DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

"Studio di algoritmi con predizione per la stima di elementi frequenti in uno stream"

Relatori: Prof. Leonardo Pellegrina
Prof. Fabio Vandin

Laureanda: Giulia Beraldo

Matricola 2032598

ANNO ACCADEMICO 2023 – 2024

Data di laurea 27 Settembre 2024

Desidero ringraziare tutti coloro che hanno contribuito alla realizzazione di questa tesi.

Per prima cosa, vorrei ringraziare i miei relatori, per i preziosi consigli fornitomi, per la disponibilità e la pazienza.

Ringrazio di cuore mia sorella Vittoria, per sopportare i miei momenti di sconforto e per strapparmi sempre un sorriso.

Il ringraziamento più sentito ed affettuoso va ai miei genitori. Grazie per il vostro sostegno costante, per aver sempre creduto in me, per i vostri sacrifici, per la vostra pazienza e per il vostro amore.

Nell'era dei big data, l'analisi efficiente di grandi volumi di dati richiede l'uso di strutture dati probabilistiche, che offrono un compromesso tra precisione e risorse computazionali. In questa tesi viene, innanzitutto, presentata la struttura dati Count-Min sketch, utilizzata per rappresentare in modo succinto un flusso di dati (data stream), permettendo di rispondere a diverse tipologie di query, come le point, range ed inner product query, in modo efficiente. Ciò che rende particolarmente vantaggioso l'utilizzo di tale struttura consiste nell'occupazione sublineare di spazio di memoria nel numero di elementi salvati. In altri termini, Count-Min sketch richiede meno memoria rispetto alle dimensioni dell'input (in questo caso del data stream) su cui opera. Questo sketch, infatti, appartiene alla famiglia dei "counting sketch", che sono strutture dati probabilistiche utilizzate per approssimare varie statistiche su grandi dataset in modo efficiente in termini di spazio e tempo. Per memorizzare, per esempio, i conteggi di ogni item, appartenente ad un particolare data stream, la strategia di mantenere un contatore separato per ogni elemento richiede una quantità eccessiva di spazio, specialmente se il flusso presenta dimensioni notevoli. Usando, invece, Count-Min sketch è possibile utilizzare meno spazio in memoria, al costo, tuttavia, di ottenere solo un conteggio approssimato (quindi non esatto) di ciascun item.
Le proprietà di questo sketch e il suo utilizzo nella risposta alle query specificate, sono descritte nei Capitoli 1 e 2.
Una volta implementata tale struttura dati, le cui dimensioni dipendono da due parametri, δ e ε, introdotti nel seguito, vengono effettuate delle analisi sperimentali volte a studiare come le prestazioni in termini di accuratezza della stima di Count-Min sketch variano con le dimensioni (modificando quindi i due parametri sopracitati). Il codice è disponibile nella seguente repository di GitHub https://github.com/giuliaberaldo2002/CMS.git.
In seguito, questa tesi esplora come la precisione delle stime possa essere migliorata attraverso l'integrazione di due tecniche: un oracolo perfetto e un predittore, il cui codice è reperibile al link sopracitato. Utilizzando queste due strategie, è possibile garantire maggiore accuratezza per il conteggio degli elementi che si presentano con maggiore frequenza e, al contempo, ridurre la possibilità di ottenere un errore notevole per un item che compare poco nel flusso. Ciò è spiegato, più nel dettaglio, nel Capitolo 5. Risultati simili si sarebbero potuti ottenere con algoritmi di machine learning, in grado di identificare le proprietà distintive degli heavy hitter. Nella Sezione 5.1.1 sono riportati i teoremi e le corrispondenti dimostrazioni che mostrano i limiti inferiori e superiori di accuratezza ottenuti sfruttando tale strategia.
Seguono, quindi, delle analisi sulle prestazioni dell'uso coadiuvato di Count-Min sketch con l'oracolo e con il predittore.

In conclusione l'obiettivo di questa tesi è quello di presentare la struttura dati Count-Min sketch, mostrarne le sue principali applicazioni e prestazioni, quando in input riceve un data stream modellato come una distribuzione di Zipf, e infine mostrare come le sue prestazioni possono essere migliorate se viene utilizzato un oracolo perfetto o un predittore, fornendo un risultato grafico dell'errore medio e dell'accuratezza nei tre casi.

Indice

1 Descrizione della struttura dati Count-Min Sketch 1

1.1 Modellizzazione del data stream 1

1.2 Introduzione alla struttura dati Count-Min sketch 4

1.2.1 Struttura di Count-Min sketch 5

1.2.2 Considerazioni sui valori di ε e δ 6

1.2.3 Sovrastima dei conteggi degli item 6

1.2.4 Descrizione della procedura di aggiornamento dei conteggi in Count-Min sketch 7

1.3 Notazione e concetti utilizzati 7

1.3.1 Simboli usati 8

1.3.2 Pair-wise e four-wise independence 8

1.3.3 Errore di stima dei conteggi 8

1.3.4 Distribuzione di Zipf 9

2 Principali tipologie di query e loro risposta usando Count-Min Sketch 11

2.1 Point Query 11

2.1.1 Caso turnstile non negativo 11

2.1.2 Caso turnstile generale 14

2.2 Inner Product Query 15

2.3 Range Query 17

2.4 Confronto tra tipologie di sketch 19

3 Applicazioni di Count-Min Sketch 23

3.1 Quantili nel modello turnstile 23

3.2 Heavy Hitter 24

3.2.1 Data stream cash register 24

3.2.2 Data stream turnstile 26

3.2.3 Heavy Hitter Gerarchici 27

3.3 Applicazioni di Count-Min sketch a task reali 27

3.3.1 Natural Language Processing 28

3.3.2 Controllo di flussi di reti 28

3.3.3 Database 29

3.3.4 Sicurezza nelle password . . . 30
3.3.5 Biologia Computazionale . . . 31
3.3.6 Ulteriori applicazioni . . . 31
3.3.7 Possibili future applicazioni . . . 32

4 Studio sperimentale delle prestazioni di Count-Min sketch 33
4.1 Descrizione dell'implementazione di Count-Min sketch . . . 33
4.2 Dipendenza dell'errore medio dalle dimensioni di Count-Min sketch . . . 34
4.3 Numero di elementi contati erroneamente in funzione delle dimensioni di Count-Min sketch . . . 40
4.4 Dipendenza del numero di match e mismatch dai parametri ε e δ . . . 42

5 Miglioramento delle prestazioni di Count-Min Sketch con tecniche di predizione 51
5.1 Introduzione di algoritmi di Machine Learning e modifiche apportate a Count-Min sketch . . . 52
5.1.1 Analisi . . . 53
5.2 Miglioramento di Count-Min sketch tramite l'utilizzo di un oracolo o di un predittore . . . 56
5.2.1 Oracolo . . . 58
5.2.2 Predittore . . . 60
5.2.3 Analisi Sperimentali ed Osservazioni . . . 63
5.3 Soluzioni alternative per aumentare l'accuratezza di Count-Min Sketch . . . 73
5.3.1 Augmented Sketch . . . 73
5.3.2 k-ary sketch . . . 74
5.3.3 gSketch . . . 75
5.3.4 Frequency-Aware Counting . . . 76

6 Conclusioni 77

Capitolo 1

Descrizione della struttura dati Count-Min Sketch

Lo scopo di questo capitolo è quello di presentare la struttura dati Count-Min sketch. Prima di introdurre tale sketch, però, si formalizza il concetto di data stream. Esso viene modellato come un vettore di dimensione pari al numero di item che lo compone, tale per cui ciascuna entry contiene un diverso elemento ad ogni istante. Quindi, si illustra l'array bidimensionale che implementa Count-Min sketch: ad ogni riga è associata una funzione di hash distinta (implementata nel codice tramite il metodo Multiple Add Divide, più brevemente MAD). Il conteggio di ciascun elemento è quindi mappato in ogni riga da una funzione di hash differente in una cella diversa: in totale, il conteggio di ogni elemento viene salvato, od aggiornato, in d celle distinte, una per riga.

1.1 Modellizzazione del data stream

Come introduzione all'argomento, si cerca di fornire una definizione di data stream, e si modella poi quest'ultimo riprendendo la notazione illustrata in [23].
Come descritto in [34], un data stream può essere interpretato come dati di input che giungono ad un tasso molto alto, tale per cui viene messa sotto pressione l'infrastruttura di comunicazione e di calcolo. Potrebbe, perciò, essere difficile trasmettere l'intero input, calcolare funzioni complesse su porzioni di dimensioni notevoli dell'input e salvare i dati temporaneamente o a lungo termine. Più intuitivamente, un data stream consiste in una sequenza di pacchetti dati che portano informazione durante la trasmissione ([39]). Il flusso viene, dunque, rappresentato come un vettore a, di dimensione n (uguale al numero di item nello stream) e tale per cui, al generico istante di tempo t, presenta come stato: $a(t) = [a_1(t),\ a_2(t),\ \dots, a_n(t)]$. L'item del vettore $a_i(t)$ memorizza quindi la frequenza con cui appare il simbolo i nel data stream all'istante di tempo t, $\forall i \in \{1, 2, \dots, n\}$. Nell'istante iniziale, $t = 0$, quando non è ancora stato registrato alcun dato, perciò, a si configura come vettore nullo, cioè $a(0) = [0, 0, \dots, 0]$. Ogni aggiornamento del conteggio di una generica entry, per esempio la i-esima all'istante t (indicata brevemente con i_t), viene

dunque interpretato come una coppia $(i_t,\ c_t)$. L'aggiornamento del conteggio dell'item osservato nel flusso all'istante di tempo t, indicato con a_{i_t} si traduce quindi nei seguenti assegnamenti:

$$a_{i_t}(t) = \ a_{i_t}(t-1) + c_t,$$
$$a_{i'}(t) = \ a_{i'}(t-1)\ ,\ i' \neq i_t$$

dove $a_{i_t}(t-1)$ rappresenta il numero di occorrenze di i_t (cioè l'elemento che compare nel flusso all'istante t), nell'istante precedente all'aggiornamento, c_t l'incremento (o eventualmente la diminuzione) nel conteggio di a_i , mentre $a_{i'}(t)$ i valori dei conteggi delle rimanenti entry del vettore, che rimangono invariati.
Come mostrato in [23] e [34], in base ai possibili valori che c_t può assumere, si distinguono i seguenti modelli di data stream:

- Cash register: sussiste la condizione $c_t > 0$, cioè, il flusso è costituito dall'aggiungersi di elementi, e non da rimozioni. Un esempio pratico di questo tipo di flusso consiste nel controllare gli indirizzi IP che accedono ad un particolare server web ([34]); infatti, una macchina, denotata da uno specifico indirizzo IP, potrebbe accedere ripetutamente lo stesso server (specie se viene utilizzato il protocollo HTTP/1.0, in cui ogni richiesta necessita la creazione di una nuova connessione). In questo caso, infatti, nel flusso possono avvenire solo inserimenti, e non cancellazioni, di indirizzi;

- Turnstile: nel flusso possono verificarsi sia aggiunte che rimozioni, indicate rispettivamente dalle condizioni $c_t > 0$ e $c_t < 0$. In questa modalità, si distingue ulteriormente tra il caso in cui i conteggi (quindi gli elementi di a_i) possano assumere valori negativi (caso generale) da quello in cui ciò non può accadere (caso non negativo). Come accennato in [23], il secondo caso viene in genere utilizzato per monitorare il contenuto di database (in questo caso è infatti possibile eliminare solo un dato precedentemente inserito) oppure il traffico IP visto da un link della rete. Il primo invece trova applicazioni in sistemi distribuiti, dove vengono eseguite operazioni come la sottrazione di vettori (che rappresentano, per esempio, due flussi di tipo cash register).

