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

Laureando Magistrale
Daniele Ferroli

Matricola
137357

Anno Accademico
2024-2025

"Da scrivere"
— Rene Descartes

Indice

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 $a \in \mathcal{U}$.
- (a_t, Δ_t): aggiornamento al tempo t, con chiave a_t e incremento Δ_t.
- $A_t[j]$: frequenza dell'elemento j dopo i primi t aggiornamenti.
- $A_0[j] = 0$: condizione iniziale del modello insertion-only.
- $f \in \mathbb{N}^{|\mathcal{U}|}$: vettore delle frequenze (componenti $f(a)$).
- $\|f\|_1 = \sum_{a \in \mathcal{U}} f(a)$: lunghezza totale della stream.
- 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); nelle sezioni su LogLog/HLL/HLL++ il numero di registri è indicato anch'esso con $m = 2^p$, quindi lo spazio complessivo dipende anche dalla larghezza di ciascun registro.
- 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.
- L: lunghezza in bit del valore hash (nel Capitolo 3 si assume $L = w$).
- w: numero di bit del valore hash (se $V = \{0,1\}^w$).
- p: parametro di precisione; tipicamente $m = 2^p$.
- p': precisione usata nella rappresentazione sparsa di HLL++.

- k_{sp}: numero di entry non nulle nella rappresentazione sparsa di HLL++.
- $\rho(\cdot)$: posizione del primo bit a 1 nel suffisso hash.
- $j(x)$: indice del registro selezionato dai $\log_2 m$ bit più significativi di $h(x)$.
- $w(x)$: suffisso di $h(x)$ usato per calcolare $\rho(w(x))$ nel registro $j(x)$.
- α_m: costante di normalizzazione dipendente da m.
- ϕ: costante di calibrazione di PCSA ($\phi \approx 0.77351$).
- V_0: numero di registri a zero (in HLL/HLL++).
- w_{cm}: numero di colonne nel Count-Min Sketch.
- d: numero di righe/funzioni hash nel Count-Min Sketch.
- m_{bf}: numero di bit del Bloom filter.
- k_{bf}: numero di funzioni hash del Bloom filter.
- n_{ins}: numero di elementi inseriti in un Bloom filter.
- $\mathcal{S}$: 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.
- $\widehat{\text{Var}}(\hat{F}_0)$: varianza campionaria delle stime su R run.
- $\widehat{\sigma}$: deviazione standard campionaria ($\widehat{\sigma} = \sqrt{\widehat{\text{Var}}(\hat{F}_0)}$).
- RE: errore relativo.
- RSE: relative standard error.
- RSE_{obs}: relative standard error osservata, definita come $\widehat{\sigma}/\bar{F}_0$.
- $\text{Bias}(\hat{\theta})$: bias di uno stimatore.
- $\text{Var}(\hat{\theta})$: varianza di uno stimatore.
- $\widehat{\text{Bias}}(\hat{F}_0)$: 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

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

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

con inizializzazione $A_0[j] = 0$ per ogni $j \in \mathcal{U}$.

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]. La forma relativa è intesa per $F_0 > 0$; nel caso $F_0 = 0$ si richiede, in modo naturale, $\Pr(\hat{F}_0 = 0) \geq 1 - \delta$.

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.

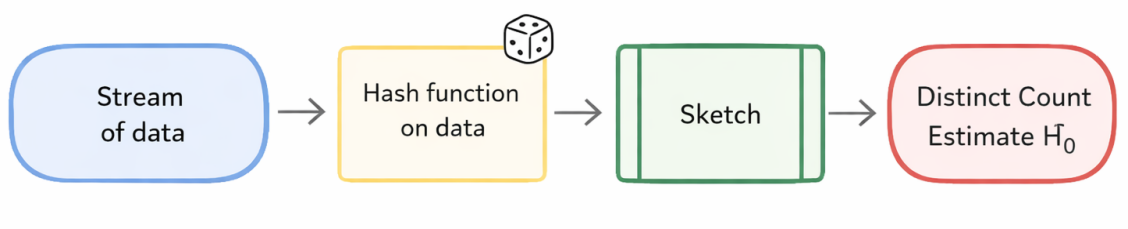

Figura 2.1: Pipeline del modello di algoritmi di streaming

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

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

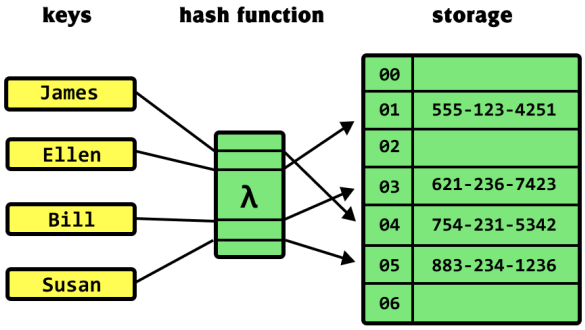

Figura 2.2: Schema di un esempio di tabella di hash

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.

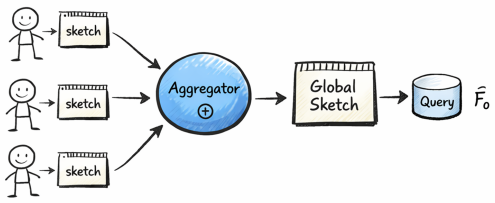

Figura 2.3: Merge di sketch in un contesto distribuito

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 della Sezione 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:

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

Nel seguito distinguiamo la varianza teorica $\mathrm{Var}(\hat{F}_0)$ dalla *varianza campionaria* delle run sperimentali:

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

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

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

Questa formula è valida per $\bar{F}_0 > 0$. Se la cardinalità media è nulla, anche la deviazione standard campionaria risulta nulla e la RSE osservata viene posta convenzionalmente a 0. Quando il valore vero è lo stesso in tutte le run ($F_0^{(r)} \equiv F_0$), questa definizione si riduce a $\widehat{\sigma}/F_0$. 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 aggiornamenti 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 Altri obiettivi di sketching

Sebbene il focus principale della tesi sia il count-distinct, il framework è pensato per includere anche sketch con obiettivi diversi.

Def. 2.22 (Stima di frequenza puntuale)**.** Dato un elemento $a \in \mathcal{U}$, una query di frequenza restituisce una stima $\hat{f}(a)$ della frequenza reale $f(a)$. Nei metodi approssimati, la garanzia è tipicamente espressa come errore additivo controllato con probabilità alta.

Def. 2.23 (Membership approssimata)**.** Data una chiave x, una query di membership decide se x appartiene all'insieme osservato. Le strutture approssimate sono spesso progettate per non produrre falsi negativi, accettando una probabilità controllata di falsi positivi.

Nel Capitolo 3, oltre alla famiglia FM→HLL++, verranno quindi introdotti anche Count-Min Sketch per frequenze e Bloom filter per membership [18, 19, 20].

2.9 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 [21, 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* [22]).

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 definire l'attuale stato dell'arte 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 [21, 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 ridotto. 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 in cui la stream può crescere senza un limite prefissato, questa soluzione non è praticabile. Per questo motivo si ricorre a sketch randomizzati con garanzie probabilistiche (ε, δ), già formalizzate nel Capitolo 2.

Dal lato teorico, esistono algoritmi ottimali per il problema dei distinti che raggiungono spazio dell'ordine $O((\varepsilon^{-2} + \log n)\operatorname{polylog} n)$, 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.

In [4] viene mostrato come i migliori risultati generali riportati prima dell'algoritmo ottimale avevano spazio $O\big(\varepsilon^{-2}(\log(1/\varepsilon) + \log\log n) + \log n\big)$ e tempo di update $O\big(\varepsilon^{-2}(\log(1/\varepsilon) + \log\log n)\big)$, mentre il risultato ottimale raggiunge spazio $O\big(\varepsilon^{-2} + \log n\big)$ bit e update/query in $O(1)$.

