Università degli Studi di Udine

Dipartimento di Scienze Matematica, Informatiche e Fisiche

Tesi Magistrale in Informatica

Algoritmi e Ragionamento Automatico

Uno studio sugli algoritmi di sketching per la stima della cardinalità

Relatore
Prof. Gabriele Puppis
Università degli Studi di Udine

Laureando Magistrale
Daniele Ferroli

Matricola
137357

Anno Accademico
2024-2025

"Da scrivere"
— Rene Descartes

Contents

Notazioni e concetti preliminari

In questa sezione si raccolgono le notazioni utilizzate nel resto della tesi.

- $\mathcal{U}$: universo degli elementi.
- $S = \langle x_1, \ldots, x_s \rangle$: stream di dati.
- $S_1 \cdot S_2$: concatenazione di due stream.
- $n = |\mathcal{U}|$: dimensione dell'universo.
- $f(a)$: frequenza di $a \in \mathcal{U}$.
- F_k: frequency moments.
- F_0: numero di distinti nella stream.
- $\hat{F}_0$: stima di F_0 prodotta da un algoritmo.
- $\bar{F}_0$: media dei valori veri su R run.
- $\bar{\hat{F}}_0$: media delle stime su R run.
- (ε, δ): parametri di accuratezza; ε è l'errore relativo ammesso e $1 - \delta$ è la probabilità di successo.
- m: memoria dello sketch (tipicamente espressa in bit).
- M: stato interno dell'algoritmo di streaming (lo sketch).
- $\mathcal{K}(\cdot)$: procedura che costruisce uno sketch da una stream.
- V: dominio di uscita di una funzione di hash.
- $h : \mathcal{U} \to V$: funzione di hash.
- $\mathcal{H}$: famiglia di funzioni di hash.
- w: numero di bit del valore hash (se $V = \{0,1\}^w$).
- $\mathcal{S}$: spazio degli stati di uno sketch.
- $\oplus$: operatore di merge tra stati di sketch.
- R: numero di run (ripetizioni) sperimentali.
- σ: deviazione standard campionaria delle stime.
- RE: errore relativo.
- RSE: relative standard error.
- $\mathrm{Bias}(\hat{\theta})$: bias di uno stimatore.
- $\mathrm{Var}(\hat{\theta})$: varianza di uno stimatore.
- $\widehat{\mathrm{Bias}}(\hat{F}_0)$: stima empirica del bias su R run.

- AB: absolute bias.
- RB: relative bias.
- MRE: mean relative error.
- MAE: mean absolute error.
- RMSE: root mean squared error.

1 Introduzione

Random citation [**?**].

2 Background

Il modello che rappresenta i dati in input, a differenza di algoritmi più tradizionali, è chiamato *modello di stream di dati* (data stream model).

2.1 Modello di stream di dati

Nel modello di stream i dati [1] arrivano in modo continuo; la stream può essere potenzialmente infinita. Rispetto all'utilizzo di un database tradizionale, non è possibile accumulare tutto in memoria o su disco e interrogare i dati. Gli elementi devono essere processati al volo oppure vengono persi.

Inoltre, la velocità con cui i dati arrivano non è controllata dal sistema (più stream possono arrivare a velocità e con formati diversi) e lo spazio di memoria disponibile è limitato. Eventuali archivi storici possono esistere, ma non sono pensati per rispondere a query online in tempi ragionevoli.

Iniziamo a definire formalmente gli elementi di una stream e come vengono processati.

Def. 2.1 (Stream di dati)**.** Sia $\mathcal{U}$ un universo di chiavi. Senza perdita di generalità, assumiamo $\mathcal{U} \subseteq \mathbb{N}$. Una *stream di dati* è una sequenza ordinata di elementi

$$S = \langle x_1, x_2, \ldots, x_s \rangle,$$

dove ogni $x_i \in \mathcal{U}$ e s può essere molto grande o non noto a priori.

Per analizzare una stream è utile descriverla tramite le frequenze degli elementi.

Def. 2.2 (Frequenze)**.** Data una stream S, la *frequenza* di un elemento $a \in \mathcal{U}$ è

$$f(a) = |\{i \mid x_i = a\}|.$$

La collezione delle frequenze può essere vista come un vettore $f \in \mathbb{N}^{|\mathcal{U}|}$.

Una volta definita la nozione di frequenza, si specifica il modello di aggiornamento con cui la stream viene osservata. Nel modello della stream dei dati esistono diverse tipologie di modelli [2]. In questa tesi adottiamo il seguente.

Def. 2.3 (Modello *insertion-only*)**.** La stream è una sequenza di aggiornamenti del tipo (a_t, Δ_t), con $a_t \in \mathcal{U}$ e $\Delta_t \geq 0$. Indichiamo con $A_t[j]$ la frequenza dell'elemento j dopo i primi t aggiornamenti; allora

$$A_t[j] = \begin{cases} A_{t-1}[j] + \Delta_t & \text{se } a_t = j, \\ A_{t-1}[j] & \text{altrimenti.} \end{cases}$$

Se la stream è una lista di valori, ogni elemento x_i può essere visto come un aggiornamento $(x_i, 1)$.

Esistono tuttavia modelli più generali. Nel *turnstile* sono ammessi anche aggiornamenti negativi, così che le frequenze possano aumentare o diminuire. Nel modello a *sliding window* si considerano solo gli ultimi W aggiornamenti della stream, scartando i più vecchi. Questi casi esulano dallo scopo della tesi, ma sono citati per completezza.

Fissato il modello, l'obiettivo principale della tesi è stimare la cardinalità dell'insieme dei distinti.

Def. 2.4 (Numero di distinti)**.** Il *numero di distinti* nella stream S è

$$F_0 = |\{a \in \mathcal{U} \mid f(a) > 0\}|.$$

Più in generale, il numero di distinti è un caso particolare di una famiglia di misure note come *frequency moments*.