Come ultima nota, si osserva che lo stream così descritto è dinamico, in quanto viene presupposto che i dati siano generati e ricevuti in modo continuo, senza un prefissato punto di fine. Negli esperimenti effettuati, invece, sono stati utilizzati dei flussi di dimensione fissata, salvati per comodità in file. Ciò, comunque non comporta che i procedimenti per l'aggiornamento di Count-Min sketch e per rispondere alle varie query, descritti a pagina 7, siano differenti.
A questo punto, sempre come presentato in [23], una query può invocare una funzione che, in un qualsiasi istante di tempo t, calcola un risultato a partire dal vettore $a(t)$. In

particolare, esistono tre tipi principali di query, che vengono usate in molteplici applicazioni di data stream e che utilizzano come operandi i vettori a e b:

1. **Point query** (denotata *Q(i)*) che ritorna una stima della frequenza dell'elemento i di a, a_i
2. **Range query** (chiamata *Q(l,r)*) che ritorna un'approssimazione della somma dei conteggi degli item compresi tra due estremi, in questo caso il l-esimo e l'r-esimo: $\sum_{i=l}^{r} a_i$
3. **Inner product query** (denotata *Q(a,b)*) usata per approssimare il prodotto interno tra i due vettori a e b, vale a dire $a \bigodot b = \sum_{i=1}^{n} a_i b_i$

Il paper [23] fa inoltre notare come una range query non è altro che una somma di point query, e sia le point query che le range query sono un caso particolare di inner product query. Per di più, oltre a trovare applicazione nel campo dei data stream, queste query vengono utilizzate anche nell'ambito dei database, dove le point e range query permettono di riassumere distribuzioni di dati, mentre le inner product query consentono di approssimare il join size delle relazioni ([23]).
Esistono, poi, altre tipologie di query, che permettono di eseguire funzioni più complesse sui data stream. Ad esempio, ricordando che il numero totale dei conteggi nel vettore a è dato da $L_1 = ||a||_1 = \sum_{i=1}^{n} |a_i(t)|$, o più in generale $L_p = |a|_p = \sqrt[p]{(\sum_{i=1}^{n} |a_i(t)|^p)}$, (norma di a) alcune di queste sono:

- $\phi - Quantiles$: sono una generalizzazione dei quantili tradizionali (come i mediani e i quartili), che consentono di spartire una distribuzione in percentuali specifiche definite dal parametro ϕ, anziché da percentuali fisse ([23]). Più formalmente, i $\phi - Quantiles$ di una sequenza ordinata di cardinalità $N = ||a||_1 = \sum_{i=1}^{n} |a_i(t)|$ consistono negli item di rango $k\ \phi\, ||a||_1$ per $k = 0, \ \ldots, \frac{1}{\phi}$([23], [20]). Un'approssimazione consiste nell'accettare all'interno di un $\phi - Quantiles$ qualsiasi intero che presenta un valore compreso tra un item di valore $(k\phi - \varepsilon)\, ||a||_1$ ed uno di valore $(k\phi + \varepsilon)\, ||a||_1$, per qualche specifico $\varepsilon < \ \phi$, che rappresenta quindi l'errore massimo ammesso.
- *Heavy Hitters*: si configurano come gli elementi in un insieme generico di item o in un data stream che compaiono significativamente in modo più frequente degli altri ([23]). Per questo sono definiti come gli elementi più " pesanti". Più precisamente, i $\phi - heavy\ hitters$ di un multiset di cardinalità $||a||_1$ in cui i valori variano tra 1…n, consistono negli item la cui molteplicità eccede la frazione ϕ della cardinalità totale, cioè $a_i \geq \ \phi\, ||a||_1$. In questo caso, un'approssimazione è data accettando un qualsiasi indice i tale per cui $a_i \geq (\phi - \varepsilon)\, ||a||_1$ per qualche $\varepsilon < \phi$, pari all'errore massimo ammesso. Questa tipologia di quantili sono anche definiti $\varepsilon - approximate\ \phi\ quantiles$ ([20]).

Come suggerito in [23], per le spiegazioni seguenti viene assunto il modello RAM (Random Access Machine), in cui si accetta che il calcolatore possa memorizzare interi fino a $\max \{||a||_1\,,\ n\}$ e tale per cui operazioni definite standard vengano eseguite in un tempo costante. Si conta perciò lo spazio in termini di numero di parole (words), qui pari agli item del data stream, e si computa il tempo in termini di numeri di operazioni "base".
Ripetendo, l'obiettivo dell'utilizzo di strutture dati sketch è quello di risolvere le query sopraelencate in un tempo e in uno spazio al più polilogaritmico rispetto alla taglia dell'input, (n o $||a||_1$).
Da questo si può osservare che, dal momento che lo spazio deve essere sublineare nelle dimensioni del data stream, la struttura dati usata dall'algoritmo per memorizzare i dati di input si configura come una compressione dei dati, da cui ne deriva che ciascuna funzione che manipola il vettore a fornisce una stima del calcolo e non un risultato preciso.
Di conseguenza, gli algoritmi che manipolano strutture di sketch si configurano come approssimativi e probabilistici.
Per le varie analisi, si necessita poi l'introduzione di due parametri:

- ε : equivale al fattore specifico che limita o controlla l'errore nell'elaborare una query.
- δ : coincide con la probabilità che si verifichi un errore nella risposta alla query.

In conclusione, in questa breve introduzione si è descritta la notazione usata per rappresentare il flusso di dati da elaborare e le varie tipologie di query ed operazioni che possono essere effettuate su di esso: le principali sono point query, range query ed inner product query, a cui si aggiungono poi operazioni più complesse, come i $\phi - Quantiles$ e gli *Heavy Hitter*. Infine, si sono descritti i due parametri ε e δ, che, come illustrato nella prossima sezione, determinano le dimensioni della matrice con la quale viene implementato il Count-Min sketch.

1.2 Introduzione alla struttura dati Count-Min sketch

L'obiettivo di questa sezione è quello di mostrare come è strutturato Count-Min sketch. Si illustra brevemente l'etimologia del nome, per poi descrivere come i parametri ε e δ determinano le dimensioni della matrice utilizzata. Brevemente, Count-Min sketch si configura come un array bidimensionale in cui ad ogni riga è associata una funzione di hash differente. Nel momento in cui, all'istante t, deve essere aggiornato il conteggio dell'elemento i_t, ciascuna funzione di hash mappa questo (aggiornando conseguentemente il corrispondente contatore) in una cella della matrice differente. Essendo le collisioni inevitabili, per calcolare l'effettivo conteggio di un item si prende il valore minore fra tutti.

1.2.1 Struttura di Count-Min sketch

Come fatto notare in [23], il nome della struttura dati Count-Min sketch deriva innanzitutto da due operazioni di base che vengono effettuate per rispondere ad una point query:

1. conteggio delle occorrenze di ciascun dato
2. calcolo del minimo fra i conteggi per uno stesso item

L'obiettivo di tale struttura dati è quella, considerato un data stream, di contare la frequenza dei diversi tipi di eventi nel flusso. In un qualsiasi momento lo sketch può essere interrogato riguardo al conteggio attuale di un particolare evento ed esso ritorna una stima di questo conteggio, con una certa distanza da quello effettivo, caratterizzata da una certa probabilità.
Come descritto in [23], la sua definizione formale è la seguente:
Un Count-Min sketch di parametri $(\varepsilon,\ \delta)$ è rappresentato da un array bidimensionale di conteggi con w colonne e d righe, tali per cui $w = \left\lceil \frac{e}{\varepsilon} \right\rceil$ e $d = \left\lceil \ln\left(\frac{1}{\delta}\right) \right\rceil$.
Nell'istante iniziale, quando nello sketch non è stato inserito alcun conteggio (si ricorda che, per come è stato modellato il flusso (1.1), in $t = 0$ il vettore $a(0) = \mathbf{0}$), ciascuna entry dell'array è inizializzata a zero. Inoltre, sono presenti d funzioni di hash (una per ciascuna riga della matrice) indipendenti (più specificatamente, queste funzioni godono di indipendenza a coppie, "pairwise", descritte nella sezione 1.3.2) e casuali:

$$h_1 \dots h_d : \{1 \dots n\} \to \{1 \dots w\}$$

Nel codice della repository GitHub le funzioni di hash sono state implementate secondo il metodo Multiply Add Divide (MAD), tale per cui, data la generica stringa i, la funzione di hash della j-esima riga mappa i nella colonna $h_j(i) = ((a \cdot i + b)\ \%p)\ \%N$, dove a e b sono termini casuali appartenenti all'intervallo $[0, p-1]$, p è un numero primo e tale per cui $p > N$, mentre N è un intero (qui posto pari a w in modo tale che i conteggi degli elementi possano essere mappati entro i range della matrice).

Ciascun elemento del data stream viene, dunque, mappato dalla funzione di hash h_j della riga j-esima in una particolare colonna: il conteggio dell'item i, in questa riga, viene perciò salvato nella cella di coordinate $(j, h_j(i))$. Questa procedura è simultaneamente eseguita per ognuna delle d righe che costituiscono lo sketch. In ciascuna cella, conseguentemente, viene memorizzata una stima del conteggio per un dato item, o per un insieme di item, nel caso si verifichino collisioni. Count-Min sketch appartiene, quindi, alla classe degli "hashing-based algorithm", dal momento che il conteggio di ogni elemento viene mappato da una o più funzioni di hash in una particolare struttura dati, come una matrice. Altre strutture facenti parte di tale gruppo sono, ad esempio, *Count-Sketch*, descritto a pagina 21 e *Count-Median sketch*, il quale, come illustrato in [7], si configura come una variante di Count-Min sketch che differisce da quest'ultimo solo per il fatto che nel calcolo del

conteggio di un item viene preso in considerazione il valore mediano piuttosto che quello minimo.
Lo spazio usato da tale struttura dati è costituito dall'array di wd contatori, e da d funzioni di hash (necessarie per conoscere la mappatura dei vari item nello sketch).

Infine, uno sketch , come fatto definito in [12] e [21], può essere considerato come una proiezione lineare casuale del vettore di input, a. Ricordando che la componente i-esima del vettore che modella il flusso corrisponde al conteggio dell'elemento i, lo sketch coincide, quindi, con il prodotto di questo vettore con una matrice, generalmente scelta in modo da occupare poco spazio. In altri termini, lo sketch è costruito a partire dal prodotto interno tra il vettore a e un insieme opportuno, di dimensioni ristrette, di vettori casuali, generati usando dei "seeds" che occupano poco spazio.

1.2.2 Considerazioni sui valori di ε e δ

Dalla definizione data si può notare che la precisione dell'approssimazione dipende dal numero di colonne w dell'array e dal numero di funzioni d di hash utilizzate. Quindi, per migliorare la precisione, si potrebbe pensare di ridurre i valori dei parametri w e d, comportando però, in questo modo, l'aumento dello spazio richiesto per salvare tale struttura dati. Si osserva, infatti, che il numero di colonne cresce in modo inversamente proporzionale al diminuire del parametro ε, mentre la quantità di righe presenta una dipendenza logaritmica dall'inverso di δ. Ad ogni dimezzamento del primo parametro, quindi, le dimensioni di Count-Min sketch raddoppiano. Dimezzando δ, invece, la grandezza dello sketch aumenta di un fattore pari a $1 + \frac{\ln 2}{\ln(1/\delta_0)}$.

1.2.3 Sovrastima dei conteggi degli item

Un'altra caratteristica fondamentale di Count-Min sketch consiste nel fatto che esso non sottostima mai il conteggio effettivo di un elemento. Dunque, considerata $\hat{a}_i$ la frequenza stimata per l'elemento i e a_i quella effettiva, in un generico istante di tempo, si verifica che $\hat{a}_i \geq a_i$. La dimostrazione è riportata nella sezione 2.1.1.
Questa proprietà garantisce, dunque, che i conteggi riportati dallo sketch non siano mai inferiori a quelli effettivi: ciò presenta diverse implicazioni, specialmente nell'analisi dei dati in tempo reale e nelle applicazioni in cui è importante ottenere delle stime conservative, considerando, cioè, il caso peggiore. Più concretamente, questa proprietà è utile in situazioni in cui è dannoso sottostimare il numero di occorrenze di un determinato item, come nell'ambito di sicurezza informatica, dove un errore di sottostima potrebbe far ignorare al sistema un' attività sospetta. Ad esempio, nel caso di rilevamento di possibili intrusioni o attacchi DDoS, è conveniente sfruttare Count-Min sketch per salvare i conteggi degli indirizzi IP in modo tale da capire quali di questi potrebbero rappresentare una minaccia, evitando il rischio di non rilevarli per via di una sottostima.

