cosi via.

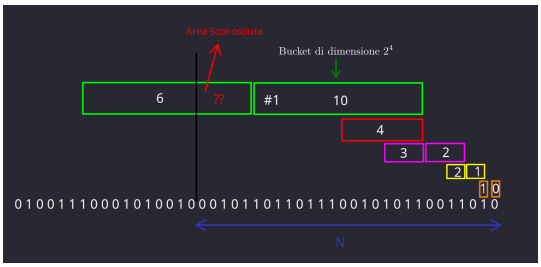
Poiché dobbiamo distinguere solo le posizioni all'interno della finestra di lunghezza N, rappresenteremo il timestamp in modulo N. In questo modo, i timestamp possono essere rappresentati con $log_2(N)$ bit.

Primo approccio (intuizione)

Dividiamo la finestra in **bucket**, ognuno dei quali comprende:

1. Il **timestamp** del suo estremo destro (cioè il bit più recente contenuto nel bucket)
2. Il numero di bit pari a 1.

I bucket rappresentano sotto-finistre di dimensione **esponenzialmente crescenti** (1, 2, 4, 8, potenze di 2). Guardando indietro lungo il flusso, ogni bucket rappresenta una parte crescente della finestra, con un bucket più grande che copre più dati rispetto ai bucket più piccoli. Se un bucket più piccolo inizia nello stesso punto di un bucket più grande, il bucket più piccolo viene scartato. Questo aiuta a ridurre il numero di bucket mantenuti in memoria.



Analizziamo l'esempio in figura:

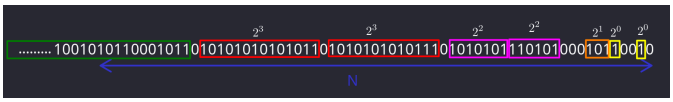
- Una finestra di larghezza $N = 16$ è divisa in buckets.
- Ogni bucket tiene traccia del numero di 1 presenti al suo interno (ad esempio, un bucket di dimensione $2^4 = 16$ contiene 6 bit pari a 1).
- Tuttavia, c'è un'area ignota ("Area Sconosciuta") verso l'inizio della finestra. Questo accade perché l'approssimazione non riesce a determinare con precisione quanti 1 si trovano in quella zona. L'errore massimo dipende dalla distribuzione degli 1 nella finestra e dall'approssimazione usata per la zona "Sconosciuta".
- L'algoritmo utilizza $O(log^2(N))$ bit per rappresentare la finestra di lunghezza N.
  Questo perché il numero di bucket è $O(log(N))$, e ciascun bucket richiede $log_2(N)$ bit per memorizzare i dati.
- Quando nuovi bit entrano nel flusso, l'aggiornamento dello stato dei bucket è facile. Questo rende il metodo pratico per applicazioni in tempo reale, dove il flusso di dati è continuo.
- L'errore massimo nel calcolo di $\#1(I, N, k)$ non supera il numero di bit pari a 1 presenti nel'area sconosciuta.

> **Note**
>
> - **Perché per una lunghezza fissata 2^t, esistono solo 2 bucket?**
> - Questo avviene perché l'algoritmo raggruppa i dati in modo esponenziale. Quando due bucket più piccoli rappresentano una stessa lunghezza 2^t, uno di essi viene eliminato per evitare ridondanze. Alla fine, ci saranno al massimo 2 bucket per ciascuna lunghezza 2^t.

Secondo approccio (giusto)

Inizialmente, la dimensione del bucket era definita come la **lunghezza totale** in termini di numero di bit, considerando sia gli 0 che gli 1. L'informazione che veniva mantenuta per ciascun bucket era il numero di bit pari a 1 contenuti al suo interno. Tuttavia, questo approccio non garantiva un'ottima approssimazione, poiché i bucket più grandi potevano contenere molti 0, contribuendo a errori significativi.

Ora, modifichiamo il criterio di definizione della dimensione del bucket: la **dimensione** sarà rappresentata dal **numero di 1** contenuti nel bucket, e non più dalla lunghezza complessiva in termini di bit. Di conseguenza, la dimensione di ciascun bucket crescerà in modo esponenziale rispetto al numero di 1 che contiene. Questo significa che il bucket non memorizza più la lunghezza totale in bit, ma esclusivamente il conteggio degli 1, rendendo la rappresentazione più precisa e riducendo il margine di errore.
Infatti, se nella finestra ci sono pochi 1, i blocchi rimangono piccoli, garantendo che gli errori di approssimazione siano proporzionalmente ridotti.