Def. 2.5 (Frequency moments)**.** Per ogni $k \geq 0$, il *frequency moment* F_k è definito come

$$F_k = \sum_{a \in \mathcal{U}} f(a)^k.$$

In particolare, F_0 corrisponde al numero di distinti.

Poiché in streaming non possiamo calcolare esattamente F_0 mantenendo memoria sublineare, adottiamo misure di accuratezza probabilistiche. Per valutare la qualità di una stima si introduce la nozione di approssimazione con parametri di accuratezza e confidenza.

Def. 2.6 ((ε, δ)-approssimazione)**.** Un algoritmo A è detto (ε, δ)-*approssimante* per F_0 se, per ogni stream, produce una stima $\hat{F}_0$ tale che

$$\Pr\big(|\hat{F}_0 - F_0| \leq \varepsilon F_0\big) \geq 1 - \delta,$$

dove la probabilità è rispetto alla randomizzazione interna dell'algoritmo [3].

Esempio. Con $\varepsilon = 0{,}05$ e $\delta = 0{,}01$, l'algoritmo deve restituire una stima entro il 5% da F_0 con probabilità almeno 99%.

Supponiamo inoltre che l'universo abbia dimensione $n = |\mathcal{U}|$ e che ogni elemento x_i richieda b bit per essere rappresentato.

Le garanzie di accuratezza devono convivere con vincoli stringenti di tempo e memoria. In questo contesto, un algoritmo di streaming deve:

- processare ciascun elemento con costo $O(1)$ o quasi costante;
- usare memoria molto più piccola di $|\mathcal{U}|$;
- produrre una stima $\hat{F}_0$ con errore controllato, utilizzando m bit, dove $m \ll n$.

Per rispettare questi vincoli si ricorre a funzioni hash che approssimano una distribuzione uniforme sugli elementi e a strutture compatte, chiamate **sketch**, che riassumono le informazioni essenziali della stream senza conservarla esplicitamente.

Il vincolo più forte è quello di memoria: si richiede che lo spazio cresca molto più lentamente della dimensione dell'universo.

Def. 2.7 (Spazio sublineare)**.** Un algoritmo usa *spazio sublineare* se la memoria m cresce asintoticamente meno di n, cioè $m = o(n)$, dove $n = |\mathcal{U}|$. Si richiede che m dipenda in modo polilogaritmico da n e polinomiale da $1/\varepsilon$ e $\log(1/\delta)$.

Esempio. Per il problema dei distinti esistono algoritmi che ottengono una $(1 \pm \varepsilon)$-approssimazione usando $O(\varepsilon^{-2} + \log n)$ bit [4], che è molto meno dei $\Omega(n)$ bit necessari per memorizzare l'insieme dei distinti.

2.2 Algoritmi di streaming

Il modello di data stream, con le caratteristiche appena descritte—flussi potenzialmente infiniti, velocità non controllata e memoria limitata—rende inapplicabili gli approcci classici basati su memorizzazione completa e analisi a posteriori.

Gli algoritmi di streaming nascono per produrre stime e statistiche utili durante l'arrivo dei dati, lavorando in un solo passaggio e mantenendo una sintesi compatta della stream.

Def. 2.8 (Algoritmo di streaming)**.** Un algoritmo di streaming elabora una stream in un solo passaggio e mantiene uno stato interno M di dimensione limitata m. Per ogni elemento x_i della stream, lo stato viene aggiornato tramite una funzione

$$M \leftarrow \mathrm{Update}(M, x_i),$$

e in qualunque momento è possibile ottenere una risposta (o stima) tramite

$$\hat{F}_0 \leftarrow \mathrm{Query}(M).$$

Nel modello classico si richiede che m sia sublineare rispetto a n e che il tempo per aggiornamento sia $O(1)$ o quasi costante [2].

Questa astrazione separa in modo netto il costo di aggiornamento per elemento dalla qualità della stima ottenuta interrogando lo stato compatto.

Dopo ogni aggiornamento, l'elemento appena osservato può essere scartato: lo stato M rappresenta una sintesi compatta dei dati, spesso chiamata *sketch* [5].

La pipeline concettuale del processo di stima è mostrata in Figura 2.1.

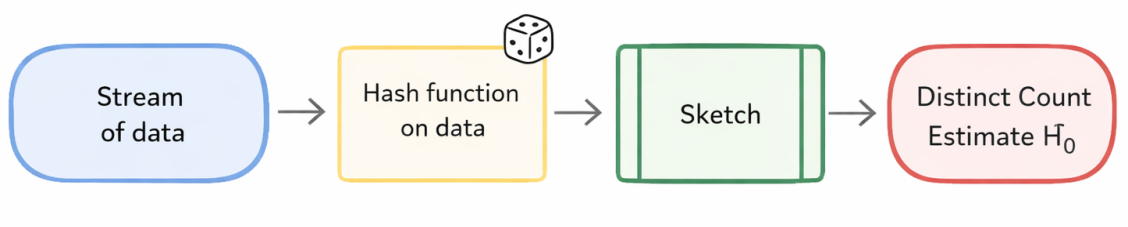

Figure 2.1: Pipeline del modello di algoritmi di streaming

Da qui derivano le metriche standard con cui la letteratura confronta gli algoritmi di streaming. In particolare, nel modello classico le prestazioni si misurano in termini di **passaggi** sulla stream, **memoria** usata, **tempo per elemento** e **accuratezza** della risposta. Per algoritmi di approssimazione, l'accuratezza è espressa tramite un rapporto di approssimazione e una probabilità di successo, spesso nel modello (ε, δ) [6].

Per collocare questo modello nel panorama più ampio degli algoritmi che processano input incompleti, è utile confrontarlo con il modello online. Esiste una tipologia di algoritmi, chiamata algoritmi *online* che sono molto simili [7], perché operano senza disporre dell'intero input; tuttavia non sono identici, poiché nel modello streaming è possibile talvolta differire l'azione fino all'arrivo di piccoli blocchi di elementi, pur mantenendo una memoria molto limitata [2, 6]. Un possibile esempio è fornito dagli algoritmi per la *sliding window*, che mantengono riassunti a blocchi per stimare statistiche recenti con memoria limitata [8]. Nel nostro contesto, tutti gli algoritmi implementati e trattati sono anche *online*, perché processano ogni elemento appena arriva, senza differire l'azione.