1.2.4 Descrizione della procedura di aggiornamento dei conteggi in Count-Min sketch

Per quanto riguarda l'aggiornamento, la procedura seguita, come descritta in [23], è la seguente: quando all'istante di tempo t, giunge nel flusso l'item i_t, rappresentato dalla coppia $(i_t,\ c_t)$, c_t è sommato ad un determinato conteggio per ciascuna riga dell'array bidimensionale. Più precisamente, all'arrivo di un nuovo item i, si applica per ciascuna riga la corrispondente funzione di hash per ottenere un indice di colonna $z = h_j(i)$. A questo punto, si aggiunge c_t (che nel caso più generale, vale a dire nel modello turnstile, può essere sia positivo che negativo) al valore salvato nella cella di coordinate $(i, h_j(i))$, cioè nella riga j e colonna z.
Più formalmente, per aggiornare la matrice di Count-Min sketch con il conteggio del nuovo item nel flusso si effettua il seguente assegnamento nella riga j, $\forall\ 1 \leq j \leq d$,

$$count\,[j,\ h_j\,(i_t)] \leftarrow\ count\,[j,\ h_j\,(i_t)] + c_t$$

.

Infine, una visualizzazione grafica di tale struttura dati è la seguente:

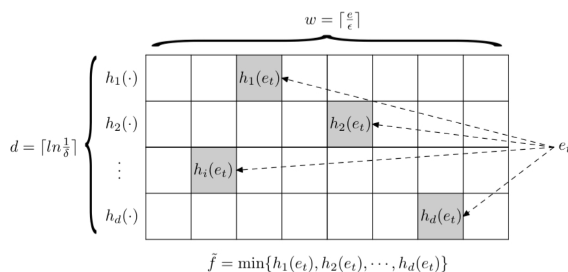


Figura 1.1: Rappresentazione visiva della struttura dati Count-Min Sketch, immagine presa da [43]

In questa sezione, riassumendo, è stato presentato Count-Min sketch. Più nel dettaglio, sono stati introdotti i parametri ε e δ, che determinano rispettivamente il numero di colonne e di righe dell'array bidimensionale su cui lo sketch si basa. A quest'ultimo si aggiungono d funzioni di hash, una per riga, che mappano ciascuna gli item del data stream in una particolare colonna. Dopo alcune considerazioni sulla dipendenza delle dimensioni di Count-Min sketch dai due parametri sopracitati, è stata poi illustrata la procedura di aggiornamento del conteggio di ciascun elemento, nel caso più generale, cioè considerando il modello turnstile per il flusso.

1.3 Notazione e concetti utilizzati

In questa breve sezione, viene illustrata la notazione utilizzata e alcuni concetti fondamentali, come l'errore di stima nei conteggi e la distruzione di Zipf, utilizzati in seguito per

l'analisi. Quetse nozioni sono state riprese da [24]:

1.3.1 Simboli usati

In questa sottosezione si illustra la notazione che verrà poi utilizzata nel seguito dell'elaborato, in particolare nella sezione 5.1.1. I simboli principalmente usati sono i seguenti:

- B: rappresenta la dimensione complessiva delle strutture dati utilizzati per memorizzare i conteggi degli item del data;
- B_r: costituisce la dimensione complessiva dei bucket in cui sono salvati gli elementi più frequenti, definiti heavy hitter.

1.3.2 Pair-wise e four-wise independence

Le funzioni di hash utilizzate in Count-Min sketch, soddisfano la proprietà di pair-wise independence, letteralmente indipendenza a coppie. Formalmente, una famiglia di funzioni di hash che soddisfa questa caratteristica presenta la seguente forma: $h(i) = (a \cdot i + b)\ \%\ p$, in cui a e b sono delle costanti scelte tra 0 e $p - 1$, con p primo ([10]). Da ciò ne consegue che la probabilità che due item, i e j, vengano mappati nello stesso bucket risulta essere pari a $\frac{1}{p}$. Il metodo Multiply-Add-Divide, con cui si sono implementate le funzioni di hash per gli esperimenti (https://github.com/giuliaberaldo2002/CMS.git), appartiene, dunque, a questa famiglia. Tali funzioni presentano, infatti, la seguente forma: $h(i) = ((a \cdot i + b)\ \%p)\ \%N$, in cui N viene posto pari a w per fare in modo che, effettivamente, i conteggi degli elementi vengano mappati in $[0, w - 1]$.
Le funzioni di hash caratterizzate, invece, da four-wise independence (indipendenza a quattro), sono tali per cui per cui qualsiasi gruppo di quattro elementi distinti viene mappato in modo casuale e indipendente. In altri termini, una famiglia tipica di funzioni che soddisfa questa proprietà si attesta ad essere pari a $h(x) = (ai^3 + bi^2 + ci + d)\ \%\ p$ ([10]). Questa famiglia di hash function viene utilizzata nel *tug-of-war sketch*, presentato nella sezione 2.4.
Queste funzioni presentano, dunque, il vantaggio di poter essere calcolate velocemente, con prestazioni migliori di altre, utilizzate specialmente in crittografia, come MD5 o SHA-1 ([10]). Di conseguenza, le operazioni di aggiornamento dei conteggi e di query in uno sketch che utilizza questo tipo di funzioni di hash, sono computazionalmente efficienti.

1.3.3 Errore di stima dei conteggi

L'errore di stima è pari a

$$e_i = |\hat{a}_i - a_i|$$

dove a_i coincide con il conteggio effettivo mentre $\hat{a}_i$ rappresenta il conteggio stimato dell'elemento i, cioè il minimo tra i conteggi per l'item i salvati in Count-Min sketch. In

questo caso, si considera un data stream statico, di dimensioni fissate, quindi la dipendenza dal tempo è trascurabile. Se così non fosse (in presenza, cioè, di un flusso dinamico), la definizione di errore dovrebbe essere riscritta come $e_i(t) = |\hat{a}_i - a_i|(t)$. Questa osservazione si applica anche alle definizioni successive. Per misurare l'errore complessivo tra le frequenze effettive $A = a_1, a_2 \ldots a_n$ e le corrispondenti stime $\hat{A} = \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_n$, si utilizza il concetto di errore atteso $E_{i \sim D[e_i]}$, dove $\mathcal{D}$ modella la distribuzione statistica delle query richieste alla struttura dati in esame. In altri termini, $\mathcal{D}$ descrive la probabilità con cui ogni query viene posta. In questo caso, si considera $\mathcal{D}$ pari alla distribuzione in input, quindi, per un qualsiasi item j si ottiene $Pr_{i \sim \mathcal{D}}[i = j] = \frac{a_j}{N}$, dove $N = ||a||_1$ è la somma di tutti i conteggi. Questo porta ad un errore di stima $\hat{A}$ rispetto a A pari a:

$$Err(A, \hat{A}) = \frac{1}{N} \sum_{i \in \{1,\ldots,n\}} |\hat{a}_i - a_i| \cdot a_i$$

Rappresentando questa grandezza nella "forma (ε, δ)", si ottiene che per ogni elemento i vale che $Pr[|\hat{a}_i - a_i| > \varepsilon N] < \delta$ ([24]).
Nel caso più semplice, invece, si suppone che ciascuna query sia equiprobabile, per cui l'errore medio può in questo caso essere calcolato come

$$Err[A, \hat{A}_i] = \frac{1}{n} \sum_{i=1}^{n} |\hat{a}_i - a_i|$$

1.3.4 Distribuzione di Zipf

Nell'analisi effettuata per studiare l'ottimizzazione di Count-Min sketch con tecniche di machine learning (capitolo 5), si assume che i conteggi degli item del data stream siano distribuiti seguendo una distribuzione di Zipf. Questo significa che, a seguito di un ordinamento degli elementi per conteggi non crescenti, $a_{i_1} \geq a_{i_2} \geq \cdots \geq a_{i_n}$ si ricava che $a_{i_j} \propto \frac{1}{j}$. Più precisamente, la legge di Zipf afferma che, considerato un insieme di elementi ordinati per conteggi non crescenti, la frequenza dell'i_j-esimo elemento è pari a

$$a_{i_j} = \frac{C}{j^\alpha}$$

[28] dove C è una costante di normalizzazione (in genere posta a 1) e α è un parametro tipicamente unitario.
In altri termini, il secondo termine più frequente appare metà delle volte del termine più frequente, il terzo termine un terzo delle volte del primo, ecc.. Da questo si evince che tale distribuzione modella un data stream di tipo cash register, in cui non si possono verificare rimozioni di elementi. Riassumendo, una distribuzione Zipfiana presenta pochi elementi molto comuni e la restante maggior parte è costituita da item rari.

In questa sezione, è stato presentato il significato dei principali simboli utilizzati nel seguito dell'elaborato, quali il concetto di bucket distinto e bucket appartenente al Count-

Min sketch. Successivamente, si sono illustrati i concetti di errore medio e distribuzione Zipfiana.

Capitolo 2

Principali tipologie di query e loro risposta usando Count-Min Sketch

Lo scopo di questo capitolo è quello di presentare come Count-Min sketch possa essere utilizzato per rispondere a point, range e inner product query. Il codice correlato si focalizza solamente sulla soluzione per la prima tipologia. Per questo motivo, anche le dimostrazioni sulla correttezza dei postulati, riguardanti le stime dei conteggi, sono mostrate solo per le point query. Successivamente, si effettua un breve confronto tra Count-Min sketch ed altre tipologie di sketch, mostrando le caratteristiche che denotano Count-Min sketch come migliore da un punto di vista di accuratezza e complessità temporale e spaziale.

2.1 Point Query

Come accennato all'inizio, l'obiettivo di una point query $Q(i)$ consiste nel determinare il numero di occorrenze dell'item i. In questa sezione si presentano, di conseguenza, le procedure seguite per rispondere a questa tipologia di query, considerando, dapprima, solo il caso di entry non negative del vettore a (modello turnstile non negativo) e, in seguito, l'eventualità di elementi minori strettamente di zero nel vettore che modella il flusso. In entrambe i casi vengono presentati, inoltre, limiti inferiori e superiori riguardo alla stime del conteggio per ciascun item.

2.1.1 Caso turnstile non negativo

Considerando inizialmente solo il caso non negativo, e sfruttando Count-Min sketch per memorizzare i conteggi dei vari elementi dello stream, la risposta approssimata alla query *Q(i)* è data da

$$\hat{a}_i = \min_j count\,[j,\ h_j(i)]$$

Questo significa che fra tutti i d conteggi per i, salvati nella matrice, viene considerato il valore minimo ([23]).

Come mostrato in [23] e come accennato nella sezione 1.2.3, tale approssimazione ha la seguente garanzia: $a_i \leq \hat{a}_i$. Il valore a_i è il conteggio effettivo dell'elemento i nel data stream. Dunque, il numero di occorrenze di un item viene al più sovrastimato.
Inoltre, con probabilità almeno pari a $1 - \delta$, vale che $\hat{a}_i \leq a_i + \varepsilon \,||a||_1$. Vale, perciò, la seguente catena di disequazioni:

$$a_i \leq \hat{a}_i \leq a_i + \varepsilon \,||a||_1$$

con probabilità almeno uguale a $1 - \delta$.

Per quanto riguarda la prova della seconda diseguaglianza, si riporta la dimostrazione fornita in [23].

Dimostrazione. Si introducono, innanzitutto, delle variabili indicatrici $I_{i,j,k}$, che assumono valore unitario se $i \neq k \wedge h_j(i) = h_j(k)$, altrimenti sono poste pari a 0. In altri termini, la variabile $I_{i,j,k}$ verifica se per due elementi distinti, i e k si verifica una collisione nella riga j. Se questo è il caso, allora suddetta variabile assume valore unitario, altrimenti valore nullo.
A questo punto, assumendo che $i \neq k$ e applicando la definizione di valore atteso, si ottiene che $\mathbb{E}[I_{i,j,k}] = 1 \cdot \Pr[h_j(i) = h_j(k)] + 0 \cdot \Pr[h_j(i) \neq h_j(k)] = \Pr[h_j(i) = h_j(k)]$. A causa dell'indipendenza a coppie ("pairwise") delle funzioni di hash, è possibile considerare le diverse probabilità come uniformi. Se, poi le funzioni di hash fossero perfette, vale a dire che ogni possibile output per un certo valore in ingresso avesse la stessa probabilità di essere prodotto, allora $Pr[h_j(i) = h_j(k)] = \frac{1}{range(h_j)}$. In genere, tuttavia, le funzioni di hash non soddisfano questa condizione, quindi $\frac{1}{range(h_j)}$ rappresenta un limite superiore. Dalle seguenti considerazioni, si ricava che

$$\mathbb{E}[I_{i,j,k}] = \Pr[h_j(i) = h_j(k)] \leq \frac{1}{range(h_j)} = \frac{\varepsilon}{e}$$

Per il generico item i, mappato nella riga j, si definisce, quindi, la variabile $X_{i,j} = \sum_{k=1}^{n} I_{i,j,k} a_k$. Tale variabile indica quanto il conteggio stimato dell'item i si discosta dal conteggio effettivo a causa delle collisioni con le occorrenze di altri elementi. Dal momento che tutti i conteggi, a_i sono non negativi, ne consegue che $X_{i,j} \geq 0$. Per costruzione, poi, si ricava che $count[j, h_j(i)] = a_i + X_{i,j}$. Perciò, $\min_j count[j, h_j(i)] \geq a_i$. Questo dimostra la prima disequazione, cioè il fatto che Count-Min sketch non può sottostimare il conteggio delle occorrenze dei vari item.