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 grandezza 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.

Def. 3.1 (PCSA)**.** Sia $h : \mathcal{U} \to \{0,1\}^L$ una funzione di hash uniforme, con $L = w$ (lunghezza in bit del valore di hash). Indicando con $\rho(y)$ la posizione del primo bit a 1 in y, vale:

$$\rho(y) = \min\{r \geq 1 : y_r = 1\},$$

e, sotto l'ipotesi di uniformità, la variabile $\rho(y)$ ha coda geometrica:

$$\Pr[\rho(y) \geq t] = 2^{-(t-1)}.$$

Questa proprietà è la base della stima di cardinalità: osservare valori grandi di ρ diventa più probabile quando cresce il numero di distinti. Nel caso di PCSA, la partizione in m sottostream rende più stabile la stima rispetto alla versione PC con una sola statistica globale [14].

In pratica, PCSA non crea m stream fisiche separate: la partizione è *virtuale* ed è indotta dalla funzione di hash. Ogni elemento x viene assegnato a un indice $j \in \{0, \ldots, m-1\}$; gli elementi con lo stesso indice aggiornano la stessa bitmap $B[j]$. In questo modo, la stima finale combina informazione proveniente da più sottogruppi indipendenti (in media), riducendo la variabilità rispetto al caso con una sola bitmap.

Per l'esempio seguente assumiamo bitmap di lunghezza 4 e rappresentiamo ogni bitmap come $[b_1 b_2 b_3 b_4]$, dove $b_r = 1$ indica che la posizione r è stata osservata almeno una volta.

Nell'esempio della Tabella 3.1, gli elementi a e c finiscono nello stesso sottostream ($j = 1$), mentre b e d vengono indirizzati a sottostream diversi. Ogni riga aggiorna quindi una sola bitmap $B[j]$ e la stima finale media i contributi delle bitmap. In particolare, nello stato finale risultano attive tre bitmap su quattro, con stima finale $\hat{F}_0 \approx 14.62$ (in questo esempio volutamente piccolo).

Passo	x	$y = h(x)$	$j = y \bmod 4$	$t = \lfloor y/4 \rfloor$	$r = \rho(t)$	Update	Stato bitmap ($B_0 \mid B_1 \mid B_2 \mid B_3$)	$\hat{F}_0$ (PCSA)
1	a	25	1	6 (110_2)	2	$B[1,2] \leftarrow 1$	0000 | 0100 | 0000 | 0000	≈ 10.34
2	b	14	2	3 (11_2)	1	$B[2,1] \leftarrow 1$	0000 | 0100 | 1000 | 0000	≈ 12.29
3	c	9	1	2 (10_2)	2	$B[1,2] \leftarrow 1$	0000 | 0100 | 1000 | 0000	≈ 12.29
4	d	7	3	1 (1_2)	1	$B[3,1] \leftarrow 1$	0000 | 0100 | 1000 | 1000	≈ 14.62

Tabella 3.1: Esempio minimale di partizione PCSA in $m = 4$ sottostream virtuali.

Algoritmo 3.1 PCSA (adapted from Fig. 1 in [14])

Choose m bitmaps $B[0], \ldots, B[m-1]$ of length L, initialize all bits to 0
Choose a hash function $h : \mathcal{U} \to \{0,1\}^L$ with L large enough
for each element x in the stream
  $y \leftarrow h(x)$
  $j \leftarrow y \bmod m$
  $t \leftarrow \lfloor y/m \rfloor$
  $r \leftarrow \rho(t)$
  Set $B[j, r] \leftarrow 1$
end for
for $j = 0$ **to** $m - 1$
  $R_j \leftarrow \textsc{FirstZero}(B[j])$
end for
$\bar{R} \leftarrow \frac{1}{m} \sum_{j=0}^{m-1} R_j$
$\hat{F}_0 \leftarrow \frac{m}{\phi} \cdot 2^{\bar{R}}$ with $\phi \approx 0.77351$
return $\hat{F}_0$

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

Nella forma PCSA, ai fini di trovare il valore corretto per la costante ϕ esso è stato calcolato empiricamente e l'errore relativo standard atteso è circa $0.78/\sqrt{m}$. Gli autori riportano anche ordini di grandezza pratici: con $m = 64$ si ottiene tipicamente un errore intorno al 10%, mentre con $m = 256$ si scende intorno al 5%.

Complessità Con m bitmap di lunghezza L, lo spazio è

$$S_{\text{PCSA}}(m, L) = \Theta(mL) \text{ bit.}$$

L'update richiede tempo $\Theta(1)$ per elemento. La query richiede $\Theta(mL)$ nel caso diretto (scansione per trovare il primo zero di ogni bitmap) oppure $\Theta(m)$ se si mantiene informazione ausiliaria sul primo zero per ciascuna bitmap.

3.2.2 LogLog

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

Def. 3.2 (LogLog)**.** Sia $m = 2^p$ il numero di registri. Per ogni elemento x si calcola $y = h(x)$, si usa il prefisso di p bit per selezionare il registro j, e il suffisso per calcolare $\rho(w)$. L'update è quindi:

$$M[j] \leftarrow \max\{M[j], \rho(w)\}.$$

Indicando con

$$A = \frac{1}{m} \sum_{j=0}^{m-1} M[j],$$

lo stimatore LogLog ha forma

$$\hat{F}_0 = \alpha_m \cdot m \cdot 2^A,$$

dove α_m è una costante di calibrazione (asintoticamente circa 0.397). L'errore relativo standard tipico è dell'ordine $\approx 1.30/\sqrt{m}$ [15].

La relazione con F_0 può essere letta anche in modo esplicito: il numero atteso di elementi che cadono in ciascun registro è circa F_0/m, quindi i massimi $M[j]$ tendono a concentrarsi attorno a $\log_2(F_0/m)$; la media dei registri fornisce quindi una stima logaritmica della cardinalità complessiva.

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

Esempio. Consideriamo un caso minimale con $p = 2$ ($m = 4$ registri) e hash su 8 bit. I primi 2 bit selezionano il registro j, i restanti 6 bit formano il suffisso w su cui si calcola $\rho(w)$.

Algoritmo 3.2 LogLog (adapted from [15])

Choose precision p and set $m = 2^p$
Initialize registers $M[0], \ldots, M[m-1] \leftarrow 0$
for each element x in the stream
  $y \leftarrow h(x)$
  $j \leftarrow \textsc{Prefix}_p(y)$
  $w \leftarrow \textsc{Suffix}_{L-p}(y)$
  $M[j] \leftarrow \max\{M[j], \rho(w)\}$
end for
$A \leftarrow \frac{1}{m} \sum_{j=0}^{m-1} M[j]$
$\hat{F}_0 \leftarrow \alpha_m \cdot m \cdot 2^A$
return $\hat{F}_0$

Passo	x	$y = h(x)$	$j = \textsc{Prefix}_2(y)$	$w = \textsc{Suffix}_6(y)$	$\rho(w)$	Update	Stato registri (M_0, M_1, M_2, M_3)	$\hat{F}_0$ (LogLog)
1	a	10010100_2	2	010100_2	2	$M[2] \leftarrow \max(0,2) = 2$	$(0,0,2,0)$	≈ 2.25
2	b	10111000_2	2	111000_2	1	$M[2] \leftarrow \max(2,1) = 2$	$(0,0,2,0)$	≈ 2.25
3	c	01001100_2	1	001100_2	3	$M[1] \leftarrow \max(0,3) = 3$	$(0,3,2,0)$	≈ 3.78
4	d	11001000_2	3	001000_2	3	$M[3] \leftarrow \max(0,3) = 3$	$(0,3,2,3)$	≈ 6.35
5	e	10000100_2	2	000100_2	4	$M[2] \leftarrow \max(2,4) = 4$	$(0,3,4,3)$	≈ 8.99

Tabella 3.2: Esempio operativo di LogLog con aggiornamento dei registri.