Nel contesto degli algoritmi di streaming, un tema centrale è il trade-off tra precisione e memoria. Per il problema del calcolo degli elementi distinti, la letteratura evidenzia un legame diretto tra accuratezza e spazio: ridurre ε implica un incremento della memoria necessaria. In particolare, si considerano efficienti gli algoritmi che usano solo spazio polinomiale in $1/\varepsilon$ e logaritmico nella lunghezza della stream e nella dimensione dell'universo, con un costo per elemento molto basso [3]. Questo mette in evidenza il **trade-off** centrale tra precisione e spazio necessario dello sketch.

Questi algoritmi sono spesso randomizzati: la probabilità nella definizione di (ε, δ) è rispetto alle scelte casuali interne dell'algoritmo e rappresenta una garanzia probabilistica sulla qualità della stima [3]. Nelle implementazioni pratiche, questa randomizzazione è tipicamente incarnata dalla funzione di hash, assunta sufficientemente vicina a una scelta casuale (o parametrizzata da un seed). La randomizzazione consente di ridurre drasticamente lo spazio rispetto alle soluzioni deterministiche, a patto di accettare un errore controllato con alta probabilità.

Un'altra differenza rilevante rispetto all'analisi tradizionale è la distinzione tra stime in tempo reale e analisi *offline*. Nei sistemi classici, gli aggiornamenti si registrano in un archivio e le analisi complesse vengono svolte in *warehouse*. Nel modello di streaming, invece, molte applicazioni richiedono elaborazioni sofisticate in quasi tempo reale, come rilevamento di anomalie, monitoraggio di trend o cambiamenti improvvisi, e questo condiziona la progettazione degli algoritmi [2].

È importante notare che in contesti distribuiti è spesso necessario combinare riassunti di porzioni diverse della stream. Il concetto di *mergeabilità* formalizza la possibilità di unire due sintesi in una sintesi della loro unione preservando le garanzie di errore e la dimensione dello stato: questo permette di scalare gli algoritmi a scenari paralleli o gerarchici ed è una proprietà centrale per gli sketch moderni [9]. Nel seguito questa proprietà verrà formalizzata a livello di struttura dati (*sketch*) e di operatore di composizione.

2.3 Funzioni di hash

Le funzioni hash sono il principale strumento per "randomizzare" lo stream e associare un universo molto grande in un dominio più piccolo, rendendo possibile l'uso di strutture compatte. In molti sketch, la qualità della stima dipende direttamente dalle proprietà della funzione di hash utilizzata [10, 11]. Gli sketch trattati in questa tesi trasformano infatti le chiavi in valori pseudo-casuali e poi estraggono statistiche semplici: per questo motivo, la qualità dell'hash entra direttamente nell'analisi dell'errore.

Def. 2.9 (Funzione di hash)**.** Una funzione di hash h è una funzione deterministica

$$h : \mathcal{U} \to V,$$

che associa a ogni chiave dell'universo $\mathcal{U}$ un valore in un dominio V di dimensione molto più piccola, tipicamente $V = \{0,1\}^w$ oppure $V = [0,1)$ tramite normalizzazione.

Quindi, una funzione di hash mappa dati di lunghezza arbitraria in un valore di lunghezza fissa, chiamato *hash value*, spesso usato per indicizzare delle strutture dati come le tabelle di hash.

Questa trasformazione permette accessi veloci e riduce lo spazio necessario rispetto alla memorizzazione diretta delle chiavi.

Poiché più chiavi possono produrre lo stesso valore, le *collisioni* sono inevitabili: una buona funzione di hash deve essere veloce da calcolare e minimizzare la probabilità di collisione, idealmente distribuendo i valori in modo uniforme sul dominio [12, 10, 11]. In questo senso, uniformità e collisioni descrivono lo stesso fenomeno da due angoli: collisioni sistematiche indicano non-uniformità delle associazioni.

La Figura 2.2 mostra in modo intuitivo la mappatura chiavi→bucket realizzata da una funzione di hash.

Proprietà desiderate. Le proprietà classiche richieste sono: l'**uniformità** (le chiavi sono distribuite in modo uniforme su V), la **bassa probabilità di collisione**, l'**indipendenza** tra le immagini di chiavi diverse e l'**efficienza** di calcolo.

Una funzione di hash con queste proprietà rende la stima degli sketch stabili e con varianza controllata [10, 11].

Def. 2.10 (Modello di hashing uniforme)**.** Nel modello ideale, h è scelta uniformemente a caso dall'insieme di tutte le funzioni $\mathcal{U} \to V$. In questo caso, per ogni chiave $x \in \mathcal{U}$, il valore $h(x)$ è uniforme in V e le immagini di chiavi distinte sono indipendenti [10, 11].

Def. 2.11 (Famiglia universale [12])**.** Una famiglia $\mathcal{H}$ di funzioni $\mathcal{U} \to V$ è *universale* se, per ogni coppia di chiavi distinte $x \neq y$, vale

$$\Pr_{h \leftarrow \mathcal{H}}\big[h(x) = h(y)\big] \leq \frac{1}{|V|}.$$

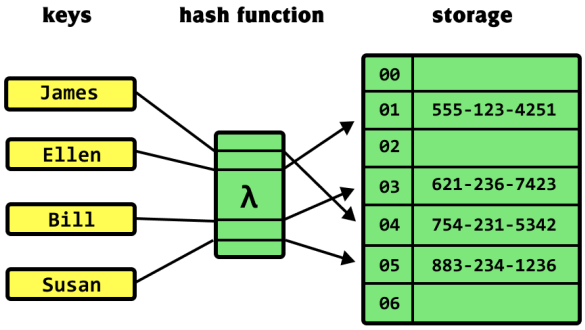

Figure 2.2: Schema di un esempio di tabella di hash

Def. 2.12 (k-wise indipendenza [10])**.** Una famiglia $\mathcal{H}$ è *k-wise indipendente* se, per ogni scelta di k chiavi distinte $x_1, \ldots, x_k$, il vettore

$$\big(h(x_1), \ldots, h(x_k)\big)$$

è distribuito uniformemente in V^k quando h è scelta a caso da $\mathcal{H}$.