Per la seconda diseguaglianza si osserva, innanzitutto, che, dalla definizione di $X_{i,j}$, segue che

$$\mathbb{E}[X_{i,j}] = \mathbb{E}\left[\sum_{k=1}^{n} I_{i,j,k} a_k\right]$$

A questo punto, sfruttando la proprietà di linearità del valore atteso, si deduce che

$$\mathbb{E}\left[\sum_{k=1}^{n} I_{i,j,k} a_k\right] \leq \sum_{k=1}^{n} a_k \mathbb{E}[I_{i,j,k}]$$

quindi, applicando il limite superiore di $E[I_{i,j,k}]$, mostrato nella prima parte di questa dimostrazione, e considerando che la somma dei conteggi dei vari item è pari a $||a||_1$, si ricava che

$$\sum_{k=1}^{n} a_k \mathbb{E}[I_{i,j,k}] \leq \frac{\varepsilon}{e}||a||_1$$

Riassumendo,

$$\mathbb{E}[X_{i,j}] = \mathbb{E}\left[\sum_{k=1}^{n} I_{i,j,k} a_k\right] \leq \sum_{k=1}^{n} a_k \mathbb{E}[I_{i,j,k}] \leq \frac{\varepsilon}{e}||a||_1$$

Quindi, si osserva che dimostrare che $\Pr[\hat{a}_i \leq a_i + \varepsilon||a||_1] \geq 1 - \delta$, equivale a provare che $1 - \Pr[\hat{a}_i \leq a_i + \varepsilon||a||_1] \leq \delta$, che a sua volta coincide con la seguente disequazione $\Pr[\hat{a}_i > a_i + \varepsilon||a||_1] \leq \delta$.
Ricordando, inoltre, che $\hat{a}_i$ si configura come il valore minimo tra le d stime del conteggio di i, allora, si deduce che

$$\begin{aligned}\Pr[\hat{a}_i > a_i + \varepsilon||a||_1] &= \Pr[\forall_{1\leq j\leq d} count[j, h_j(i)] > a_i + \varepsilon||a||_1] \\ &= \Pr[\forall_{1\leq j\leq d} a_i + X_{i,j} > a_i + \varepsilon||a||_1]\end{aligned}$$

applicando in quest'ultimo passaggio la definizione di $count[j, h_j(i)]$. Semplificando a_i e ricordando che $\varepsilon||a||_1 \geq \mathbb{E}[X_{i,j}]e$ (deriva dalla disequazione dimostrata per $\mathbb{E}[X_{i,j}]$, si riscrive, perciò, che $\Pr[\forall_{1\leq j\leq d} a_i + X_{i,j} > a_i + \varepsilon||a||_1 = \Pr[\forall_{1\leq j\leq d} X_{i,j} > e\mathbb{E}[X_{i,j}]]$
A questo punto, si richiama il concetto di disuguaglianza di Markov. Come riportato in [25], essa stabilisce che, data una qualsiasi variabile aleatoria X non negativa, con aspettazione finita, vale che $\Pr[X \geq a] \leq \mathbb{E}[X]/a$. Applicando questa disequazione nel caso di un singolo conteggio (ad una sola riga), si ottiene

$$\Pr[X_{i,j} > e\mathbb{E}[X_{i,j}]] \leq \frac{\mathbb{E}[X_{i,j}]}{e\mathbb{E}[X_{i,j}]} = \frac{1}{e}$$

Nel caso più generale, invece, si ottiene che

$$\Pr[\forall_{1\leq j\leq d} X_{i,j} > e\mathbb{E}[X_{i,j}]] = \prod_{j=1}^{d} \Pr[X_{i,j} > e\mathbb{E}[X_{i,j}]] \leq \frac{1}{e^d}$$

. Si è posta una disuguaglianza stretta perché si considera una stima conservativa, al caso peggiore. Come ultimo passaggio, si ricorda che, per definizione, $d = \lceil \ln \frac{1}{\delta} \rceil$. Da ciò segue che $\frac{1}{e^d} \leq \delta$, e, dunque, $\Pr[\forall_{1\leq j\leq d} X_{i,j} > e\mathbb{E}[X_{i,j}]] \leq \delta$, concludendo dunque la dimostrazione.
Si riscrivono, infine, in forma più compatta le disequazioni presentate, che portano alla

dimostrazione della seconda disequazione:

$$\begin{aligned}\Pr[\hat{a}_i > a_i + \varepsilon||a||_1] &= \Pr[\forall_{1\le j\le d} count[j, h_j(i)] > a_i + \varepsilon||a||_1] \\ &= \Pr[\forall_{1\le j\le d} a_i + X_{i,j} > a_i + \varepsilon||a||_1] \\ &= \Pr[\forall_{1\le j\le d} X_{i,j} > e\mathbb{E}[X_{i,j}]] \le e^{-d} \le \delta\end{aligned}$$

□

Riguardo alle prestazioni, il tempo necessario per produrre la stima si attesta a $O(d) = O\left(\log \frac{1}{\delta}\right)$, poiché trovare il conteggio minimo può essere effettuato in un tempo lineare nei d valori considerati. Lo stesso limite temporale si applica anche per gli aggiornamenti.
Una nota deve essere effettuata per quanto riguarda l'utilizzo del numero di Nepero, e, nel calcolo della larghezza dell'array: come suggerito in [23], si può scegliere di porre $w = \frac{b}{\varepsilon}$ e $d = \log_b \frac{1}{\delta}$, per un qualsiasi $b > 1$. Scegliendo $b = e$, si minimizza lo spazio usato, dal momento che risolve $\frac{d(wd)}{db} = 0$ con un costo di $\left(2 + \frac{e}{\varepsilon}\right) \ln \frac{1}{\delta}$ entry. Per le implementazioni si suole usare un altro valore intero, in modo tale da ottenere calcoli e aggiornamenti più veloci.

2.1.2 Caso turnstile generale

I bound presentati nella sezione precedente sono validi solo per il caso in cui gli elementi di a siano positivi o nulli (caso turnstile non negativo). Più in generale, invece, nell'eventualità di entry negative, si risponde alla point query $Q(i)$ con il valore $\text{mediana}_j\, count\,[j,\ h(j)]$. In tal caso, con probabilità $1 - \delta^{\frac{1}{4}}$, vale che

$$a_i - 3\varepsilon\,||a||_1 \le\ \hat{a}_i \le a_i + 3\varepsilon\,||a||_1$$

Dimostrazione. La dimostrazione di queste disuguaglianze è la seguente: si osserva, per prima cosa, che

$$\mathbb{E}[|count[j, h_j(i)] - a_i|] \le \mathbb{E}\left[\sum_{k=1}^{n} I_{i,j,k} a_k\right] \le \frac{\varepsilon}{e}||a||_1$$

per la dimostrazione vista per il caso turnstile non negativo.
Dunque, la probabilità che una qualsiasi stima di un conteggio devi da quest'ultimo per più di $3\varepsilon||a||_1$ è minore di $\frac{1}{8}$. Per dimostrare questa affermazione, si ricorre, nuovamente, al bound di Markov. Infatti, avendo mostrato che $\mathbb{E}[|count[j, h_j(i)] - a_i|] \le \frac{\varepsilon}{e}||a||_1$ e applicando Markov si ottiene:

$$\Pr[|count[j, h_j(i)] - a_i| \ge 3\varepsilon||a||_1] \le \frac{\varepsilon||a||_1}{3e\varepsilon||a||_1} = \frac{1}{3e} < \frac{1}{8}$$

Per ridurre ulteriormente la stima della probabilità di errore, per l'item i, si tiene conto della mediana delle $\ln \frac{1}{\delta}$ stime indipendenti di a_i. Come garantito in [23], quindi, si

applicano i bound di Chernoff, e si ottiene, di conseguenza, che la probabilità con cui la mediana delle $\ln \frac{1}{\delta}$ stime sia sbagliata è minore di $\delta^{\frac{1}{4}}$. □

Anche in questo caso, il tempo necessario per produrre la stima si attesta a $O\left(\log \frac{1}{\delta}\right)$, mentre lo spazio utilizzato è di $\left(2 + \frac{e}{\varepsilon}\right) \ln \frac{1}{\delta}$ "celle".
In entrambi i casi, l'utilizzo di un Count-Min sketch permette una dipendenza inversamente proporzionale a ε, anziché una inversamente proporzionale a ε^2, che si otterrebbe utilizzando altri tipi di sketch (descritti nella sezione 2.4).

In conclusione, in questa prima sezione si è mostrato come rispondere ad una point query usando un Count-Min sketch, prima considerando solo il modello turnstile non negativo, vale a dire per conteggi positivi o nulli, e poi per il caso più generale. Sono state quindi illustrate le garanzie ammesse in entrambi i casi, con le corrispondenti dimostrazioni.

2.2 Inner Product Query

In questa sezione, si mostra come rispondere ad un inner product query con i rispettivi bound sul valore stimato, riprendendo i risultati ottenuti in [23]. Si tralasciano, però, le corrispondenti dimostrazioni poichè nell'implementazione di Count-Min sketch, come già accennato, si considera solo il caso non negativo di risposta a point query. Viene poi mostrata la complessità temporale e spaziale per questo tipo di operazione. Come ultima nota, si considera solamente il modello turnstile non negativo.

Ciò che una inner product query deve restituire è la seguente quantità $(a \hat{\odot} b)_j = \sum_{k=1}^{w} count_a[j,k] \cdot count_b[j,k]$. In altre parole, questa tipologia di query restituisce la somma degli elementi del vettore risultante dal prodotto componente per componente delle due righe j-esime delle due matrici che approssimano i conteggi dei due stream, a e b. É implicito il fatto che gli array bidimensionali per i due flussi devono presentare lo stesso numero di righe, e quindi lo stesso valore per il parametro δ.
Per definizione, quindi, la stima di $Q(a,b)$ per vettori non negativi è

$$a \hat{\odot} b = \min_j a \hat{\odot} b_j$$

Questo significa che, dopo aver calcolato d inner product (corrispondenti alle d righe delle due matrici), si considera come risposta approssimata alla query il valore minimo di essi.
In questo caso, [23] mostra il seguente teorema, che fornisce un limite inferiore del valore stimato:

$$a \hat{\odot} b \geq a \odot b$$

Inoltre, con probabilità almeno pari a $1 - \delta$ è verificato che

$$a \hat{\odot} b \leq a \odot b + \varepsilon \, ||a||_1 \, ||b||_1$$

ottenendo, in questo modo, anche un limite superiore. Riassumendo, con probabilità almeno pari a $1-\delta$, vale la seguente catena di disequazioni:

$$a \odot b \leq a \hat{\odot} b \leq a \odot b + \varepsilon \left|\left|a\right|\right|_1 \left|\left|b\right|\right|_1$$

In seguito, si osserva che la complessità temporale e spaziale di un'operazione di risposta ad una inner product query si attesta essere pari a $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\right)$. Tale procedura, come descritto precedentemente, consiste, infatti, nel moltiplicare componente per componente i $w = \lceil\frac{e}{\varepsilon}\rceil$ elementi di una riga, che vengono poi sommati tra loro. Questa operazione viene ripetuta per tutte le $d = \lceil\ln\frac{1}{\delta}\rceil$ righe. Infine, viene preso il valore minimo. É evidente, quindi che la complessità temporale si attesta ad essere al più pari a $\frac{e}{\varepsilon}\ln\frac{1}{\delta}$, o più semplicemente, ignorando ostanti moltiplicative e basi del logaritmo, $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\right)$. Lo spazio richiesto da Count-Min sketch è, invece, esattamente pari a $wd = \lceil\ln\frac{1}{\delta}\rceil\lceil\frac{e}{\varepsilon}\rceil$, quindi lo spazio richiesto si attesta, effettivamente, a $O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\right)$.
Per un aggiornamento delle matrici, invece, è richiesto un tempo pari a $O\left(\log\left(\frac{1}{\delta}\right)\right)$. Infatti, è sufficiente accedere alle $d = \lceil\ln\frac{1}{\delta}\rceil$ "celle" dello sketch, una per riga, per aggiornare il corrispondente contatore.