Per rappresentare un bucket servono $O(log(N))$ bit perché:

1. **Timestamp dell'estremo destro del bucket (modulo N):**
   Questo richiede $log_2(N)$ bit, poiché si tratta di un valore compreso tra 0 e $N-1$.
2. **Numero di 1 contenuti nel bucket:**
   Siccome il numero di 1 (i) è una potenza di 2 (2^j), possiamo rappresentare i codificando j in binario. Dato che $j \leq log_2(N)$, servono $log_2(log_2(N))$ bit per rappresentare j.

> **Note**
>
> **Regole per la creazione dei bucket**
>
> 1. L'estremo destro del bucket deve essere un 1.
>   Ogni bucket termina sempre in una posizione contenente un bit pari a 1.
> 2. Ogni 1 appartiene a un bucket.
>   Tutti i bit pari a 1 nel flusso devono essere inclusi in almeno un bucket.
> 3. Un bit non può appartenere a più di un bucket.
>   Ogni posizione con un 1 è assegnata esattamente a un unico bucket, senza sovrapposizioni.
> 4. Massimo due bucket per ogni dimensione.
>   Per ciascuna dimensione 2^j (numero di 1), ci possono essere al massimo due bucket.
> 5. Le dimensioni dei bucket devono essere potenze di 2:
>   Le dimensioni valide per i bucket sono $2^0, 2^1, 2^2, \ldots$.
> 6. Le dimensioni dei bucket non diminuiscono andando a sinistra.
>   Procedendo indietro nel tempo (verso sinistra), le dimensioni dei bucket devono rimanere costanti o aumentare.

Come mantenere le condizioni del DGIM

Supponiamo di avere una finestra di lunghezza N rappresentata correttamente dai bucket che soddisfano le condizioni dell'algoritmi **DGIM**. Cosa succede quando un nuovo bit entra nel flusso? Come aggiorniamo correttamente i bucket e quanto tempo impieghiamo?

1. **Verifica del bucket più vecchio:**
   Quando un nuovo bit entra nel flusso, l'algoritmo deve assicurarsi che i bucket rappresentino solo i dati all'interno della finestra corrente di lunghezza N. Ciò significa che, se il timestamp del bucket più a sinistra (il più vecchio) è fuori dalla finestra, questo bucket deve essere rimosso.
2. **Gestione del nuovo bit:**
   - Se il nuovo bit è uno 0, non sono necessarie ulteriori modifiche ai bucket.
   - Se il nuovo bit è uno 1, invece, potrebbero essere necessari alcuni aggiustamenti:

   i. **Crea un nuovo bucket** con il timestamp corrente e dimensione pari a 1 (2^0).

   ii. Se c'è **solo un bucket** di dimensione 1, non è necessario fare altro. Se ci **sono tre bucket** di dimensione 1, prendi i due più vecchi e combinali tra di loro. Per combinare due bucket adiacenti, sostituiscili con un nuovo bucket di dimensione doppia. Il **timestamp** del nuovo bucket sarà quello del bucket più a destra (più recente) tra i due.

   iii. La combinazione di due bucket di dimensione 1 potrebbe creare un terzo bucket di dimensione 2. Se ciò accade, combina i due bucket più a sinistra di dimensione 2 in un bucket di dimensione 4. Questo processo può propagarsi attraverso le dimensioni dei bucket. Se tre bucket di dimensione 4 si formano, combinali in un bucket di dimensione 8, e così via.

In coclusione, poiché ci sono al massimo $log_2(N)$ diverse dimensioni di bucket, l'effetto a cascata è limitato e il tempo necessario per processare un nuovo bit è $O(log(N))$

> **Note**
>
> Un **albero AVL** (o un qualsiasi albero di ricerca bilanciato come i Red-Black Trees) potrebbe essere una scelta migliore per l'implementazione di **Exp-Buckets**, soprattutto per garantire aggiornamenti efficienti e mantenere i bucket ordinati per timestamp.
>
> ### Pro degli AVL:
>
> 1. **Accesso ordinato**: Mantiene un ordine totale sui bucket (ad esempio, in base ai timestamp). Puoi scorrere i bucket ordinati in $O(\log N)$.

2. **Aggiornamenti efficienti**: Le operazioni di inserimento, eliminazione e rotazione sono tutte $O(\log N)$, garantendo che il sistema rimanga bilanciato anche con flussi di aggiornamenti frequenti.
3. **Facilità di fusione**: Quando un bucket più piccolo deve essere eliminato o fuso con uno più grande, gli AVL ti permettono di accedere al bucket precedente o successivo in modo efficiente.
4. **Memoria limitata**: Non ci sono overhead significativi oltre alla struttura bilanciata, il che è utile per un approccio come $O(\log^2 N)$ richiesto dall'algoritmo.