Assunzioni tipiche. Nelle analisi teoriche si assume spesso il modello ideale di hashing uniforme; in alternativa si usa una famiglia con un grado limitato di indipendenza (ad esempio k-wise), o una famiglia universale [12, 10]. In pratica, l'uso di funzioni semplici è comune, ma le garanzie possono degradare rispetto al modello ideale se le assunzioni non sono soddisfatte.

Impatto delle scelte. Una funzione di hash non sufficientemente uniforme può introdurre collisioni sistematiche o correlazioni tra registri, con un aumento del bias e della varianza degli stimatori [10, 11].

2.4 Sketch

Uno *sketch* è una struttura dati probabilistica che riassume una stream attraverso uno stato M aggiornabile con operazioni *update* e interrogabile con operazioni *query*. Lo sketch non conserva gli elementi originali, ma solo le informazioni necessarie per stimare una quantità d'interesse con memoria limitata che deve essere molto inferiore rispetto a conservare i dati originali.

Come accennato prima negli algoritmi di streaming, nei contesti distribuiti, per ottenere un dato globale è necessario poter combinare due sketch costruiti su porzioni diverse della stream. Come possibile esempio, si pensi a due server che creano in maniera indipendente il loro sketch dei dati e che si voglia unire queste informazioni. Intuitivamente, ciò è possibile solo se gli sketch condividono gli stessi parametri strutturali e la stessa funzione di hash (o lo stesso seed), altrimenti la fusione può degradare le garanzie.

Def. 2.13 (Sketch mergeable)**.** Sia $\mathcal{K}(\cdot)$ la procedura che costruisce uno sketch e sia $\oplus$ un operatore sullo stato. Lo sketch è *mergeable* se, per due stream S_1, S_2 costruite con la stessa parametrizzazione e la stessa funzione di hash,

vale

$$\mathcal{K}(S_1) \oplus \mathcal{K}(S_2) \approx \mathcal{K}(S_1 \cdot S_2),$$

preservando le garanzie di errore (o con degradazione nota) [9].

Nel caso specifico del count-distinct, la concatenazione induce la stessa informazione rilevante dell'unione insiemistica delle chiavi osservate.

Def. 2.14 (Operatore chiuso)**.** Sia $\mathcal{S}$ lo spazio degli stati di uno sketch. Un operatore $\oplus$ è *chiuso* se

$$\oplus : \mathcal{S} \times \mathcal{S} \to \mathcal{S},$$

cioè se la combinazione di due stati produce ancora uno stato valido dello stesso tipo.

La Figura 2.3 illustra lo scenario tipico in cui più nodi producono sketch locali che vengono aggregati gerarchicamente.

In pratica, l'operatore $\oplus$ deve essere *chiuso* sullo stato (ad esempio utilizzando la funzione di massimo per registro o la funzione di somma per componente) e, per aggregazioni robuste, è preferibile che sia *commutativo* e *associativo*; in molti casi è utile anche l'*idempotenza* per tollerare duplicazioni.

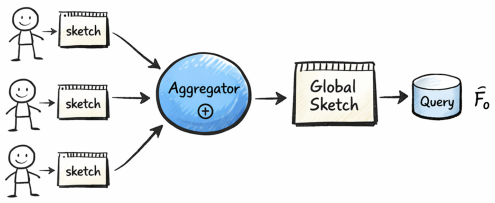

Figure 2.3: Merge di sketch in un contesto distribuito

Osservazione: Per gli sketch a registri considerati in questa tesi (LogLog, HLL e HLL++), se due sketch hanno stessi parametri e stessa funzione di hash/seed, l'operatore di merge è il massimo componente-per-componente. Con queste condizioni, lo sketch ottenuto dal merge coincide con quello costruito processando la concatenazione delle due stream.

2.5 Stimatori

In statistica, uno *stimatore* è una funzione dei dati osservati che restituisce una stima di un parametro d'interesse. Nel contesto dello streaming dei dati, la stima dipende sia dalla stream osservata sia dalla randomizzazione interna dell'algoritmo (ad esempio la funzione di hash) [13]. Nel seguito, la stima $\hat{F}_0$ prodotta da uno sketch viene quindi trattata come variabile aleatoria e descritta con il lessico standard della stima statistica.

Adesso andremo a definire alcuni concetti essenziali affinché sia possibile valutare la qualità di una stima e confrontare diversi algoritmi.

Def. 2.15 (Stimatore)**.** Sia θ un parametro d'interesse e siano X i dati osservati. Uno *stimatore* è una funzione misurabile T tale che

$$\hat{\theta} = T(X),$$

dove $\hat{\theta}$ è una variabile aleatoria [13].

Def. 2.16 (Bias e correttezza)**.** Il *bias* di uno stimatore è

$$\mathrm{Bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta.$$

Lo stimatore è *corretto* (unbiased) se il bias è nullo; altrimenti è *biased* [13].

Un bias positivo indica una tendenza sistematica a sovrastimare il valore vero, mentre un bias negativo indica una sottostima. Un bias nullo non garantisce stima precisa in ogni run, ma elimina lo spostamento medio.

Def. 2.17 (Varianza ed errore standard)**.** La *varianza* di uno stimatore è

$$\mathrm{Var}(\hat{\theta}) = \mathbb{E}\big[(\hat{\theta} - \mathbb{E}[\hat{\theta}])^2\big],$$

mentre l'*errore standard* è la sua radice quadrata $\mathrm{SE}(\hat{\theta}) = \sqrt{\mathrm{Var}(\hat{\theta})}$ [13].