Nella Tabella 3.2 si vede la logica centrale di LogLog: ogni elemento aggiorna un solo registro e l'operazione è sempre un massimo, quindi i registri crescono monotonicamente. La colonna $\hat{F}_0$ mostra la stima LogLog ottenuta dopo ogni passo, usando la formula del paragrafo con $\alpha_m \approx 0.397$.

Complessità Se ogni registro codifica valori in $[0, L-p+1]$, servono $\lceil \log_2(L-p+2) \rceil$ bit per registro. Quindi

$$S_{\text{LogLog}}(m, L, p) = \Theta\big(m \log(L-p+2)\big) \text{ bit.}$$

L'update è $\Theta(1)$ per elemento e la query è $\Theta(m)$.

Limiti pratici. LogLog riduce nettamente la varianza rispetto a FM, ma resta sensibile alla costante di normalizzazione e alla qualità dell'hash. In particolare, quando la cardinalità è molto piccola rispetto a m, la stima tende ad avere bias maggiore rispetto alle varianti successive.

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 $\text{RSE} \approx 1.04/\sqrt{m}$ [16].

Def. 3.3 (HyperLogLog)**.** Definendo

$$Z = \sum_{j=0}^{m-1} 2^{-M[j]},$$

la stima grezza di HLL è:

$$E = \alpha_m \frac{m^2}{Z},$$

con costanti pratiche

$$\alpha_{16} = 0.673, \qquad \alpha_{32} = 0.697, \qquad \alpha_{64} = 0.709,$$

e per $m \geq 128$:

$$\alpha_m \approx \frac{0.7213}{1 + 1.079/m}.$$

La scelta della media armonica non è solo formale: attenua il peso dei registri con valori estremi e produce una stima più stabile rispetto a LogLog. Dal punto di vista statistico, la varianza scende mantenendo la stessa struttura di update, con un vantaggio diretto nel rapporto memoria/accuratezza.

Nella variante pratica del paper, si applicano correzioni di range. Se V_0 è il numero di registri a zero:

$$\hat{F}_0 = \begin{cases} m \log(m/V_0), & \text{se } E \leq \frac{5}{2}m \text{ e } V_0 > 0 \\ E, & \text{nel regime centrale} \\ -2^{32} \log\left(1 - \frac{E}{2^{32}}\right), & \text{nel regime vicino a } 2^{32}. \end{cases}$$

L'articolo [16] fornisce sia l'analisi asintotica, sia la variante pratica con correzioni di range, che costituisce il riferimento per implementazioni sperimentali riproducibili.

Esempio. Se $m = 1024$ e dopo gli update molti registri restano nulli (V_0 grande), la stima $m \log(m/V_0)$ risulta più affidabile della stima grezza E. Quando la cardinalità cresce e i registri nulli diminuiscono, la stima entra nel regime centrale basato sulla media armonica.

Complessità Con $m = 2^p$ registri, lo spazio è

$$S_{\text{HLL}}(m, L, p) = \Theta\big(m \log(L - p + 2)\big) \text{ bit},$$

che in pratica viene spesso implementato con registri di ampiezza fissa (tipicamente 5–6 bit). Il tempo di update è $\Theta(1)$ per elemento e la query è $\Theta(m)$.

Le formule precedenti assumono parametrizzazione coerente dello sketch ($m = 2^p$), stessa definizione di ρ, e funzione di hash sufficientemente uniforme. In presenza di hash distorto, i registri non campionano più correttamente la stream e la stima può mostrare bias non trascurabile.

Nel paper originale, la procedura è esplicitamente separata in due parti: *raw estimator* e *range correction*. Questa separazione è importante perché il comportamento asintotico (regime centrale) e quello ai bordi (cardinalità molto piccole o vicine al dominio hash) sono governati da meccanismi diversi [16].

Algoritmo 3.3 HyperLogLog (adapted from Fig. 3 in [16])

Choose precision p, set $m = 2^p$, initialize $M[0..m-1] \leftarrow 0$
for each element x in the stream
  $y \leftarrow h(x)$
  $j \leftarrow \textsc{Prefix}_p(y)$
  $w \leftarrow \textsc{Suffix}_{L-p}(y)$
  $M[j] \leftarrow \max\{M[j], \rho(w)\}$
end for
$Z \leftarrow \sum_{j=0}^{m-1} 2^{-M[j]}$
$E \leftarrow \alpha_m m^2 / Z$ (raw estimate)
if $E \leq \frac{5}{2}m$
  $V_0 \leftarrow$ number of registers equal to 0
  if $V_0 \neq 0$
    $E^\star \leftarrow m \log(m/V_0)$
  else
    $E^\star \leftarrow E$
  end if
else if $E \leq \frac{1}{30} \cdot 2^{32}$
  $E^\star \leftarrow E$
else
  $E^\star \leftarrow -2^{32} \log(1 - E/2^{32})$
end if
return $\hat{F}_0 \leftarrow E^\star$

3.2.4 HyperLogLog++

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

- uso di una 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.

Def. 3.4 (HyperLogLog++)**.** La struttura mantiene la stessa regola di update di HLL sui registri, ma introduce due passaggi aggiuntivi: rappresentazione sparsa nelle cardinalità piccole e correzione empirica del bias. Indicando con E la stima HLL grezza, la stima corretta è:

$$E_{\text{corr}} = E - \text{bias}(E, p),$$

dove il termine di bias è tabulato per ciascun livello di precisione p (interpolazione tra punti noti). La stima finale sceglie il ramo con errore atteso minore tra linear counting ed E_{corr} [17].

Nel formato *sparse*, lo sketch memorizza solo i registri non nulli (coppie indice-valore), riducendo spazio e costo costante nelle cardinalità piccole. Il passaggio a formato *dense* avviene quando la rappresentazione sparsa non è più conveniente in memoria.

La formulazione in [17] usa due precisioni: p per lo sketch *dense* e p' (con $p \leq p' \leq 64$) per la codifica *sparse*; in questo modo lo stesso algoritmo può avere uno spazio molto più piccolo quando la cardinalità è bassa, senza perdere compatibilità con la stima HLL nel regime normale [17].

Le routine ausiliarie usate nel pseudocodice hanno significato preciso: $\textsc{LinearCounting}(m, V_0) = m \log(m/V_0)$ per $V_0 > 0$, $\textsc{EstimateBias}(E, p)$ applica l'interpolazione sulla tabella empirica del bias (nel paper con schema di nearest neighbours), e $\textsc{Threshold}(p)$ è la soglia empirica di switch tra i due rami di stima riportata in Figura 6 [17]. Nel ramo *sparse* la stessa formula di linear counting viene applicata con $m' = 2^{p'}$ e con il numero di registri non nulli dedotto dalla sparse list.

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].

Il paper specifica che $\textsc{Threshold}(p)$ è determinata empiricamente (tabulata per ogni precisione) e che le routine ausiliarie Merge/ToNormal sono parte essenziale della transizione sparse to dense. Nella discussione implementativa, gli autori descrivono anche una strategia pratica in cui il *temporary set* viene fuso periodicamente (ad esempio intorno al 25% della capacità massima della rappresentazione sparsa) per mantenere efficiente la gestione degli inserimenti [17].

Esempio. Con $p = 14$ ($m = 16384$), per cardinalità piccole lo sketch può restare in formato sparse, riducendo la memoria effettiva. Superata una soglia, la struttura passa in dense e usa l'estimatore HLL corretto dal bias.