Inoltre, se il vettore b è non nullo solo in posizione i, cioè $b_i = 1$ e $b_j = 0\ \forall j \neq i$, allora la procedura illustrata in questa sezione coincide con quella seguita per stimare una point query. Infatti, se il secondo flusso è costituito da un solo elemento, allora il prodotto interno $a \hat{\odot} b$ risulta essere pari ad a_i. Ciò fornisce le stesse garanzie sull'errore viste per la prima tipologia di query studiata, vale a dire che con probabilità almeno pari a $1-\delta$ si ottiene che $a_i \leq \hat{a}_i \leq a_i + \ \varepsilon \left|\left|a\right|\right|_1$ (dal momento che $||b||_1 = 1$). Un risultato simile si ottiene nel caso generale, dove i vettori potrebbero avere entry negative.

La stima dell'inner product query con Count-Min sketch viene generalmente applicata nel *join size estimation*, utilizzato nei database query planner per determinare il miglior ordine con cui valutare le query ([23]). Più nel dettaglio, la *join size* di due relazioni di un database su un particolare attributo coincide con il numero di item nel prodotto cartesiano che presentano lo stesso valore per quell'attributo ([23]). In altre parole, questa quantità coincide con il numero di tuple risultanti a seguito di un join tra due relazioni su un particolare attributo. Nel caso specifico della query considerata, senza perdita di generalità, si può assumere che i valori degli attributi delle due relazioni su cui eseguire il join siano interi, appartenenti al range $\{1 \dots n\}$. Le due relazioni vengono, dunque, rappresentate come vettori, a e b. Questi ultimi sono costruiti in modo tale che l'elemento i-esimo rappresenti il numero di tuple che assumono valore i. Da questo si può dunque dedurre che $a \odot b$ coincide con il *join size* delle due relazioni. Se, come succede nei casi reali, le dimensioni dei due vettori sono elevate, è possibile salvare i conteggi in modo approssimato mediante l'uso degli sketch. In questo modo, è possibile effettuare delle stime sul conteggio dei vari item in presenza di inserimenti e cancellazioni. Come suggerito in [23], seguendo questa procedura, la stima del risultato di un *join size* di due relazioni

su uno specifico attributo si discosta al più di $\varepsilon||a||_1||b||_1$ con probabilità almeno pari a $1-\delta$. Per applicare questa procedura si mantiene uno spazio pari a $O(\frac{1}{\varepsilon}\ln\frac{1}{\delta})$. Ciò deriva dai bound illustrati all'inizio di questa sezione.

In conclusione, in questa parte è stata presentata la procedura seguita per rispondere ad una inner product query quando viene utilizzato Count-Min sketch per salvare i conteggi dei vari item. In seguito, è stato mostrato il limite inferiore e superiore entro cui è contenuta la stima della risposta alla query. Infine, si è illustrato come le inner product query, con lo sketch studiato, possano essere utilizzati per implementare l'operazione di *join size* nell'ambito dei database.

2.3 Range Query

In questa sezione viene analizzata l'ultima tipologia di query considerata, vale a dire la range query. Seguendo la spiegazione in [23], si fornisce, innanzitutto, un primo metodo di risposta, che consiste nel ricondurre questa tipologia di query ad una inner product query. Si mostra, tuttavia, che questa procedura comporta degli svantaggi, in termini di efficienza ed errore. Di conseguenza, si introduce il concetto di *dyadic range*, che può essere utilizzato per rispondere più efficacemente a tale query. Vengono, quindi, mostrati dei bound sulla stima della risposta.

Per definizione, dato il vettore a, la risposta alla range query di estremi l e r, denotata come $Q(l,r)$, risulta essere pari a $\sum_{i=l}^{r} a_i$.
Si definisce, quindi, $\chi(l,r)$, cioè un vettore indicatore di dimensione n tale per cui la componente i-esima presenta valore unitario solo se i appartiene all'intervallo $[l,r]$. Più formalmente, riprendendo la notazione usata in [23], $\chi(l,r)_i = 1 \leftrightarrow l \leq i \leq r$, altrimenti $\chi(l,r)_i = 0$.
Da questo si osserva che

$$Q(l,r) = \sum_{i=l}^{r} a_i = \sum_{i=1}^{n} a_i \cdot \chi(l,r)$$

Quindi ne consegue che un range query può essere interpretata come inner product query $Q\left(a,\ \chi(l,r)\right)$.
Tuttavia, come spiegato in [23], questo metodo ha due difetti:

- La risposta stimata presenta come limite superiore $(a \odot \chi(l,r)) + \varepsilon\,||a||_1\,||\chi(l,r)||_1$. Ciò significa che somme di range elevate presentano un errore che cresce linearmente con la lunghezza del range. In altre parole, all'aumentare delle dimensioni del range aumenta di conseguenza anche l'eventuale discostamento dal valore effettivo, incrementando il valore massimo dell'errore ammesso. Questa soluzione non presenta, dunque, garanzie di accuratezza "strette".

- Il costo temporale per costruire lo sketch per $\chi(l,r)$ dipende linearmente dalla lunghezza del range, pari a $r-l+1$. Si può, infatti, notare che calcolare con la modalità

presentata somme degli elementi nel range $[l, r]$ coincide con il computare delle point query per ogni item nel range e poi sommare le stime. Questo è particolarmente critico quando i range comprendono molti elementi.

Un modo per evitare il problema dell'elevata complessità temporale consiste nell'usare dei *dyadic range*. Essi si configurano come intervalli che assumono la seguente forma:

$$[x2^y + 1, \dots (x+1)2^y]$$

di parametri interi x e y ([23], [20]).
In un dyadic range, dunque, sono contenuti tutti gli interi compresi tra $x2^y + 1$ e $(x+1)2^y$, per un totale di 2^y valori, quindi una potenza di due. In particolar modo, y può essere riscritto come $y = \log_2(n) - j$, con j definito "livello di risoluzione" ([20]). In tutto, sono presenti $\log_2 n + 1$ livelli di risoluzione.
Come illustrato in [23], la procedura di stima per una range query consiste, innanzitutto, nel mantenere $\log_2 n$ Count-Min sketch (uno per ogni livello di risoluzione), in modo da rispondere in modo approssimato alle stesse range query. A questo punto, qualsiasi range query può essere ridotta ad al massimo $2\log_2 n$ dyadic range query, che a loro volta possono essere ridotte a ad una singola point query ([23]). Ciascun punto nel range $[1 \dots n]$ è una componente delle $\log_2 n$ dyadic range, una per ogni y nel range $[0 \dots \log_2(n) - 1]$. Uno sketch è mantenuto per ogni set di dyadic range di lunghezza 2^y. Quindi, data $Q(l, r)$ si calcolano al massimo $2\log_2 n$ dyadic range che coprono il range considerato, restituendo la stima della somma delle query, come voluto.
Dunque, posto $a[l, r] = \sum_{i=l}^{r} a_i$ pari alla risposta alla range query $Q(l, r)$ e $\hat{a}[l, r]$ la sua stima, vale il seguente teorema:

$$a[l, r] \leq \hat{a}[l, r]$$

E con probabilità almeno pari a $1 - \delta$, vale che

$$a[l, r] \leq \hat{a}[l, r] + 2\varepsilon logn \, ||a||_1$$

Il tempo necessario per calcolare la stima o per effettuare un aggiornamento è $O\left(\log(n) \log \frac{1}{\delta}\right)$, in quanto, per rispondere ad una range query è necessario reperire $d = \Theta\left(\ln \frac{1}{\delta}\right)$ stime di un particolare item per ognuno dei $\log_2 n$ Count-Min sketch utilizzati. Da ciò ne consegue che la complessità temporale sia limitata superiormente da $O\left(\log(n) \log \frac{1}{\delta}\right)$. Lo spazio usato si attesta invece a $O\left(\frac{\log(n)}{\varepsilon} \log \frac{1}{\delta}\right)$, poiché, come spiegato, è necessario mantenere $\log_2 n$ sketch di dimensioni pari a $\lceil \frac{e}{\varepsilon} \rceil \lceil \ln \frac{1}{\delta} \rceil$. Questi si configurano come bound al caso peggiore.

In conclusione, in questa sezione è stato presentato il concetto di range query. Più nel dettaglio, dopo una descrizione iniziale di questa tipologia di query sono state introdotte delle tecniche che permettono di produrre una risposta. Come prima, ma poco efficiente,

strategia si è convertita la query in una inner product query. Successivamente, per migliorare la complessità temporale e i bound di errore della risposta è stato introdotto il concetto di range diadico. Di questa seconda soluzione, si sono mostrati, quindi, i limiti sulla stima della risposta e il valore asintotico della complessità temporale e spaziale.

2.4 Confronto tra tipologie di sketch

In questa sezione, si descrivono brevemente altre strutture di sketch, quali tug-of-war sketch, random subset sums e Count sketch. Similmente a Count-Min sketch, essi si compongono di un array bidimensionale a cui sono associate delle funzioni di hash. Per ognuna di esse, poi, si confrontano l'accuratezza nella stima ed efficienza con quelle di Count-Min sketch. L'obiettivo è dimostrare che quest'ultimo sketch sia più vantaggioso.

- *tug-of-war sketch* o *AMS sketch* Questo sketch è stato introdotto in [2] allo scopo di stimare i momenti di frequenza di un data stream (come il secondo momento, che si configura come somma dei quadrati delle frequenze). Più in generale, esso viene utilizzato nella risposta ad inner product query. Secondo la descrizione fornita in [23], esso è costituito da un vettore di dimensione $O\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\right)$. Ad ogni entry dell'array è quindi associata una funzione di hash g_j, che soddisfa la proprietà di four-wise independence e che mappa ciascun item o in +1 o in -1. Il valore salvato in una generica entry, per esempio la j-esima, risulta quindi essere pari a $\sum_{i=1}^{n} a_i \cdot g_j(i)$. Essendo lo sketch costituito da $O\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\right)$ entry, si evince che la complessità spaziale presenta lo stesso upper bound. Come suggerito in [23] e [2], inoltre la complessità temporale per operazioni di aggiornamento e di risposta a (inner product) query si attesta anch'essa a $O\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\right)$. Sia nel caso di un aggiornamento che in quello di una query è necessario, infatti, effettuare esattamente un accesso in ogni componente del vettore. L'errore nella stima della risposta risulta, invece, essere uguale a $\pm\varepsilon||a||_2||b||_2$ ([23]).

  Tramite un'opportuna modifica dello sketch, che consiste o nell'aggiungere delle variabili per la somma di range o un secondo insieme di funzioni di hash (che realizza Count Sketch), è possibile rispondere rispettivamente a range e point query. Tuttavia, le complessità temporale e spaziale crescono entrambe di un fattore $\log n$. L'errore nella stima per la prima tipologia si attesta a $\varepsilon(r-l+1)||a||_1$, mentre per la seconda a $\pm||a||_2$ ([23]).

- *Fast-Count sketch* Come descritto in [37], questo sketch si configura come un miglioramento di tug-of-war sketch. Esso è infatti costituito da un vettore delle stesse dimensioni di AMS sketch. Tuttavia, ad esso è associata un'unica funzione di hash (che soddisfa la four-wise independence). Questo significa che il conteggio di ciascun item viene mappato una sola volta. Ciò comporta un tempo di aggiornamento costante. Questo sketch viene utilizzato principalmente nell'ambito delle inner product

query. Sempre in [37], viene mostrato quindi che l'errore di accuratezza è limitato dal termine $\pm||a||_2||b||_2$.

- *Random subset sums* Gli autori in [20] definiscono un random subset sum nel seguente modo: si suddivide un insieme universo S in un set di sottoinsiemi, $\{S_1, \ldots S_l \ldots S_m\}$, contenenti valori (o di intervalli diadici) casuali distinti che presentano probabilità $\frac{1}{2}$ di appartenere ad un particolare sottoinsieme. Un random subset sum si configura come il numero complessivo di elementi che appartengono a S_i. Esso può dunque essere considerato come un contatore, che viene aggiornato ogni qualvolta si verifichi un inserimento o una rimozione di uno dei componenti del sottoinsieme a cui è associato. Nel caso più generale, si considera un sottoinsieme di intervalli diadici a livello j. L'insieme S è costituito, quindi, dall'unione di questi intervalli. Si ripete questa operazione $24 \log(\log n/\delta) \log(n)/\varepsilon^2$ volte, per ogni livello. Il random subset sum, per il sottoinsieme S_l è, perciò, pari alla somma dei conteggi dei vari elementi appartenenti ad esso. Per salvare questi dati, come presentato, poi, in [23], si utilizza uno sketch che si compone di una matrice con due colonne e $d = \frac{24}{\varepsilon^2} \log\left(\frac{1}{\delta}\right)$ righe. Ad ogni riga è associata una funzione di hash, che mappa l'elemento o in 0 o in 1. Inoltre ognuna di queste, soddisfa l'indipendenza a coppie. Questa tecnica viene applicata essenzialmente per calcolare range query ([23]); questa strategia presenta complessità spaziale e temporale (sia per operazioni di aggiornamento che di risposta di query) a $O(\log(n)^2/\varepsilon^2 \ln(1/\delta))$.