La varianza descrive quanto le stime oscillano tra esecuzioni: valori piccoli indicano stabilità, valori grandi indicano dispersione. L'errore standard è una misura nella stessa unità di θ e fornisce una scala naturale della variabilità.

Def. 2.18 (Errore di stima e rischio)**.** L'*errore di stima* è la variabile aleatoria

$$E = \hat{\theta} - \theta.$$

Dato una funzione di *loss* $L(E)$, il *rischio* (o rischio atteso) è

$$R(\hat{\theta}) = \mathbb{E}[L(E)].$$

Delle scelte comuni per la funzione di loss sono $L(E) = |E|$ (rischio assoluto) e $L(E) = E^2$ (MSE) [13].

L'errore di stima misura lo scostamento puntuale dalla verità, mentre il rischio riassume l'errore atteso secondo una loss scelta. Loss diverse privilegiano aspetti diversi: l'errore assoluto è più robusto, l'errore quadratico penalizza maggiormente gli scostamenti grandi.

Def. 2.19 (MAE e MSE)**.** Il *Mean Absolute Error* (MAE) è

$$\mathrm{MAE} = \mathbb{E}\big[|\hat{\theta} - \theta|\big],$$

mentre il *Mean Squared Error* (MSE) è

$$\mathrm{MSE} = \mathbb{E}\big[(\hat{\theta} - \theta)^2\big].$$

Il MAE misura l'errore medio assoluto, mentre l'MSE penalizza maggiormente gli errori grandi [13].

Il MAE è facilmente interpretabile come distanza media dalla verità. L'MSE (e la sua radice, l'RMSE) enfatizza gli errori grandi e quindi è sensibile a outlier o code pesanti nelle stime.

Def. 2.20 (Errore relativo)**.** L'*errore relativo* è definito come

$$\mathrm{RE} = \frac{|\hat{\theta} - \theta|}{|\theta|},$$

quando $\theta \neq 0$.

L'errore relativo normalizza lo scarto rispetto alla grandezza del vero valore e rende confrontabili stime su dataset di scala diversa.

Def. 2.21 (Consistenza)**.** Una sequenza di stimatori $\hat{\theta}_n$ è *consistente* se

$$\hat{\theta}_n \xrightarrow{P} \theta \quad \text{quando } n \to \infty,$$

ossia se la stima converge al valore vero al crescere della quantità di informazione disponibile [13].

Nei capitoli successivi si parlerà anche di **bias correction**, cioè di tecniche che riducono lo spostamento medio della stima tramite calibrazione empirica o aggiustamenti analitici delle formule. Queste tecniche non eliminano necessariamente la varianza, ma migliorano l'accuratezza media.

2.6 Metriche di errore

A seguire vengono definite le metriche usate per valutare empiricamente gli stimatori del Capitolo 2.5. Vengono fatte R esecuzioni indipendenti, chiamate *run*, con stime $\hat{F}_0^{(r)}$ e valori veri $F_0^{(r)}$.

Per una singola run r vengono calcolati l'errore di stima, il suo valore assoluto e l'errore relativo:

$$e^{(r)} = \hat{F}_0^{(r)} - F_0^{(r)}, \qquad |e^{(r)}|, \qquad \mathrm{RE}^{(r)} = \frac{|e^{(r)}|}{F_0^{(r)}} \ (\text{se } F_0^{(r)} > 0).$$

Successivamente, vengono calcolati il bias, la varianza, l'errore standard e per collegarsi alla letteratura sugli sketch, si usa la *Relative Standard Error* (RSE), che normalizza la deviazione standard rispetto alla scala del problema.

Siccome nel nostro framework le run sono eseguite su sottoinsiemi potenzialmente diversi, si introduce la media degli stimatori per il numero di elementi distinti:

$$\bar{F}_0 = \frac{1}{R}\sum_{r=1}^{R} F_0^{(r)}, \qquad \bar{\hat{F}}_0 = \frac{1}{R}\sum_{r=1}^{R} \hat{F}_0^{(r)}.$$

La *varianza campionaria* delle stime è

$$Var(\hat{F}_0) = \frac{1}{R-1}\sum_{r=1}^{R}\big(\hat{F}_0^{(r)} - \bar{\hat{F}}_0\big)^2, \qquad \sigma = \sqrt{Var}.$$

Come per l'errore, viene stimato il bias, il suo valore assoluto e il suo errore relativo:

$$\widehat{\mathrm{Bias}}(\hat{F}_0) = \bar{\hat{F}}_0 - \bar{F}_0, \qquad \mathrm{AB} = |\widehat{\mathrm{Bias}}(\hat{F}_0)|, \qquad \mathrm{RB} = \frac{\widehat{\mathrm{Bias}}(\hat{F}_0)}{\bar{F}_0} \ (\bar{F}_0 \neq 0).$$

L'*errore relativo medio* empirico (Mean Relative Error, MRE) è

$$\mathrm{MRE} = \frac{1}{R}\sum_{r=1}^{R} \frac{|\hat{F}_0^{(r)} - F_0^{(r)}|}{F_0^{(r)}},$$

e le metriche aggregate empiriche usate nei risultati includono il Mean Absolute Error (MAE) e il Root Mean Squared Error (RMSE):

$$\mathrm{MAE} = \frac{1}{R}\sum_{r=1}^{R} |\hat{F}_0^{(r)} - F_0^{(r)}|, \qquad \mathrm{RMSE} = \sqrt{\frac{1}{R}\sum_{r=1}^{R} \big(\hat{F}_0^{(r)} - F_0^{(r)}\big)^2}.$$

Queste quantità sono stime empiriche dei corrispondenti concetti teorici presentati nella sezione precedente. La *RSE osservata* è stimata come

$$\mathrm{RSE}_{\mathrm{obs}} = \frac{\sigma}{\bar{F}_0}.$$

Quando il valore vero è lo stesso in tutte le run ($F_0^{(r)} \equiv F_0$), questa definizione si riduce a σ/F_0. Quando disponibile, si confronta l'RSE osservata con una *RSE teorica* fornita dalla letteratura (ad esempio formule del tipo $c/\sqrt{m}$), per verificare la coerenza tra predizioni teoriche e risultati sperimentali. Quando invece $F_0^{(r)}$ varia tra run, $\mathrm{RSE}_{\mathrm{obs}}$ va interpretata come normalizzazione rispetto alla scala media del problema, e non rispetto a un unico valore vero fisso.