Algoritmo 3.4 HyperLogLog++ (adapted from Fig. 6 in [17])

Input: stream S, precisions p and p' with $p \leq p' \leq 64$
$m \leftarrow 2^p, \quad m' \leftarrow 2^{p'}$
$\alpha_{16} \leftarrow 0.673, \alpha_{32} \leftarrow 0.697, \alpha_{64} \leftarrow 0.709$
$\alpha_m \leftarrow 0.7213/(1 + 1.079/m)$ for $m \geq 128$
Initialize sparse mode, temporary set, sparse list, and dense registers
for each element x in S
  $y \leftarrow h_{64}(x)$
  Encode sparse value from y using (p, p')
  if mode is sparse
    Insert encoded value in temporary set
    if temporary set is full ($|\text{temporary\_set}| \geq 0.25 \cdot 2^{p'}$)
      Merge temporary set into sparse list
    end if
    if $|\text{sparse\_list}| > 6m$ bits
      Convert sparse representation to dense registers
      mode $\leftarrow$ normal
    end if
  else
    Extract (j, ρ) from y
    $M[j] \leftarrow \max\{M[j], \rho\}$
  end if
end for
if mode is sparse
  Merge temporary set into sparse list
  return LinearCounting$(m', m' - |\text{sparse_list}|)$
else
  $E \leftarrow \alpha_m m^2 \left(\sum_{j=0}^{m-1} 2^{-M[j]}\right)^{-1}$
  if $E \leq 5m$
    $E' \leftarrow E -$ EstimateBias(E, p)
  else
    $E' \leftarrow E$
  end if
  $V_0 \leftarrow$ number of registers equal to 0
  if $V_0 \neq 0$
    $H \leftarrow$ LinearCounting(m, V_0)
  else
    $H \leftarrow E'$
  end if
  if $H \leq$ Threshold(p)
    return H
  else
    return E'
  end if
end if

Complessità In modalità *sparse*, lo spazio è $\Theta(|\text{sparse\_list}|)$ (in bit codificati) fino alla soglia di conversione; in modalità *dense*, lo spazio è $\Theta(m)$ registri (spesso circa $6m$ bit in implementazioni pratiche). L'update è $\Theta(1)$ ammortizzato, con costo $\Theta(m)$ durante la conversione sparse to dense. La query è $\Theta(|\text{sparse\_list}|)$ in sparse e $\Theta(m)$ in dense.

Limiti e compromessi. HLL++ riduce bias pratico, soprattutto nei range piccoli e intermedi, ma introduce maggiore complessità ingegneristica: soglie di switching, tabelle di correzione e doppia rappresentazione dello stato. Per questo motivo è importante distinguere sempre tra specifica teorica dell'estimatore e dettagli implementativi della variante "in practice".

3.3 Confronto sintetico tra gli approcci

La Tabella 3.3 riassume le differenze principali tra gli algoritmi della linea storica.

Dal punto di vista formale, la differenza principale non è l'ordine asintotico ($\Theta(1/\sqrt{m})$ domina asintoticamente), ma la costante moltiplicativa dell'errore, il comportamento ai bordi di range e la robustezza in scenari reali dove hash, cardinalità e distribuzioni non sono ideali.

Algoritmo	Stato dello sketch	Regola di stima	Space	Ordine errore
PCSA	Bitmap (una o più)	Primo zero / media su bitmap	$O(\varepsilon^{-2} \log n)$	$\Theta(1/\sqrt{m})$
LogLog	Array registri (max di ρ)	Media geometrica normalizzata	$O(\varepsilon^{-2} \log\log n + \log n)$	$\Theta(1/\sqrt{m})$
HLL	Array registri (max di ρ)	Media armonica + range corrections	$O(\varepsilon^{-2} \log\log n + \log n)$	$\approx 1.04/\sqrt{m}$
HLL++	Registri + formato sparse	HLL + bias correction + adaptive switching	$O(\varepsilon^{-2} \log\log n + \log n)$	$\Theta(1/\sqrt{m})$ con minore bias pratico

Tabella 3.3: Confronto ad alto livello tra i principali algoritmi count-distinct.

La colonna "Space" è uniformata in notazione teorica rispetto a n ed ε. In questa forma, LogLog/HLL/HLL++ hanno lo stesso ordine $O(\varepsilon^{-2} \log\log n + \log n)$; HLL++ migliora soprattutto costanti e regimi pratici (bias e small-range), non l'ordine asintotico dominante. Nella colonna "Ordine errore" si riporta invece l'andamento della RSE rispetto al numero di registri m (non alla memoria totale in bit): l'ordine asintotico resta $\Theta(1/\sqrt{m})$, ma con costanti diverse (ad esempio 1.04 per HLL) e con correzioni pratiche aggiuntive nel caso HLL++.

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.

La tabella precedente evidenzia anche che, a parità di ordine asintotico dominante, le differenze pratiche dipendono molto dalla gestione dei regimi di cardinalità e della correzione del bias.

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++, mantenendo la notazione del Capitolo 2, la regola di merge naturale è:

$$\big(\mathcal{K}(S_1) \oplus \mathcal{K}(S_2)\big)[j] = \max\{\mathcal{K}(S_1)[j], \mathcal{K}(S_2)[j]\}.$$

Per ogni registro j, lo stato dello sketch di una stream S contiene:

$$\mathcal{K}(S)[j] = \max_{x \in S:\, j(x)=j} \rho(w(x)).$$

Qui $j(x)$ è la funzione che mappa l'elemento x nel registro selezionato dai primi $\log_2 m$ bit dell'hash, mentre $w(x)$ è il suffisso su cui si valuta ρ, in linea con le sezioni precedenti su LogLog e HLL. Se i due sketch sono costruiti con stessa parametrizzazione e stessa funzione di hash/seed, allora per ogni j:

$$\mathcal{K}(S_1 \cdot S_2)[j] = \max(\mathcal{K}(S_1)[j], \mathcal{K}(S_2)[j])\,.$$

Quindi il merge registro per registro coincide con lo stato che si otterrebbe processando la concatenazione delle due stream:

$$\mathcal{K}(S_1) \oplus \mathcal{K}(S_2) = \mathcal{K}(S_1 \cdot S_2).$$

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 PC/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].

3.6 Estensioni oltre il count-distinct

Oltre alla stima di F_0, il framework può trattare sketch con obiettivi diversi: stima di frequenze e query di membership. Riprendendo la notazione (ε, δ) e i simboli di memoria già introdotti nei capitoli precedenti, queste strutture mantengono la stessa impostazione metodologica: stato compatto, garanzie probabilistiche e trade-off esplicito tra precisione e spazio. Nel caso frequenze, ε controlla un errore additivo $\varepsilon\|f\|_1$ con probabilità $1-\delta$; nel caso membership (Bloom filter), il trade-off è espresso soprattutto dalla probabilità di falso positivo p_{fp}, determinata da m_{bf} e k_{bf}.

3.6.1 Count-Min Sketch

Def. 3.5 (Count-Min Sketch)**.** Il Count-Min Sketch mantiene una matrice di contatori $C \in \mathbb{N}^{d \times w_{\text{cm}}}$ e d funzioni di hash $h_1, \ldots, h_d : \mathcal{U} \to [w_{\text{cm}}]$. Per ogni update dell'elemento x, si incrementa $C[j, h_j(x)]$ per ogni riga j; la stima puntuale è

$$\hat{f}(x) = \min_{j \in [d]} C[j, h_j(x)].$$

Per $w_{\text{cm}} = \lceil e/\varepsilon \rceil$ e $d = \lceil \ln(1/\delta) \rceil$, vale con probabilità almeno $1-\delta$:

$$f(x) \le \hat{f}(x) \le f(x) + \varepsilon\|f\|_1.$$

dove $\|f\|_1$ è la somma delle frequenze (lunghezza della stream). [18]

Esempio. Consideriamo un CMS con $d = 2$ righe e $w_{\text{cm}} = 5$ colonne (indici $0, \ldots, 4$), inizialmente tutto a zero. Supponiamo: $h_1(a) = 1,\ h_2(a) = 3, h_1(b) = 4,\ h_2(b) = 1, h_1(c) = 1,\ h_2(c) = 4$. Processiamo la stream a, b, a, c, a.