- *Count sketch* Questo sketch è stato introdotto e descritto in [6] quale miglioramento di *tug-of-war sketch*, per permettere di rispondere alle point query. Esso consiste nell'utilizzare un array bidimensionale di contatori, C (di dimensioni $d \cdot w$, con $w = O\left(\frac{1}{\varepsilon^2}\right)$ e $d = O(\log(\frac{1}{\delta})))$, e due funzioni di hash per ogni riga (entrambe con pairwise independence). La prima di queste, h_j mappa ciascun item in una posizione appartenente all'intervallo $[0, w-1]$, mentre la seconda funzione, g_j mappa l'elemento o in -1 o in +1 (similmente a quanto visto per *tug-of-war sketch*). Ogni item nel flusso comporta la somma o la sottrazione (a seconda del valore assunto da g) del valore $g_j(i) \cdot c_i$ alla entry in posizione $[j, h_j(i)]$, per ogni riga. La risposta stimata ad un point query per l'item i si attesta essere, dunque, pari a $\hat{c}_i =$ mediana$_j g_j(i) \cdot C[j, h_j(i)]$. Come viene fatto notare in [6], per calcolare la stima del conteggio si utilizza l'operatore di mediana. Ciò è dovuto al fatto che in questa struttura dati possono verificarsi collisioni tra item heavy hitter ed elementi meno frequenti. Questo comporta, quindi, la presenza di stime "outlier", che si discostano, cioè, significativamente dal conteggio effettivo. Il motivo per cui, dunque, viene scelta la mediana consiste nella sua maggior robustezza a questi conteggi inesatti rispetto ad altri operatori, come la media.

  Infine, per come è stato definito lo sketch, si evince che lo spazio richiesto in questo caso si attesta essere pari $\Theta(1/\varepsilon^2 \log(1/\delta))$. Sia nel caso di aggiornamento e di

risposta ad una query, è necessario effettuare $d = \Theta(\log 1/\delta)$ accessi. La complessità temporale per eseguire queste due operazioni si attesta, perciò a $\Theta(\log 1/\delta)$.

Come mostrato dagli autori dell'articolo [23] e [13], il Count-Min sketch presenta, rispetto alle altre strutture di sketch illustrate, i seguenti vantaggi:

- La complessità spaziale è proporzionale a $\frac{1}{\varepsilon}$, ε volte più piccolo dello spazio richiesto da Count Sketch e di tug-of-war sketch.

- Per quanto riguarda la complessità temporale, le operazioni di aggiornamento e di risposta alle point query si attestano essere pari a $\Theta\left(\log \frac{1}{\delta}\right)$, similmente a quanto ottenuto utilizzando Count sketch. Tuttavia, a differenza di quest'ultima struttura, Count-Min sketch può rispondere anche ad inner product e range query. Si confronta, quindi, la complessità temporale di Count-Min sketch e di AMS sketch per operazioni di risposta a range query e inner product query. In entrambe i casi, nella prima struttura dati il tempo migliora di un fattore ε rispetto al tug-of-war sketch. In Count-Min sketch, infatti, il tempo per rispondere ad una range query si attesta a $O(\log(1/\delta)\log(n)/\varepsilon)$, mentre quello per una inner product query a $O(\log 1/\delta 1/\varepsilon)$. Count-Min sketch presenta, inoltre, un tempo di aggiornamento sublineare nelle dimensioni dello sketch([23]). Rispetto a tug-of-war sketch e random subset sums, la complessità per effettuare questo task è migliore di un fattore $\varepsilon^2/\log n$.

- Come accennato nel punto precedente, Count-Min sketch permette di rispondere a tutte le tipologie di query studiate, a differenza di random subset sums, che si applica solo a range query e tug-of-war sketch, che non permette di rispondere a point query, a meno che non venga aggiunto un secondo insieme di funzioni di hash (ottenendo così Count sketch).

- La struttura dati richiede solo funzioni di hash a coppie indipendenti (pairwise independence) che sono semplici da costruire. In tug-of-war sketch, invece, le funzioni di hash sono tali da soddisfare la four-wise independence.

In conclusione, in questa sezione sono state presentate strutture di sketch precedenti a Count-Min sketch, quali tug-of-war sketch, random subset sums e Count sketch. Per ciascuna di esse sono state analizzate le prestazioni in termini di errore nella stima e la complessità temporale e spaziale. Sotto questi aspetti, è dunque emerso come Count-Min sketch sia più efficace ed efficiente. Sono state quindi evidenziate le principali caratteristiche e vantaggi che contraddistinguono questo sketch dagli altri.

Capitolo 3

Applicazioni di Count-Min Sketch

In questo capitolo vengono presentate le principali applicazioni (alcune più teoriche, altre più pratiche) di Count-Min sketch. In particolar modo, il capitolo introduce come l'utilizzo di questo sketch porti ad un miglioramento dei bound temporali e spaziali per i due problemi dei quantili e degli heavy hitter (descritti sinteticamente nel capitolo 1, pagina 3). Per questi, si riporta quindi la descrizione presentata in [23]. Successivamente, si mostrano altre applicazioni di Count-Min sketch, volte ad ottimizzare le prestazioni per task in database, biologia computazionale, sicurezza e controllo di flussi di pacchetti IP.

3.1 Quantili nel modello turnstile

In questa sezione viene mostrato come è possibile determinare, dato un flusso in cui possono verificarsi sia inserimenti che cancellazioni di item, i $\phi - Quantili$ utilizzando Count-Min sketch. Vengono poi presentate brevemente alcune applicazioni dei quantili a database e data mining.

Come dimostrato dagli autori dell'articolo [20] e come ripreso in [23], il problema di determinare i $\phi - Quantili$ approssimati (calcolare esattamente i quantili comporta la scansione dell'intera relazione, che si configura come procedura computazionalmente intensiva), in un flusso turnstile, può essere ricondotto ad un problema di calcolo di somme di range. Più nel dettaglio, il procedimento consiste nell'effettuare delle ricerche binarie per determinare i range (diadici) $\{1 \dots r\}$ tali per cui la somma de conteggi degli elementi $a[1, r]$ è pari a $k\phi \, ||a||_1 \;\; \forall k : \;\; 1 \leq k \leq \frac{1}{\phi} - 1$. In questo modo, si estrapolano i $\phi - Quantili \; \varepsilon - approssimati$, cioè gli elementi che assumono valore appartenente all'intervallo $[(k\phi - \varepsilon)N, (k\phi + \varepsilon)N]$, in cui ε è il massimo errore ammesso.
Per calcolare queste somme di range, gli autori di [20] propongono l'utilizzo di random subset sums, struttura dati presentata sinteticamente a pagina 20.

Ciò che suggeriscono, invece, gli autori di [23], consiste nel sostituire le range sums con $\log n$ Count-Min sketch, uno per livello dei dyadic range e nell'impostare il parametro di accuratezza a $\varepsilon / \log n$, in modo tale che l'errore massimo tollerabile sia distribuito in modo

uniforme tra gli sketch. Inoltre, la probabilità di fallimento si attesta a $\phi\delta/\log n$. Con tale strategia, è possibile ottenere le ε-approssimazioni dei $\phi - Quantili$ con probabilità almeno pari a $1-\delta$ utilizzando uno spazio di $\left[\lceil \frac{e}{\varepsilon}\log n\rceil \lceil \ln\left(\frac{\log n}{\phi\delta}\right)\rceil\right] \cdot \log n \in O\left(\frac{1}{\varepsilon}(\log n)^2 \log\left(\frac{\log n}{\phi\delta}\right)\right)$ (almeno $\frac{34}{\varepsilon}$ volte migliore della prima soluzione). Migliora, inoltre, la complessità temporale per le operazioni di aggiornamento e query, la cui complessità temporale si attesta a $O\left(\log n \log\left(\frac{\log n}{\phi\delta}\right)\right)$. Scegliere come struttura dati Count-Min sketch rispetto a random subset sums riduce, dunque, il tempo risposta ad una query e di aggiornamento di un fattore almeno pari a $\frac{34}{\varepsilon^2}\log n$.
Come suggerito in [20], infine, i quantili presentano diverse applicazioni:

- I quantili vengono quindi applicati nei database, dove permettono di riassumere proprietà di insiemi di dimensioni elevate di relazioni. In questo caso, molti DBMS utilizzano, nel processo di ottimizzazione delle query, i cosiddetti *equi-depth histograms*, che sono quantili, che permettono di stimare le dimensioni dei risultati intermedi, in modo tale da scegliere l'ordine di esecuzione;

- I quantili possono, inoltre, essere sfruttati per evincere regole di associazione in applicazioni di data mining. Più nel dettaglio, [40] e [1] definiscono una regola di associazione come un'implicazione dalla forma $X \to y_i$, con X insieme di item (o transazione) e Y elemento appartenente alla transazione Y e non presente in X. La regola è soddisfatta con un fattore di confidenza $0 \leq c \leq 1$ se e solo se almeno il c delle transazioni che soddisfano X, soddisfano anche y_i Un esempio più pratico consiste nella seguente affermazione: "Il 90 % delle transazioni contenenti x_1 e x_2, contiene anche y_1".

In conclusione, Count-Min sketch si configura come una soluzione più efficiente di range subset sums per determinare i $\phi - Quantili$, che si applicano a diversi ambiti, tra cui database e data mining.

3.2 Heavy Hitter

Questa sezione studia l'applicazione di Count-Min sketch alla determinazione degli elementi heavy hitter. Secondo la definizione fornita in [16], dato uno stream S di taglia $|||a||_1 = N$ e una soglia ϕ, un elemento è classificato come heavy hitter se il il numero di volte che compare in S supera il valore $\lfloor \phi N \rfloor$. Si studia prima il caso cash register, per poi trattare anche del caso più generale, in cui è possibile rimuovere elementi (e quindi decrementare il conteggio, senza che, però, possa diventare negativo). Quanto descritto fa riferimento principalmente a [23].

3.2.1 Data stream cash register

Ricordando che $||a||_1 = \sum_{i=1}^{t} c_i$, si considera l'operazione di inserimento della entry (i_t, c_t) all'istante di tempo t.

A riguardo, in [23] viene dimostrato il seguente teorema:

Gli heavy hitter possono essere evinti da sequenze cash register, costituite da $L_1 = ||a||_1$ elementi, usando Count-Min sketch. Questa strategia porta ad occupare uno spazio $O\left(\frac{1}{\varepsilon}\log\frac{||a||_1}{\delta}\right)$ e presenta una complessità temporale pari a $O\left(\log\frac{||a||_1}{\delta}\right)$ per item. Un heavy hitter si configura come un item che si presenta con un conteggio superiore a $\phi\,||a||_1$. Con probabilità $1-\delta$ nessun item con conteggio minore a $(\phi-\varepsilon)\,||a||_1$ viene classificato come tale.

In pratica, per determinare questi item, si analizza il flusso, considerando un elemento alla volta. Ad ogni aggiornamento, ad esempio (i_t, c_t), si modifica il valore del corrispondente elemento nello sketch e, contemporaneamente, si pone la point query $Q(i_t)$. Se si riscontra che la stima del conteggio per i è superiore a $\phi||a||_1$, allora questo item viene aggiunto in un heap (nell'implementazione effettuata si utilizza alternativamente una mappa, per maggiore semplicità). In questa struttura dati, quindi, vengono mantenuti tutti gli elementi che superano la threshold fissata. Un elemento ivi contenuto viene rimosso dall'heap qualora il corrispondente conteggio sia inferiore alla soglia considerata. Alla fine dell'operazione, l'heap contiene gli elementi heavy hitter. In questo modo, è possibile monitorare in modo efficiente solo gli elementi più significativi, mantenendo sotto controllo la memoria utilizzata e assicurando che solo gli elementi rilevanti vengano considerati nel risultato finale.
Si prosegue con la dimostrazione del teorema, illustrata in [23].