2.7 Famiglia di algoritmi per count-distinct

Il calcolo esatto del numero di distinti F_0 in streaming richiede, nel caso generale, memoria proporzionale al numero di chiavi distinte osservate, poiché occorre mantenere informazione sufficiente a distinguere chiavi già viste da chiavi nuove. Gli algoritmi di sketching rinunciano all'esattezza e producono una stima $\hat{F}_0$ con garanzie probabilistiche (ε, δ) usando spazio sublineare e update a costo costante [3].

Una linea storica fondamentale per il count-distinct parte dal *Probabilistic Counting* di Flajolet–Martin [14], che usa una funzione di hash per trasformare le chiavi in bitstring pseudo-casuali e ricavare F_0 da statistiche sui pattern di bit. In tutti questi approcci, l'hash rende i bit osservati simili a campioni casuali e la cardinalità viene ricostruita da statistiche di rarità, come pattern rari o sequenze lunghe di zeri. Successivamente, LogLog [15] e HyperLogLog [16] introducono una struttura a m registri (ottenuti partizionando via hash), aggiornata tramite il numero di zeri iniziali: l'aggregazione su più registri riduce la varianza e porta a un errore relativo che decresce tipicamente come $O(1/\sqrt{m})$. HyperLogLog++ [17] mantiene l'impianto a registri ma introduce correzioni e accorgimenti pratici (ad esempio per il small-range) che riducono il bias e migliorano l'accuratezza su diverse scale di cardinalità.

Questi algoritmi condividono inoltre una proprietà operativa importante: essendo lo stato basato su aggior-

namenti monotoni per registro, la fusione di sketch costruiti su partizioni disgiunte della stream è naturale (ad esempio tramite massimo componente-per-componente), rendendo tali metodi adatti a scenari distribuiti [16, 9]. Il Capitolo 3 riprende questa linea evolutiva confrontando, per ogni algoritmo, la struttura dello sketch e i meccanismi di correzione dell'errore.

2.8 Spazio e accuratezza

Analizzando algoritmi di streaming per il count-distinct, è importante caratterizzare il rapporto tra spazio e accuratezza. Senza entrare nei dettagli di uno specifico algoritmo, esistono limiti teorici generali che guidano la progettazione degli sketch.

Utilizzando il modello probabilistico (ε, δ) per la qualità della stima, esistono degli algoritmi di streaming che producono una $(1 \pm \varepsilon)$-approssimazione di F_0 con probabilità almeno $1 - \delta$ usando spazio $\tilde{O}\big(\varepsilon^{-2} \log(1/\delta) + \log n\big)$ bit, dove $n = |\mathcal{U}|$ e $\tilde{O}$ nasconde fattori polilogaritmici [18, 4].

Inoltre, esistono lower bound dello stesso ordine di grandezza, per cui tale dipendenza è ottimale a livello di ordine [4].

Di conseguenza, ridurre ε di un fattore 2 richiede circa 4 volte la memoria, poiché lo spazio cresce come ε^{-2}.

Mentre, ridurre δ richiede solo un fattore logaritmico, ottenibile tramite ripetizioni indipendenti e combinazione (ad esempio *median trick* [19]).

Questo andamento rende naturale ottimizzare soprattutto la dipendenza da ε, mentre δ viene spesso gestito tramite amplificazione della probabilità di successo.

Nel caso di HyperLogLog, la deviazione standard relativa (RSE) scala come $\Theta(1/\sqrt{m})$, dove m è il numero di registri; in altre parole, raddoppiare m riduce l'errore di un fattore circa $\sqrt{2}$ [16]. Nei capitoli sperimentali questa relazione verrà verificata empiricamente variando m e osservando l'andamento di RSE.

3

Un'analisi sullo stato dell'arte

Nel capitolo 2 abbiamo definito le nozioni necessarie collegate agli *algoritmi di streaming*, in questo capitolo analizzeremo lo *stato dell'arte* degli algoritmi per la stima di F_0 (il numero di elementi distinti). L'obiettivo è ricostruire la linea evolutiva degli approcci classici e chiarire quali scelte algoritmiche portano ai migliori compromessi tra memoria, accuratezza e componibilità distribuita.

3.1 Obiettivo del problema e vincoli teorici

Il problema del *count-distinct* consiste nello stimare

$$F_0 = |\{a \in \mathcal{U} : f(a) > 0\}|,$$

ossia la cardinalità del supporto del vettore delle frequenze. Nel quadro dei *frequency moments*, F_0 rappresenta il primo problema canonico di stima con memoria sublineare in streaming [18, 3].

In un modello *insertion-only*, un algoritmo deve processare ogni elemento in uno o pochi passaggi, con tempo di aggiornamento molto basso e stato interno compatto. In generale, il calcolo esatto richiede memoria lineare nel numero di distinti osservati, di conseguenza non può scalare all'aumentare degli elementi visti. In un contesto di dati in streaming, cioé che non c'è un limite di elementi, questa soluzione non può funzionare per questo motivo si ricorre a utilizzare sketch randomizzati che seguono il modello di *streaming probabilistico*, che segue il modello (ε, δ).