Passo	x	$(h_1(x), h_2(x))$	Update	Stato riga 1	Stato riga 2
1	a	$(1, 3)$	$C[1,1]++,\ C[2,3]++$	$[0,1,0,0,0]$	$[0,0,0,1,0]$
2	b	$(4, 1)$	$C[1,4]++,\ C[2,1]++$	$[0,1,0,0,1]$	$[0,1,0,1,0]$
3	a	$(1, 3)$	$C[1,1]++,\ C[2,3]++$	$[0,2,0,0,1]$	$[0,1,0,2,0]$
4	c	$(1, 4)$	$C[1,1]++,\ C[2,4]++$	$[0,3,0,0,1]$	$[0,1,0,2,1]$
5	a	$(1, 3)$	$C[1,1]++,\ C[2,3]++$	$[0,4,0,0,1]$	$[0,1,0,3,1]$

Tabella 3.4: Esempio operativo di Count-Min Sketch.

Dallo stato finale della Tabella 3.4, per a si ottiene $\hat{f}(a) = \min\{C[1,1], C[2,3]\} = \min\{4,3\} = 3$ (valore esatto), mentre la struttura resta in generale sovrastimante in presenza di collisioni.

Complessità Lo spazio è $\Theta(d\, w_{\text{cm}})$ contatori, quindi $\Theta(d\, w_{\text{cm}} \log \|f\|_1)$ bit con contatori interi standard. Update e query puntuale costano $\Theta(d) = \Theta(\log(1/\delta))$. La mergeabilità è per somma componente-per-componente di matrici compatibili.

3.6.2 Bloom Filter (membership)

Def. 3.6 (Bloom Filter)**.** Un Bloom filter è un array di m_{bf} bit con k_{bf} hash. L'inserimento di x imposta a 1 le celle $B[h_1(x)], \ldots, B[h_{k_{\text{bf}}}(x)]$; la query di membership risponde "presente" se tutte queste celle valgono 1. La struttura non produce falsi negativi, ma può produrre falsi positivi con probabilità approssimativa

$$p_{\text{fp}} \approx \left(1 - e^{-k_{\text{bf}} n_{\text{ins}} / m_{\text{bf}}}\right)^{k_{\text{bf}}},$$

con n_{ins} numero di elementi inseriti nel filtro. [19, 20]

Esempio. Consideriamo un Bloom filter con $m_{\text{bf}} = 10$ bit (indici $0, \ldots, 9$) e $k_{\text{bf}} = 2$ hash. Supponiamo: $h_1(a) = 1,\ h_2(a) = 6, h_1(b) = 4,\ h_2(b) = 6, h_1(c) = 1,\ h_2(c) = 8$.

Passo	Operazione	(h_1, h_2)	Celle toccate	Stato bitset $[b_0 \ldots b_9]$	Esito
1	insert(a)	$(1, 6)$	$b_1, b_6 \leftarrow 1$	[0100001000]	–
2	insert(b)	$(4, 6)$	$b_4, b_6 \leftarrow 1$	[0100101000]	–
3	insert(c)	$(1, 8)$	$b_1, b_8 \leftarrow 1$	[0100101010]	–
4	query(z)	$(1, 4)$	verifica b_1, b_4	[0100101010]	"presente" (falso positivo)
5	query(q)	$(2, 9)$	verifica b_2, b_9	[0100101010]	"assente"

Tabella 3.5: Esempio operativo di Bloom filter con un falso positivo.

La Tabella 3.5 mostra il comportamento tipico: nessun falso negativo sugli elementi inseriti, ma possibile falso positivo su query di elementi non inseriti.

Complessità Lo spazio è esattamente m_{bf} bit. Update e query costano $\Theta(k_{\text{bf}})$. La mergeabilità, a parità di parametri e hash, è la OR bit-a-bit.

4 Implementazione

In questo capitolo si descrive l'implementazione del sistema sperimentale sviluppato per valutare algoritmi di stima del numero di distinti su flussi di dati. L'obiettivo progettuale è separare in modo netto:

- la logica algoritmica (sketch e stimatori),
- la gestione dei dataset,
- il protocollo di valutazione e l'esportazione delle metriche.

L'implementazione è realizzata principalmente in `C++` (algoritmi, framework, CLI), con supporto `Python` per generazione dataset, orchestrazione esperimenti e analisi.

4.1 Architettura del sistema

La struttura del progetto è organizzata nei seguenti moduli principali:

- `src/satp/algorithms`: implementazioni degli algoritmi;
- `src/satp/io`: caricamento dataset binari compressi;
- `src/satp/simulation`: framework di valutazione e ciclo di esecuzione;
- `main.cpp`: CLI interattiva per benchmark in modalità *normale* e *streaming*;
- `scripts`: generazione dataset, orchestrazione campagne sperimentali, analisi.

Il processo completo di un esperimento è:

1. generazione del dataset partizionato con script Python;
2. indicizzazione dei metadati e caricamento di una partizione alla volta;

3. esecuzione dell'algoritmo sulle partizioni (esecuzioni indipendenti);
4. aggregazione metriche statistiche nel framework;
5. esportazione CSV in cartelle standardizzate sotto `results/`.

4.2 Dataset binario compresso

4.2.1 Motivazione del formato

Per gestire flussi di grandi dimensioni e numerose esecuzioni, è stato adottato un formato binario compresso per partizione in luogo dei formati testuali. Questa scelta riduce l'occupazione su disco e consente una lettura sequenziale efficiente.

Il formato è implementato in `scripts/generate_partitioned_dataset_bin.py` (generazione) e `src/satp/io/BinaryDatasetIO.h` (parsing e I/O).

4.2.2 Struttura del file

Il file inizia con un'intestazione globale (magic, versione, parametri), seguito da tabella partizioni e payload compressi.

Campo header	Significato
`MAGIC=SATPDBN2`	Identificatore formato
`VERSION=2`	Versione schema
`n`	Elementi per partizione
`d`	Distinti per partizione
`p`	Numero di partizioni (esecuzioni)
`seed`	Seed globale del dataset

Tabella 4.1: Campi principali dell'header del dataset binario.

Ogni voce della tabella delle partizioni contiene:

- offset e dimensione in byte dei valori compressi (`uint64` little-endian);
- offset e dimensione in byte dei bit di verità compressi (`uint64` little-endian);
- metadati locali `n`, `d` (`uint64`);
- codifiche payload: `ENCODING_ZLIB_U32_LE` per i valori e `ENCODING_ZLIB_BITSET_LE` per il bitset.

4.2.3 Bitset di verità per $F_0(t)$

Per ogni partizione, oltre ai valori del flusso, viene salvata una sequenza binaria b_t dove:

$$b_t = \begin{cases} 1 & \text{se l'elemento in posizione } t \text{ è una nuova chiave,} \\ 0 & \text{altrimenti.} \end{cases}$$

Il numero esatto di distinti sul prefisso è quindi:

$$F_0(t) = \sum_{i=1}^{t} b_i.$$

Questa informazione viene usata dal framework in modalità *streaming* per confrontare in modo esatto $\hat{F}_0(t)$ e $F_0(t)$ senza ricostruire insiemi globali durante la valutazione.