Dimostrazione. La procedura descritta si basa sul fatto che la soglia, essendo definita come $\phi||a||_1$, incrementa in modo monotono, man mano che pervengono nuovi item nel flusso. Quindi, ad ogni aggiornamento, la soglia viene incrementata e, conseguentemente, si verifica che il conteggio di ciascun elemento contenuto nell'heap superi ancora il valore di threshold. Questo assicura che nessun heavy hitter sia omesso dall'output. Tuttavia, vi è la possibilità che elementi non-heavy hitter vengano erroneamente inclusi (falsi positivi). Per ridurre al minimo questi errori, il parametro δ deve essere scalato in modo da garantire che la probabilità di includere un elemento non-heavy hitter sia limitata a $1-\delta$, per tutte le query complessivamente poste. Si suppone che la probabilità di commettere un errore all'istante t sia pari a δ', evento indicato con E_k. Si ricorre, quindi, alla disuguaglianza di Boole ([3]), la quale permette di affermare che, dati gli eventi $E_1, \ldots E_m$,

$$\Pr[\bigcup_{i=1}^{m} E_i] \leq \sum_{i=1}^{m} \Pr[E_i] = \delta' ||a||_1 \leq \delta$$

. Il valore a cui fissare δ' deve essere al più $\frac{\delta}{||a||_1}$.

□

3.2.2 Data stream turnstile

Viene ora considerato il problema di determinare gli elementi heavy hitter in uno stream turnstile, in cui, cioè, sono possibili sia inserimenti che rimozioni di item.
Come descritto in [14], per determinare gli heavy hitter si adotta una strategia divide and conquer, applicata ad intervalli diadici e a cui si aggiunge un oracolo, che permette di stabilire se nell'intervallo il conteggio complessivo degli elementi contenuti supera una valore di soglia fissato. Si tengono in considerazione $\log n$ sketch indipendenti, uno per ogni livello (o lunghezza) di range diadico, per calcolare somme di range. Quando all'istante di tempo t si verifica l'aggiornamento $(i_t,\ c_t)$, ciascuno degli sketch coincidenti con un range diadico a cui l'item appartiene, viene aggiornato. A questo punto, per identificare gli heavy hitter si effettua una ricerca binaria parallela, in cui in ogni invocazione si esaminano ricorsivamente intervalli diadici di lunghezza decrescente, fino ad arrivare al caso baso in cui si esaminano singoli elementi. Più precisamente, l'algoritmo suddivide a metà, in due subarray (corrispondenti a loro volta a due dyadic range), il range diadico di lunghezza n. Questi intervalli vengono bipartiti ricorsivamente solo se il conteggio complessivo degli item in essi contenuti supera la soglia $(\phi + \varepsilon)\,||a||_1$. I casi base risultanti da queste invocazioni, costituiti da un unico elemento e tali per cui il conteggio supera la threshold fissata, coincidono con il risultato cercato.
Per limitare, invece, il numero di falsi positivi in output, si ricorre nuovamente al bound di Boole, in modo tale da impostare la probabilità di fallimento per ogni sketch a $\delta' \leq \frac{\delta\phi}{2\log n}$. Infatti, ad ogni livello ci sono al massimo $\frac{1}{\phi}$ item con frequenza maggiore di ϕ. Inoltre, per come è stato costruito, l'algoritmo esplora entrambe i rami di un nodo della gerarchia, quindi, in tutto, pone al più $\frac{2}{\phi}$ query su livello. Essendoci in tutto $\log n$ livelli, e applicando il bound di unione, si deduce che $\delta' \frac{2\log n}{\phi} \leq \delta$. Da ciò ne consegue che $\delta' \leq \frac{\delta\phi}{2\log n}$.
Siccome Count-Min sketch non sottostima i conteggi, nell'output vengono inclusi tutti gli elementi heavy hitter. La probabilità di ammettere falsi positivi è limitata da δ.
Gli autori di [23], quindi dimostrano il conseguente teorema:

L'algoritmo per determinare gli heavy hitter in un data stream di tipo turnstile, occupa uno spazio $O\left(\frac{1}{\varepsilon}\log n \log\left(\frac{2\log n}{\delta\phi}\right)\right)$ e presenta una complessità temporale pari a $O\left(\log n \log\left(\frac{2\log n}{\delta\phi}\right)\right)$ per ogni aggiornamento. Ciascun item con conteggio almeno pari a $(\phi + \varepsilon)\,||a||_1$ è posto nel risultato finale e con probabilità $1 - \delta$ nessun falso positivo è inserito nell'output.

Si è mostrato, infatti, che vengono complessivamente mantenuti $\log n$ sketch di dimensioni $\lceil\frac{e}{\varepsilon}\rceil \lceil\ln\left(\ln \frac{2\log n}{\delta\phi}\right)\rceil \in O\left(\frac{1}{\varepsilon}\log n \log\left(\frac{2\log n}{\delta\phi}\right)\right)$. La complessità dell'operazione di aggiornamento discende, a sua volta, dalla struttura degli sketch: il conteggio di un elemento è presente in al più tutti i $\log n$ sketch. In ciascuno di essi, il conteggio è salvato $\lceil\ln \frac{2\log n}{\delta\phi}\rceil$. Quindi si effettuano al più $O\left(\log n \log\left(\frac{2\log n}{\delta\phi}\right)\right)$ accessi.
Adottando questa strategia, è possibile determinare solo gli heavy hitter ε-approssimati, similmente a quanto visto nel caso dei quantili nella sezione precedente. Infatti, con uno

streaming algorithm, non è possibile determinare con esattezza gli heavy hitter in uno spazio sublineare ([5]). Come definito in [39], si considera un flusso di n elementi distinti e due parametri k e ε. Gli item il cui conteggio supera la soglia $\frac{n}{k} - \varepsilon n$ sono classificati come heavy hitter.
In entrambe i casi, infine, non è possibile determinare gli heavy hitter leggendo una sola volta il flusso con uno spazio di memoria limitato in real time ([39]).

3.2.3 Heavy Hitter Gerarchici

Secondo la definizione fornita in [16], un heavy hitter gerarchico si configura come un elemento appartenente una struttura gerarchica il cui numero complessivo di discendenti è superiore alla soglia $\lfloor \phi N \rfloor$. Inoltre, i nodi discendenti non possono essere a loro volta heavy hitter gerarchici, quindi il loro contributo viene eventualmente ignorato. La determinazione di questi elementi si rivela efficace in numerosi ambiti, come il "Network-Aware Clustering", che mira a identificare dei gruppi (come gli host sotto lo stesso dominio amministrativo) sulla base di pattern di accesso, che sono, per esempio le richieste poste ad un server. Un'altra applicazione riguarda il monitoraggio di possibili attacchi DoS, che possono essere identificati da un'elevata superiorità del numero di pacchetti "SYN" rispetto alla quantità di pacchetti "ACK" che vengono recepiti dall'host sotto attacco ([16]).
Sinteticamente, l'algoritmo di determinazione degli heavy hitter in una gerarchia di profondità h si basa su una ricerca top-down, in cui, a partire dalla radice si calcola il peso complessivo w di un nodo. Questo peso equivale alla somma dei conteggi dei nodi foglia discendenti. Se questo valore è inferiore della soglia $\lfloor \phi N \rfloor$, allora si ritorna 0 e nessun elemento è heavy hitter. Altrimenti si investigano ricorsivamente in nodi figli e si determina il loro conteggio, W. Se $w - W \geq \lfloor \phi N \rfloor$ allora il nodo di partenza è un heavy hitter gerarchico, altrimenti si ritorna W. Per memorizzare i conteggi intermedi, [16] suggerisce l'uso di random subset sums. In questo modo, la complessità spaziale e la complessità temporale per aggiornare gli sketch si attestano a $O\left(\frac{h}{\varepsilon^2} \log \frac{1}{\delta}\right)$. Se al posto di utilizzare random subset sums si sfrutta Count-Min sketch, come suggerito in [23], presenta così la complessità spaziale si riduce a $O\left(\frac{h}{\varepsilon} \log \frac{1}{\delta}\right)$ e il tempo di aggiornamento migliora a $O\left(h \log \frac{1}{\delta}\right)$.

In conclusione, in questa sezione sono stati presentati i concetti di $\phi - Quantili$ e di *Heavy Hitter*. É stato, quindi, mostrato che sfruttando Count-Min sketch la risoluzione a questi problemi risulta più efficiente da un punto di vista spaziale e temporale, rispetto a soluzioni proposte antecedentemente.

3.3 Applicazioni di Count-Min sketch a task reali

Oltre a questi problemi, più astratti, la stima dei conteggi degli elementi in un data stream trova la sua applicazione in diversi task, come in problemi relativi al machine learning, quali

feature selection, ranking, semi-supervised learning, natural language processing ma anche a misure delle reti (controllo indirizzi IP, che permettono di dedurre informazioni statistiche significative sulle operazioni svolte in rete), di sicurezza e riguardanti i database. Lo scopo di questa sezione è, dunque, quello di mostrare i principali ambiti in cui Count-Min sketch può essere applicato ai fini di migliorare l'accuratezza e i bound di complessità.

3.3.1 Natural Language Processing

Il Natural Language Processing (NLP) è un insieme di tecniche computazionali, sviluppate su basi teoriche, per l'analisi e la rappresentazione di testi naturali a diversi livelli di analisi linguistica, con lo scopo di realizzare un'elaborazione del linguaggio simile a quella umana, per una varietà di compiti o applicazioni ([30]). Come mostrato in [22], Count-Min sketch può essere applicato ai task di predizione dell'orientazione semantica delle parole, approcci distributivi per identificare la somiglianza di parole e, infine, parsing non supervisionato di dipendenza con una ristretta conoscenza linguistica. Più nel dettaglio, queste operazioni richiedono di calcolare delle coppie di associazione tra parole. É necessario, quindi, contare il numero di volte in cui la stessa coppia di parole appare entro una certa finestra di n word. Per ridurre lo spazio utilizzato, gli autori in [22] propongono di salvare questi conteggi in Count-Min sketch.

3.3.2 Controllo di flussi di reti

Una prima soluzione per l'analisi del traffico di una rete consiste nel contare pacchetti periodicamente campionati (NetFlow). Tuttavia, questa strategia è lenta ed inaccurata. Le soluzioni proposte dagli autori dell'articolo [17] si concretizzano negli algoritmi *sample and hold* e *multistage filters*, le cui performance sono simili tra loro, anche se la prima è più semplice, mentre la seconda presenta garanzie di accuratezza migliori.
Brevemente, nel primo caso, il metodo inizia con la selezione di un pacchetto casuale (sample) da uno specifico flusso di traffico di rete. Se non è ancora presente alcuna entry per questo stream, in memoria SRAM, viene creata una nuova entry. Quindi, dopo aver selezionato il pacchetto campione, il sistema "trattiene" (hold) questo flusso di traffico per un certo intervallo di tempo. Ciò significa che ogni pacchetto successivo appartenente allo stesso stream (identificato da parametri come indirizzo IP di origine e destinazione, numeri di porta, ecc.) viene registrato. In altri termini, durante questo intervallo di tempo, in una tabella di hash vengono mantenuti i conteggi dei pacchetti. A termine di questo periodo, il sistema seleziona un nuovo pacchetto campione e ripete il processo. Questo garantisce che vari flussi di traffico vengano campionati e analizzati nel tempo.
Nel caso dell'algoritmo *multistage filters*, invece, la procedura si compone di più stadi di hash (come suggerito dal nome), ognuno dei quali contiene una serie di contatori e che operano in parallelo. Ogni stadio è composto da un insieme di contatori, indicizzati da una funzione di hash che agisce sull'ID del pacchetto (che rappresenta il flusso a cui il

pacchetto stesso appartiene). Tutti i pacchetti appartenenti ad uno stesso stream vengono dunque mappati nello stesso bucket. Quindi, quando un pacchetto viene analizzato, l'algoritmo calcola il valore di hash corrispondente ed incrementa della dimensione del pacchetto il corrispettivo contatore. Questo processo è ripetuto simultaneamente per tutti gli stadi, ognuno dei quali ha associata una funzione di hash indipendente dalle altre. Successivamente, ogni stadio agisce come un filtro che lascia passare solo i flussi i cui contatori superano una certa soglia. Al termine del processo, gli stream che hanno superato tutti gli stadi (più semplicemente basta che il contatore con il valore minimo per un certo flusso sia superiore alla threshold) sono considerati flussi di interesse, paragonabili a heavy hitter. Questa tecnica permette di ridurre i falsi positivi.
Quest'ultima soluzione presenta una struttura molto simile a Count-Min sketch. Tuttavia, i multistage filters non sono sketch: essi non approssimano alcun valore, ma ritornano una risposta binaria che informa se un determinato stream ha superato una certa soglia numerica ([23]). Inoltre, queste strutture ammettono solo aggiornamenti positivi. Si potrebbe, quindi, utilizzare Count-Min sketch per trattare dello stesso problema con flussi di tipo turnstile. In questo modo non è necessario imporre di avere esclusivamente aggiornamenti positivi, anche se è necessaria l'indipendenza tra una qualsiasi coppia di item mappati da funzioni di hash distinte.