Dal lato teorico, esistono algoritmi ottimali per il problema dei distinti che raggiungono spazio dell'ordine $O(\varepsilon^{-2} + \log n)$ (a fattori logaritmici noti), mostrando che la dipendenza da ε^{-2} è strutturale e non un artefatto implementativo [4]. Questa cornice giustifica l'uso di famiglie algoritmiche in cui l'accuratezza cresce come $\Theta(1/\sqrt{m})$, dove m è la memoria effettiva dello sketch.

3.2 Sviluppo storico

La progressione storica degli algoritmi di cardinalità può essere vista come una sequenza di miglioramenti sulla stessa idea centrale: usare una funzione di hash per trasformare la stream di dati in valori pseudo-casuali e comprimere l'informazione in un unico stato di dimensioni molto ridotte.

3.2.1 Probabilistic Counting

Il lavoro in [14] introduce il paradigma del conteggio probabilistico: da ogni chiave si ricava un valore di hash e si osserva la posizione del primo bit significativo (oppure il numero di zeri iniziali). Gli eventi rari, come molti zeri iniziali, diventano indicatori della scala di F_0.

La formulazione di base mantiene una bitmap e marca posizioni osservate; la successiva variante, chiamata *Probabilistic Counting with Stochastic Averaging* (**PCSA**), partiziona la stream in più sottostream indipendenti tramite hash, riducendo la varianza rispetto a una singola statistica globale.

Algorithm 3.1 PCSA

Scegliere una hash $h : \mathcal{U} \to \{0,1\}^L$
Inizializzare m bitmap $B_1, \ldots, B_m$ a zero
for ogni elemento x della stream
  $y \leftarrow h(x)$
  $j \leftarrow$ indice sottostream (da un prefisso di y)
  $w \leftarrow$ bit rimanenti di y
  $r \leftarrow \rho(w)$ (posizione del primo 1, o numero di zeri iniziali $+1$)
  Porre a 1 il bit r nella bitmap B_j
end for
for $j = 1$ **to** m
  $R_j \leftarrow$ indice del primo 0 nella bitmap B_j
end for
Aggregare $R_1, \ldots, R_m$ e applicare la costante di calibrazione del paper
Restituire la stima $\hat{F}_0$

Nel paper originale vengono anche discussi bias, errore standard e modalità di uso pratico (numero di bitmap, correzioni per piccoli range, parallelizzazione) [14].

3.2.2 LogLog

LogLog sostituisce la bitmap con un array di registri e applica *stochastic averaging* in modo più efficiente: il prefisso hash seleziona il registro, mentre il suffisso determina il valore ρ da propagare come massimo. La stima finale usa una media geometrica normalizzata [15].

Algorithm 3.2 Schema LogLog (adattato da [15])

Scegliere precisione p e porre $m = 2^p$
Inizializzare i registri $M[0], \ldots, M[m-1] \leftarrow 0$
for ogni elemento x della stream
  $y \leftarrow h(x)$
  $j \leftarrow$ primi p bit di y
  $w \leftarrow$ bit rimanenti di y
  $M[j] \leftarrow \max\{M[j], \rho(w)\}$
end for
Calcolare la media geometrica dei registri e applicare la normalizzazione del paper
Restituire $\hat{F}_0$

Il passaggio chiave rispetto a FM è che, a parità di memoria, la dispersione della stima si riduce sensibilmente grazie all'aggregazione su molti registri.

3.2.3 HyperLogLog

HyperLogLog (HLL) mantiene la stessa struttura a registri di LogLog ma cambia la funzione di stima: usa una media armonica delle quantità $2^{-M[j]}$, ottiene una migliore analizzabilità e una costante di errore più favorevole. Il risultato classico è una deviazione standard relativa tipica $\mathrm{RSE} \approx 1.04/\sqrt{m}$ [16].

L'articolo del 2007 fornisce sia l'analisi asintotica, sia la variante pratica con correzioni di range, che costituisce il riferimento per implementazioni sperimentali riproducibili.

3.2.4 HyperLogLog++

HLL++ nasce come evoluzione ingegneristica di HLL in scenari industriali ad alta scala. I miglioramenti principali sono:

- uso di hash a 64 bit per ritardare gli effetti di saturazione;
- rappresentazione *sparse* per cardinalità piccole;
- bias correction empirica tramite tabelle/interpolazione;
- scelta adattiva tra linear counting e stima HLL corretta.

Nel lavoro del 2013, il contributo non è una nuova famiglia teorica separata, bensì una rifinitura sistematica di HLL per ridurre il bias pratico e migliorare l'accuratezza nelle cardinalità piccole e intermedie [17].

3.3 Confronto sintetico tra gli approcci

La Tabella 3.1 riassume le differenze principali tra gli algoritmi della linea storica.

Algorithm 3.3 Schema HLL pratico (adattato da [16])

Scegliere precisione p, porre $m = 2^p$, inizializzare $M[0..m-1] \leftarrow 0$
for ogni elemento x della stream
  $y \leftarrow h(x)$
  $j \leftarrow$ prefisso di p bit di y
  $w \leftarrow$ suffisso di y
  $M[j] \leftarrow \max\{M[j], \rho(w)\}$
end for
$Z \leftarrow \sum_{j=0}^{m-1} 2^{-M[j]}$
$E \leftarrow \alpha_m m^2 / Z$ (stima grezza)
$V \leftarrow$ numero di registri a zero
if $E \leq \frac{5}{2}m$ e $V > 0$
  $E^\star \leftarrow m \log(m/V)$ (small-range, linear counting)
else if E è vicino al limite del dominio hash
  Applicare la large-range correction del paper
else
  $E^\star \leftarrow E$
end if
Restituire $\hat{F}_0 \leftarrow E^\star$

Algoritmo	Stato dello sketch	Regola di stima	Ordine errore
FM/PCSA	Bitmap (una o più)	Primo zero / media su bitmap	$\Theta(1/\sqrt{m})$
LogLog	Array registri (max di ρ)	Media geometrica normalizzata	$\Theta(1/\sqrt{m})$
HyperLogLog	Array registri (max di ρ)	Media armonica + range corrections	$\approx 1.04/\sqrt{m}$
HyperLogLog++	Registri + formato sparse	HLL + bias correction + adaptive switching	$\Theta(1/\sqrt{m})$ con minore bi

Table 3.1: Confronto ad alto livello tra i principali algoritmi count-distinct.