4.2.4 Caricamento "una partizione alla volta"

La classe `BinaryDatasetPartitionReader` in `src/satp/io/BinaryDatasetIO.h` mantiene aperto il descrittore del file e carica solo la partizione richiesta, decomprimendo esclusivamente il blocco necessario. In questo modo la memoria RAM rimane proporzionale alla singola partizione corrente, non all'intero dataset.

4.3 Generazione dataset

Lo script `scripts/generate_partitioned_dataset_bin.py` genera file con nome:

`dataset_n_{n}_d_{d}_p_{p}_s_{seed}.bin`.

Per ogni partizione:

1. si fissano d chiavi distinte;
2. si forzano posizioni casuali per garantire che tutte appaiano almeno una volta;
3. si riempiono le altre posizioni campionando uniformemente tra le d chiavi;
4. si aggiorna il bitset b_t delle prime occorrenze;
5. si comprime con zlib.

La generazione supporta parallelismo multiprocesso (parametro `-workers`) e barra di avanzamento su `stderr`. La correttezza strutturale è verificata nei test Python in `scripts/tests/test_generate_dataset.py`.

4.4 Interfacce comuni e hashing

4.4.1 Interfaccia base degli algoritmi

Il contratto comune è definito in `src/satp/algorithms/Algorithm.h`:

- `process(uint32_t id)`: aggiorna lo sketch;
- `count()`: restituisce la stima corrente di F_0;
- `reset()`: azzeramento stato;
- `getName()`: nome algoritmo.

Questa interfaccia permette al framework di istanziare in modo uniforme algoritmi diversi, mantenendo invariato il protocollo sperimentale.

4.4.2 Hashing uniforme nel codice

La randomizzazione è centralizzata in `src/satp/hashing.h`:

- `splitmix64(uint64_t)` come hash a 64 bit;
- `hash32_from_64(uint64_t)` per derivare una versione 32 bit.

La scelta implementativa usa `splitmix64` come sorgente unica, con troncamento quando gli algoritmi richiedono dominio a 32 bit (LogLog/HLL in modalità paper-strict).

4.5 Implementazione degli algoritmi

4.5.1 NaiveCounting

`NaiveCounting(src/satp/algorithms/NaiveCounting.cpp)` mantiene un `std::set<uint32_t>` delle chiavi osservate.

- aggiornamento: inserimento nel set;
- interrogazione: cardinalità del set.

È la baseline esatta, con spazio $\Theta(F_0)$ e tempo di aggiornamento $O(\log F_0)$.

4.5.2 ProbabilisticCounting

`ProbabilisticCounting` (`ProbabilisticCounting.cpp`) usa bitmap a 32 bit e parametro $L \in [1, 31]$.

- aggiornamento: calcolo del bit della prima posizione utile e OR sul bitmap;
- interrogazione: ricerca del primo zero e stima $\propto 2^R$ con costante ϕ.

È una variante compatta del paradigma FM/PC, con aggiornamento costante e spazio costante nel codice attuale.

4.5.3 LogLog

`LogLog` (`LogLog.cpp`) è implementato in modalità *paper-strict*:

- $k \in [4, 16]$, $m = 2^k$;
- dominio hash 32 bit ($L = 32$).

Ogni aggiornamento:

1. seleziona registro dal prefisso hash;
2. calcola ρ sul suffisso;
3. applica l'aggiornamento di massimo sul registro.

L'implementazione mantiene $\sum_j M[j]$ incrementale (variabile `sumRegisters`), quindi `count()` è $O(1)$ senza scansione completa dei registri.

4.5.4 HyperLogLog

`HyperLogLog` (`HyperLogLog.cpp`) segue anch'esso modalità *paper-strict* ($k \in [4, 16]$, $L = 32$).

Lo stato mantiene:

- registri $M[j]$,
- $\sum_j 2^{-M[j]}$ incrementale (`sumInversePowers`),
- numero di registri nulli V_0 (`zeroRegisters`).

Questo consente interrogazioni in $O(1)$ e applica le tre correzioni standard:

- regime di piccola cardinalità (linear counting);
- regime intermedio (stima grezza);
- regime di grande cardinalità (correzione logaritmica su 2^{32}).

4.5.5 HyperLogLog++

`HyperLogLogPlusPlus` (`HyperLogLogPlusPlus.cpp`) implementa la variante con:

- parametro di precisione $p \in [4, 18]$, $m = 2^p$ (nei risultati del Capitolo 5 lo stesso parametro è indicato con k);
- hash a 64 bit;
- doppia rappresentazione *sparse*/*normale*;
- soglia di passaggio sparse→normal basata su costo in bit;
- correzione del bias tabulata (`hllpp_tables`);
- soglie `threshold_for_k(p)` per selezione tra linear counting e stima corretta.

La rappresentazione sparsa usa una lista codificata e ordinata per indice; la fusione dei contributi avviene mantenendo il ρ massimo per indice. In modalità normale, l'aggiornamento è analogo a HLL classico con registri e massimo. Dal punto di vista computazionale, in modalità *normale* aggiornamento e interrogazione hanno costo costante; in modalità *sparse*, invece, l'aggiornamento include flush/merge periodici (costo ammortizzato costante nel regime operativo considerato). Lo spazio è adattivo: compatto in *sparse* e proporzionale a $m = 2^p$ in *normale*.

4.6 Framework di valutazione

4.6.1 Doppia modalità: normale e streaming

La classe `EvaluationFramework` (`src/satp/simulation/EvaluationFramework.h`) espone:

- `evaluate(...)` per analisi all'endpoint (una stima per esecuzione);
- `evaluateStreaming(...)` per analisi prefisso $t \mapsto \hat{F}_0(t)$.

In modalità dataset binario, `runs`, `sample_size` e `seed` vengono letti dai metadati del file.

4.6.2 Metriche implementate

Per entrambe le modalità, il framework calcola le metriche definite nel Capitolo 2: media, varianza campionaria, deviazione standard, bias, errore relativo medio, RMSE, MAE, RSE osservata, oltre a $\bar{F}_0$ e $\bar{\hat{F}}_0$.

In streaming, le metriche sono aggregate sulle esecuzioni per checkpoint comuni. I checkpoint sono fissati a:

$$K_{\text{chk}} = \min\{n, 200\},$$

con posizione

$$t_i = \left\lceil \frac{i\,n}{K_{\text{chk}}} \right\rceil, \quad i = 1, \ldots, K_{\text{chk}}.$$

4.6.3 Output CSV

Le funzioni `evaluateToCsv` e `evaluateStreamingToCsv` scrivono le colonne:

```
algorithm, params, mode, runs, sample_size, element_index,
distinct_count, seed, f0_mean, f0_hat_mean, mean, variance,
stddev, rse_theoretical, rse_observed, bias, difference,
bias_relative, mean_relative_error, rmse, mae
```

In modalità streaming, la colonna `distinct_count` riporta il valore globale d del dataset, mentre la verità al checkpoint t è `f0_mean`. Nel formato corrente del framework, `mean` e `f0_hat_mean` coincidono (ridondanza mantenuta per compatibilità con analisi pregresse).

4.7 CLI e orchestrazione sperimentale

4.7.1 CLI interattiva

Il file `main.cpp` implementa una CLI con comandi: `set`, `show`, `run`, `runstream`, `list`, `help`, `quit`.

I risultati sono salvati automaticamente in `results/<Algorithm>/<params>/results_oneshot.csv` (comando `run`) oppure `results/<Algorithm>/<params>/results_streaming.csv` (comando `runstream`), senza richiedere l'inserimento manuale del percorso.

4.7.2 Orchestrazione batch

Lo script `scripts/orchestrate_benchmarks.py` automatizza:

- generazione matrice dataset $(n, d, seed)$;
- invocazione CLI in modalità *oneshot* e/o *streaming*;
- popolamento strutturato della cartella `results/`.