Algorithm 3.4 Schema HLL++ (adattato da [17])

Inizializzare struttura in formato *sparse*
for ogni elemento x della stream
  $y \leftarrow h_{64}(x)$
  Estrarre coppia (j, ρ) da y
  if formato sparse
    Inserire/aggiornare la coppia codificata nella struttura sparsa
    if la struttura supera la soglia di memoria
      Convertire in formato dense (registri)
    end if
  else
    $M[j] \leftarrow \max\{M[j], \rho\}$
  end if
end for
if formato sparse e cardinalità stimata piccola
  Usare stima di linear counting
else
  Calcolare stima HLL grezza su registri
  Applicare bias correction empirica
  Applicare eventuali correzioni di range
end if
Restituire $\hat{F}_0$

La traiettoria evolutiva è quindi chiara: dalla robustezza concettuale di FM si passa a strutture a registri sempre più stabili, fino a HLL/HLL++, che rappresentano oggi il punto di riferimento pratico per il rapporto spazio-accuratezza.

3.4 Correzioni di range e riduzione del bias

Nel confronto tra algoritmi non basta riportare una formula di stima grezza; conta anche la gestione dei regimi in cui la formula asintotica non è ancora stabile.

Per HLL, il riferimento classico distingue almeno tre zone [16]:

- **small-range**: uso di *linear counting* quando molti registri restano a zero;
- **raw-range**: uso dell'estimatore armonico principale;
- **large-range**: correzione per la vicinanza al limite del dominio hash.

HLL++ mantiene questa logica ma aggiunge una correzione empirica del bias e una gestione sparse/dense che riduce l'errore nei range in cui HLL classico tende a sovrastimare [17].

In generale, il principio metodologico è che il comportamento empirico deve essere confrontato con la teoria: quando la letteratura fornisce una RSE theoretical (ad esempio $1.04/\sqrt{m}$), le misure sperimentali vanno lette in quella cornice e non isolate.

3.5 Mergeabilità e scenari distribuiti

Una proprietà fondamentale degli sketch di cardinalità moderni è la *mergeabilità*: la possibilità di costruire sketch locali su partizioni della stream e combinarli in uno sketch globale senza dover rivedere i dati originali [9].

Per LogLog, HLL e HLL++, la regola di merge naturale è il massimo componente-per-componente dei registri:

$$(M_1 \oplus M_2)[j] = \max\{M_1[j], M_2[j]\}.$$

Questa operazione è commutativa, associativa e idempotente, quindi è adatta a pipeline distribuite (albero, catena, map-reduce). Inoltre, a parità di parametri e funzione hash/seed, il risultato del merge è equivalente allo stato che si otterrebbe processando sequenzialmente l'unione delle stream.

Per FM/PCSA, la mergeabilità dipende dalla codifica dello stato: in forma bitmap la combinazione è una OR componente-per-componente, mentre in forme basate su massimi si usa ancora il massimo per componente [14].

4 Implementazione

4.1 Algoritmi

4.2 Framework di benchmark

Example of Algorithm 4.1 reference.

Algorithm 4.1 Pseudocode

```
i ← 10
if i ≥ 5
    i ← i − 1
else
    if i ≤ 3
        i ← i + 2
    end if
end if
```

5
Risultati sperimentali

Example of Table 5.1, made using `https://www.tablesgenerator.com/`.

A	B
1	2
3	4

Table 5.1: Interesting results.

6
Conclusioni

References

[1] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining Data Streams*. Cambridge University Press, 2014, p. 123–153.

[2] S. Muthukrishnan, "Data streams: Algorithms and applications," Rutgers University, Tech. Rep., 2005.

[3] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *Randomization and Approximation Techniques in Computer Science (RANDOM 2002)*, ser. Lecture Notes in Computer Science, vol. 2483. Springer, 2002, pp. 1–10.

[4] D. M. Kane, J. Nelson, and D. P. Woodruff, "An optimal algorithm for the distinct elements problem," in *Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2010)*. ACM, 2010.

[5] G. Cormode, "Data sketching," *Communications of the ACM*, 2017.

[6] N. Prezza, "Algorithms for massive data – lecture notes," 2025, lecture notes, Ca' Foscari University of Venice.

[7] R. M. Karp, "On-line algorithms versus off-line algorithms: How much is it worth to know the future?" in *IFIP Congress (1)*. World Computer Congress, 1992, pp. 416–429.

[8] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows: (extended abstract)," in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '02. USA: Society for Industrial and Applied Mathematics, 2002, pp. 635–644.

[9] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, "Mergeable summaries," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2012)*. ACM, 2012.

[10] S. Vadhan and M. Mitzenmacher, "Why simple hash functions work: Exploiting the entropy in a data stream," 2007, manuscript.

[11] A. Pagh and R. Pagh, "Uniform hashing in constant time and optimal space," 2007, manuscript.

[12] J. L. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, pp. 143–154, 1979.

[13] S. van de Geer, "Mathematical statistics," 2015, lecture notes, September 2015.

[14] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of Computer and System Sciences*, vol. 31, no. 2, 1985.

[15] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *Proceedings of the European Symposium on Algorithms (ESA 2003)*, 2003.

[16] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *Proceedings of the 2007 Conference on Analysis of Algorithms (AofA 07)*, 2007, pp. 127–146.

[17] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the International Conference on Extending Database Technology (EDBT/ICDT '13)*. ACM, 2013.

[18] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *Journal of Computer and System Sciences*, vol. 58, no. 1, pp. 137–147, 2002.

[19] R. Motwani and P. Raghavan, "Randomized algorithms," *ACM Computing Surveys*, vol. 28, no. 1, March 1996, copyright ŏ0a9 1996, CRC Press.

Acknowledgments