4.8 Validazione e testing

La validazione include test C++ e Python.

4.8.1 Test algoritmici C++

In `tests/algorithms` sono presenti test per:

- accuratezza con intervalli basati su RSE teorica per HLL, HLL++ e LogLog;
- validazione parametri per HLL, HLL++ e LogLog;
- baseline esatta `NaiveCounting`;
- controllo di consistenza per `ProbabilisticCounting` con bound conservativi.

4.8.2 Test framework

`tests/simulation/EvaluationFrameworkTest.cpp` verifica che:

- le statistiche siano finite e che le metriche di errore/dispersione (`variance`, `stddev`, `rmse`, `mae`, `mean_relative_error`, `difference`) siano non negative;
- in modalità streaming con `NaiveCounting` valga $\hat{F}_0(t) = F_0(t)$ a ogni checkpoint;
- l'ultimo checkpoint coincida con il numero di distinti noto del dataset.

4.8.3 Test generazione dataset

`scripts/tests/test_generate_dataset.py` valida:

- conformità binaria dello schema;
- correttezza di $n, d, p, seed$;
- coerenza del bitset di verità prefisso;
- è presente un controllo di determinismo con seed fisso (anche con worker diversi), ma nello stato attuale non costituisce ancora una validazione forte della ripetibilità su output indipendenti.

4.9 Scelte progettuali e limiti attuali

Le scelte implementative principali sono:

- separazione netta tra algoritmi e framework di misura;
- dataset compresso con caricamento per partizione;
- metriche allineate alla formalizzazione statistica del Capitolo 2;

- pipeline riproducibile tramite seed e naming deterministico dei file.

I limiti correnti sono:

- assenza di fusione/serializzazione nel contratto `Algorithm`;
- assenza di implementazioni Count-Min Sketch e Bloom Filter nel codice C++;
- campagne complete multi-seed più onerose per HLL/HLL++ ai parametri più grandi.

Nel capitolo successivo si analizzano i risultati sperimentali ottenuti con il framework descritto.

5
Risultati sperimentali

Questo capitolo presenta i risultati empirici ottenuti con il framework implementato nel Capitolo 4. L'analisi è centrata sulla modalità *streaming*, cioè su metriche calcolate sui prefissi del flusso e aggregate per partizione (esecuzione indipendente).

5.1 Obiettivi sperimentali

Le domande sperimentali principali sono:

1. come si comportano gli algoritmi implementati in termini di errore relativo, bias e variabilità osservata;
2. dove e quanto `HyperLogLog++` migliora `HyperLogLog`;
3. come evolve la stima lungo il flusso (non solo all'endpoint).

5.2 Protocollo sperimentale

5.2.1 Dataset e configurazione

I dataset sono generati con `scripts/generate_partitioned_dataset_bin.py` secondo il disegno sperimentale pianificato:

- $n \in \{10^2, 10^3, 10^4, 10^5, 10^6, 10^7\}$;
- $d/n \in \{0.01, 0.1, 0.5, 1.0\}$;

- seed $\in \{21041998, 42, 137357, 10032018, 29042026\}$;
- numero partizioni $p = 50$.

La modalità streaming usa $\min\{n, 200\}$ checkpoint per partizione (campionamento uniforme in indice, come definito nel Capitolo 4).

La copertura effettivamente disponibile nei risultati analizzati non è uniforme:

- HLL/HLL++: 56 scenari endpoint, 1 seed, 4 valori di n;
- LogLog/Probabilistic Counting: 120 scenari endpoint, 5 seed, 6 valori di n.

Per questo motivo, i confronti globali tra famiglie algoritmiche vanno letti in chiave descrittiva; il confronto metodologicamente più solido è quello su scenari comuni (HLL vs HLL++).

5.2.2 Parametri algoritmici usati nei risultati

Nei risultati disponibili al momento della stesura:

- `HyperLogLog`: $k \in \{10, 12, 14, 16\}$, $L = 32$;
- `HyperLogLog++`: $k \in \{10, 12, 14, 16\}$ (nel codice il parametro equivalente è indicato con p);
- `LogLog`: $k = 16$, $L = 32$;
- `Probabilistic Counting`: $L = 16$.

Le metriche analizzate sono quelle riportate nel CSV del framework: MRE, MAE, RMSE, RB (bias relativo), RSE_{obs}, bias e varianza. Unità statistica delle sintesi endpoint: ogni riga a $t = n$ nel CSV rappresenta uno scenario (algoritmo, parametri, n, d, seed) già aggregato sulle $p = 50$ partizioni (esecuzioni indipendenti). Nelle tabelle di sintesi, media e mediana sono calcolate come aggregazioni non pesate tra scenari. Salvo diversa indicazione, i valori tabellari sono arrotondati a quattro cifre decimali per leggibilità.

5.3 Confronto globale all'endpoint

Per confrontare scenari eterogenei, si considera anzitutto il valore all'endpoint $t = n$ per ciascuna esecuzione, poi aggregato. La Tabella 5.1 riassume le statistiche su tutti gli scenari disponibili per ciascun algoritmo, ma non rappresenta una graduatoria rigorosamente omogenea tra algoritmi (coperture diverse per seed e scala n, oltre a una griglia parametri non uniforme: HLL/HLL++ aggregano $k \in \{10, 12, 14, 16\}$, mentre LogLog e Probabilistic Counting sono riportati in una sola configurazione). I dati sorgente delle tabelle endpoint provengono dai file `results/*/results_streaming.csv`, mentre le curve dinamiche sono analizzate nei notebook citati in seguito.

Dal confronto emergono tre aspetti:

- nel confronto su scenari comuni con HLL, **HLL++** mostra valori medi più bassi di errore relativo e bias relativo assoluto; il confronto con LogLog e Probabilistic Counting resta descrittivo, dato il diverso perimetro sperimentale;

Algoritmo	Scenari	Seed	$\lvert\mathcal{N}\rvert$	MRE media	MRE mediana	$\lvert$RB$\rvert$ media	$\mathrm{RSE}_{\mathrm{obs}}$ media
HyperLogLog	56	1	4	0.0136	0.0134	0.0074	0.0112
HyperLogLog++	56	1	4	0.0061	0.0002	0.0034	0.0047
LogLog	120	5	6	1360.80	3.4766	1360.80	0.0452
Probabilistic Counting	120	5	6	0.6420	0.6350	0.4625	0.5880

Tabella 5.1: Sintesi endpoint su tutti i CSV streaming disponibili. $\lvert\mathcal{N}\rvert$ indica il numero di valori distinti di n presenti nei risultati per algoritmo.

- **LogLog** mostra errore relativo molto alto quando la cardinalità vera è piccola rispetto a m, perché la stima senza correzioni per il regime di piccola cardinalità rimane strutturalmente sovrastimata;
- **Probabilistic Counting** con $L = 16$ ha variabilità elevata sul intervallo considerato, coerente con saturazione/instabilità per cardinalità grandi.

5.4 Analisi per algoritmo e scala del problema

Per evitare che medie globali siano dominate da casi estremi, si osservano anche le metriche endpoint al variare di n.

5.4.1 LogLog

n	Scenari	MRE mediana	MRE media	$\lvert$RB$\rvert$ media
10^2	20	1560.6268	7349.5813	7349.5813
10^3	20	155.6624	734.5651	734.5651
10^4	20	15.1730	73.0659	73.0659
10^5	20	1.1937	6.9997	6.9997
10^6	20	0.0198	0.5568	0.5567
10^7	20	0.0036	0.0111	0.0098

Tabella 5.2: Endpoint LogLog per scala n, con $k = 16$.

La Tabella 5.2 evidenzia un comportamento tipico: nel regime molto piccolo, LogLog sovrastima fortemente; quando n cresce, la stima rientra progressivamente nel regime centrale e l'errore si riduce.

5.4.2 Probabilistic Counting

Con bitmap a 16 bit, la qualità di Probabilistic Counting rimane limitata su cardinalità elevate: l'errore relativo resta alto e poco stabile al crescere di n.

n	Scenari	MRE mediana	MRE media	|RB| media
10^2	20	0.4042	0.6072	0.4920
10^3	20	0.5966	0.6256	0.3355
10^4	20	0.7201	0.6106	0.3365
10^5	20	0.5362	0.5679	0.2561
10^6	20	0.7580	0.6531	0.5671
10^7	20	0.9492	0.7876	0.7876

Tabella 5.3: Endpoint Probabilistic Counting per scala n, con $L = 16$.

5.5 Confronto HLL vs HLL++

Il confronto diretto tra HLL e HLL++ usa scenari comuni (stessi n, d, seed e k). Definiamo il guadagno:

$$\text{gain}_{\text{MRE}} = \text{MRE}_{\text{HLL}} - \text{MRE}_{\text{HLL++}}.$$

Valori positivi indicano vantaggio di HLL++.

k	Scenari	miglioramento MRE medio	miglioramento MRE mediano	% casi HLL++ migliore	miglioramento m
10	16	0.0131	0.0177	68.75%	
12	16	0.0083	0.0087	68.75%	
14	12	0.0050	0.0045	75.00%	
16	12	0.0016	0.0020	75.00%	

Tabella 5.4: Guadagno di HLL++ rispetto a HLL su scenari endpoint comuni.

In media HLL++ migliora HLL in tutti i k testati. Il trend di guadagno maggiore per k basso va però letto con cautela, perché per $k = 14, 16$ la copertura scenari è più ridotta (assenza dei casi con $n = 10^7$) e i valori riportati sono descrittivi, non accompagnati da inferenza statistica.

5.5.1 Dove la differenza è massima

La differenza è più netta nei regimi di piccola e media cardinalità, dove i meccanismi di correzione di HLL++ incidono maggiormente.

5.5.2 Caso grande $n = 10^7$

Per $n = 10^7$, seed 21041998 e sole configurazioni disponibili ($k = 10, 12$), HLL e HLL++ risultano quasi sovrapposti: i guadagni all'endpoint sono dell'ordine 10^{-6} –10^{-7} in più casi. Si tratta quindi di un risultato locale al blocco osservato, non di una generalizzazione multi-seed.

k	n	d	MRE HLL	MRE HLL++	guadagno MRE
10	10^5	10^5	0.03224	0.00005	0.03219
12	10^6	10^4	0.02853	0.00011	0.02842
12	10^5	10^4	0.02643	0.00011	0.02632
10	10^4	10^3	0.02604	0.00000	0.02604
10	10^5	10^3	0.02522	0.00000	0.02522

Tabella 5.5: Scenari endpoint con maggiore vantaggio di HLL++ su HLL.

5.6 Dinamica lungo il flusso

L'analisi streaming mostra come l'errore evolve con l'indice t e non solo all'endpoint. In particolare:

- per HLL/HLL++, i grafici su $n = 10^7$ mostrano curve molto vicine nel tratto ad alta cardinalità;
- nei casi con d più piccolo rispetto a n, le differenze appaiono prima e sono più visibili nei checkpoint iniziali/intermedi.

Per rendere il confronto quantitativo nel capitolo, la Tabella 5.6 riporta la MRE media sui checkpoint per i casi con $n = 10^7$ e seed 21041998. Per ciascuna coppia (k, d), la MRE media lungo il flusso è definita come:

$$\overline{\mathrm{MRE}}_{\mathrm{curve}} = \frac{1}{K_{\mathrm{chk}}} \sum_{i=1}^{K_{\mathrm{chk}}} \mathrm{MRE}(t_i), \quad K_{\mathrm{chk}} = \min\{n, 200\}.$$

k	d	MRE media (HLL)	MRE media (HLL++)	guadagno medio
10	10^5	0.025156	0.025156	0.0
10	10^6	0.027883	0.027883	$1.17 \cdot 10^{-8}$
10	$5 \cdot 10^6$	0.023185	0.023185	$-6.29 \cdot 10^{-8}$
10	10^7	0.020136	0.020136	$2.61 \cdot 10^{-7}$
12	10^5	0.013854	0.013854	0.0
12	10^6	0.013293	0.013293	$-1.01 \cdot 10^{-9}$
12	$5 \cdot 10^6$	0.013546	0.013546	$1.27 \cdot 10^{-7}$
12	10^7	0.016646	0.016645	$5.77 \cdot 10^{-7}$

Tabella 5.6: Confronto HLL/HLL++ sulla MRE media lungo il flusso per $n = 10^7$, seed 21041998. Le colonne MRE sono arrotondate a 10^{-6}, mentre il miglioramento è calcolato sui valori non arrotondati.

I grafici interattivi usati in questa analisi sono nei notebook:

- `notebooks/streaming_hll_hllpp_paper_analysis.ipynb`;
- `notebooks/streaming_seed_21041998_n10000000_by_d.ipynb`;

- `notebooks/hll_vs_hllpp_difference_n10000000.ipynb`.

5.7 Discussione dei risultati

I risultati ottenuti sono coerenti con la linea teorica discussa nei capitoli precedenti:

- HLL++ tende a ridurre bias ed errore relativo rispetto a HLL nei regimi in cui le correzioni pratiche sono rilevanti;
- HLL classico resta competitivo e, su cardinalità molto grandi, le due varianti possono diventare quasi indistinguibili;
- LogLog e Probabilistic Counting, pur utili come riferimenti storici, mostrano limiti pratici marcati con i parametri correnti su intervalli ampi di scala.

5.8 Limiti sperimentali e validità

I risultati vanno letti con i seguenti vincoli:

- per HLL/HLL++ il blocco completo disponibile usa un solo seed (21041998), mentre per LogLog/PC sono disponibili 5 seed;
- distribuzione testata: uniforme, flusso mescolato casualmente;
- confronto centrato su accuratezza statistica, senza misure complete di tempi di aggiornamento/interrogazione e occupazione empirica di RAM.

Questi limiti riducono la validità esterna del confronto e richiedono cautela nell'interpretazione causale. Il quadro resta comunque utile come evidenza preliminare e come guida per una campagna successiva più bilanciata.

5.9 Sintesi

In sintesi:

- il framework implementato permette una valutazione riproducibile, con verità $F_0(t)$ integrata nel dataset e metriche coerenti con la teoria;
- HLL++ risulta mediamente migliore di HLL nel blocco analizzato, soprattutto in regimi di piccola e media cardinalità;
- nei regimi ad alta cardinalità le due varianti convergono;
- i risultati forniscono una base operativa per estensioni future su operazioni di fusione, robustezza all'hash e famiglie di sketch non *count-distinct*.

6
Conclusioni

Volendo idea: sarebbe da prendere i dati in una stream continua e fare misurazione real-time di quello che accade, ovviamente non si puo' sapere il valore esatto in questo caso.

Pero' si potrebbe invece fare che la stream esiste gia' pero' vengono dati un valore alla volta, in modo tale da avere effettivamente i valori.

Pensare anche all'implementazione di un sistema reale di streaming di dati come Apache Kafka.

Bibliografia

[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] G. Cormode, "Count-min sketch," 2010, encyclopedia entry; local source: thesis/figures/cmencyc.pdf.

[19] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[20] S. Agarwal and A. Trachtenberg, "Approximating the number of differences between remote sets," 2006, technical report/manuscript; local source: thesis/figures/bloom.pdf.

[21] 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.

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

Ringraziamenti