3.3.3 Database

Tramite una query, è possibile ricavare informazioni utili dalle tabelle del database ([39]). Data una relazione definita `table` che presenta tra gli attributi `feature`, un esempio di query (in cui viene applicata una funzione di aggregazione, `count`) è il seguente: `SELECT count(feature) FROM table WHERE feature = f`.
In questo caso, il DBMS cerca e seleziona, tra tutte le righe di `table`, quelle che soddisfano la condizione `feature = f`. Viene, dunque, calcolato il numero complessivo di tuple risultanti. Se le dimensioni della tabella sono elevate, questa operazione si configura computazionalmente dispendiosa, specie se la query è posta frequentemente. Per risolvere questo problema, si può, di conseguenza, utilizzare Count-Min sketch, in modo tale da approssimare la frequenza di ciascuna combinazione possibile di item della tabella.

Count-Min sketch viene, inoltre, applicato nelle cosiddette *iceberg queries*. Secondo la definizione in [18], esse consistono in query che calcolano funzioni aggregate (spesso `count`, ma sono ammesse anche `max`, `average`, ...) su uno o più attributi, per poi eliminare i valori risultanti sotto una certa soglia, tipicamente fissata dall'utente. Il nome deriva proprio dal fatto che la quantità di elementi selezionata è relativamente molto piccola rispetto alla mole di dati in input. Queste query sono, quindi, simili al problema di identificazione degli heavy hitter in un data stream di tipo cash register. Il procedimento di risoluzione di una iceberg query prevede, dunque, di salvare in un heap i valori degli attributi superiori ad una certa soglia e trascurare i restanti.

Infine, l'utilizzo di Count-Min sketch si rivela utile nel join distribuito. Esso consiste nel join di due tabelle diverse appartenenti a database risiedenti in server distinti ([39]). Una prima soluzione (definita Bloom join) volta ad ottimizzare questa operazione consiste nell'utilizzo di un Filtro di Bloom. Esso si configura come una struttura dati probabilistica costituita da un array di m possibili valori binari (inizialmente posti a 0) e k funzioni di hash. Considerando un flusso, ogni elemento viene mappato da ciascuna hash function nel range $[1, \ldots, m]$. I corrispondenti k bit nel vettore vengono quindi impostati a 1. A questo punto, per verificare se un item appartiene allo stream è sufficiente verificare che i corrispondenti bit siano tutti inizializzati a 1 ([4]). Dunque, ad entrambe i database si associa un filtro di Bloom per tenere traccia, sommariamente, delle tuple che li compongono. Nel momento in cui si effettua un join, uno dei due server invia il "riassunto" dei dati in esso contenuti all'altro, il quale, a sua volta, invia al server originario il risultato finale. Di conseguenza, non viene trasferito l'intero contenuto di uno dei due database e, quindi, viene speso meno tempo per lo scambio di dati tra i due server ([39]). Un miglioramento di questa tecnica consiste nell'utilizzare uno *Spectral Bloom Filter*. Questo filtro, come illustrato in [8], sostituisce l'array binario con un vettore di m contatori. Ad esso, sono sempre associate k funzioni di hash, che mappano ciascun elemento nel range $[1, \ldots m]$. Il conteggio di un item è, dunque, salvato k volte. In questo modo eseguire operazioni di join distribuito (quali operazioni di conteggio o di filtraggio dei risultati sulla base di un valore di soglia) con maggiore efficienza. Per garantire maggiore accuratezza e velocità nello svolgere queste operazioni, quest'ultima struttura può essere sostituita con Count-Min sketch.

3.3.4 Sicurezza nelle password

Un ulteriore contesto in cui si usufruisce di Count-Min sketch, come introdotto in [39], consiste nel prevenire possibili *dictionary attack*. Per definizione, essi sono una forma di attacco di forza bruta in cui si cerca di indovinare la password dell'utente, basandosi, in particolar modo, su una lista nota di password più comuni (spesso un dizionario, da cui il termine). I meccanismi messi in atto per contrastare tale tipo di attacco consistono essenzialmente nel limitare il numero di tentativi nell'immissione della password e nel minimizzare la quantità di account che utilizzano le password più popolari (come "123456", ...). Nel paper [38], come ripreso poi da [39], viene quindi mostrato un meccanismo che impedisce agli utenti di scegliere password già popolari. Viene, cioè, implementato un oracolo, mediante un Count-Min sketch, che permette di identificare queste ultime. Nello sketch vengono salvati i conteggi delle password di ciascun utente e in seguito, con un procedimento del tutto analogo al tracciamento degli heavy hitter (infatti una password è considerata troppo comune se supera una soglia di popolarità, r), vengono determinate le password più frequenti. A questo punto, nel momento della creazione di un nuovo account, viene impedito all'utente di impostare la propria password ad una delle stringhe denotate come popolari dall'oracolo. É comunque ammesso un tasso minimo di falsi positivi, in

modo tale da confondere l'attacker che eventualmente ha avuto accesso all'oracolo. Alla fine, se la soglia di popolarità è impostata a r, e l'agente malevolo può effettuare al più G tentativi, la percentuale di possibili account compromessi è pari a rG ([38]).

3.3.5 Biologia Computazionale

Con il termine di biologia computazionale si intende lo "sviluppo e l'applicazione di metodi di analisi dei dati e teoretici, di tecniche di modellizzazione matematica e di simulazione computazionale per studiare i sistemi biologici, comportamentali e sociali. In altri termini, essa si configura come la scienza che usa dati biologici per sviluppare algoritmi o modelli in modo da comprendere i sistemi biologici e le loro relazioni." ([39], [26])
In questo campo, un importante applicazione di Count-Min sketch, suggerita in [42], consiste nel conteggio dei *k-mer*, che sono sottostringhe di lunghezza k, ottenute, per esempio, dal sequenziamento del DNA. L'analisi di questi elementi trova concretizzazione in problemi quali l'identificazione di sequenze ripetute (come i trasposoni, che sono elementi genetici in grado di spostarsi tra locazioni del genoma [36]), lo studio dei meccanismi di duplicazioni di sequenze del genoma, lo studio di eventuali sovrapposizioni di sequenze (utilizzata negli assemblatori di genomi) e la stima delle dimensioni del genoma([32]). La soluzione in questione, definita *Khmer*, utilizza tale tipo di sketch per ottenere operazioni di conteggio online efficienti da un punto di vista temporale e di memoria, che permettono di contare direttamente i dati nel momento in cui sono caricati, senza dover, quindi, accedere ripetutamente in memoria. In altre parole, durante la lettura del DNA, viene salvato il conteggio di ogni sequenza di interesse di k elementi in Count-Min sketch, le cui dimensioni variano a seconda dell'accuratezza stabilita dall'utente. Per determinare, dunque, la stima del conteggio dell'item i è sufficiente porre una point query Il problema di identificazione dei *k-mer* più frequenti può essere, poi, ricondotto all'individuazione di heavy hitter in un flusso cash register (3.2.1). In conclusione, rispetto ad altre soluzioni proposte, questo metodo permette di ottenere i conteggi dei singoli *k-mer* in modo significativamente più veloce.

3.3.6 Ulteriori applicazioni

Esistono, comunque, altre possibile applicazioni di Count-Min sketch. Ad esempio, gli autori dell'articolo [15] suggeriscono di utilizzare tale struttura dati per identificare variazioni nei data stream, sfruttando la proprietà di ricerca degli heavy hitter. In particolare, essi si concentrano sull'individuazione dei cosiddetti *deltoidi*, che sono item che presentano differenze notevoli in termini di traffico tra finestre temporali, interfacce o router distinti in una rete. Queste differenze possono, dunque, essere ricondotte a heavy hitter. Per calcolare i deltoidi tra due flussi, si associa a ciascuno di essi un Count-Min sketch. Per ogni nuovo item i del primo flusso si aggiorna lo sketch corrispondente con il valore p, pari al traffico dello stream, per un numero di volte pari alla finestra temporale considerata.

Allo stesso modo, sempre per l'item i, lo sketch del secondo flusso viene modificato della quantità $p^{'}$ (dimensioni del traffico) tante volte quanto grande è l'intervallo temporale esaminato. L'item i è dunque parte di un deltoide se la differenza tra i corrispettivi valori nei due sketch risulta essere maggiore di $\phi \sum_z D[z]$, dove $\sum_z D[z]$ rappresenta la somma delle differenze per ogni elemento, mentre $\phi \in [0, 1]$ è un parametro fissato dall'utente.

Un'altra applicazione consiste nell'esaminare flussi di dati che prendono la forma di array multidimensionali ([23]), che possono essere per esempio immagini in larga scala, in cui ogni riga della matrice coincide con una immagine che contiene valori di pixel o altri valori caratteristici [29].

3.3.7 Possibili future applicazioni

Come viene fatto notare in [9], oltre a trovare applicazioni nell'analisi di data stream, Count-Min sketch può essere utilizzato per "Sensor Networks", "Matrix Algorithm", geometria computazionale e "Privacy-Perserving Computations". Inoltre, per rendere più efficiente questa struttura dati, è possibile sfruttare l'eventuale architettura parallela del processore per calcolare le varie funzioni di hash per ogni item simultaneamente.

In conclusione, in questo capitolo si sono presentate le principali applicazioni di Count-Min sketch: sono stati prima considerati casi più astratti, quali $\phi - Quantiles$ (di cui, comunque, si è fatto accenno a qualche problema reale) ed heavy hitter, per poi considerare casi più pratici come il controllo di flussi di indirizzi IP, database, NLP, sequenziamento del DNA e sicurezza informatica. Infine, sono stati messi in evidenza possibili nuovi ambiti futuri per l'applicazione di Count-Min sketch. Si nota quindi come la struttura dati esaminata sia estremamente versatile e offra numerosi vantaggi, quali l'utilizzo di uno spazio in memoria di dimensioni fissate e relativamente ridotte, e il fatto di presentare un errore unilaterale, nel senso che il conteggio effettivo di ogni item può essere solo sovrastimato e mai sottostimato.

Capitolo 4

Studio sperimentale delle prestazioni di Count-Min sketch

In questo capitolo vengono illustrate, sotto forma di grafici, le prestazioni di Count-Min sketch in termini di errore medio (Sezione 4.2) e di match tra valore effettivo e valore stimato (Sezione 4.4) al variare dei parametri ε e δ (e conseguentemente della grandezza dello sketch) e delle dimensioni del flusso. L'implementazione dello sketch (in Java) e il codice per la realizzazione dei grafici (in cui si è fatto uso della libreria `Mathplotlib` di Python) sono disponibili al seguente link https://github.com/giuliaberaldo2002/CMS.git. Per prima cosa, viene quindi riportata una breve descrizione sull'implementazione di Count-Min sketch, poi ripresa nella Sezione 5.1, in cui si aggiunge l'oracolo e il predittore, al fine di migliorare le prestazioni di stima dello sketch.

4.1 Descrizione dell'implementazione di Count-Min sketch

In questa sezione vengono presentate le principali scelte implementative per Count-Min sketch e per il codice necessario per costruire i grafici rappresentanti l'errore medio e il numero di corrispondenze tra stime e conteggi effettivi in funzione dei parametri ε e δ.

L'implementazione in Java di Count-Min sketch segue la definizione illustrata nella Sezione 1.2.1. La classe `CountMinSketch` è caratterizzata, quindi, dagli attributi `epsilon` e `delta`, che indicano i corrispondenti parametri. Viene utilizzato quindi un array bidimensionale, le cui dimensioni nel costruttore sono inizializzate a $\lceil \frac{e}{\varepsilon} \rceil \times \lceil \ln \frac{1}{\delta} \rceil$. Infine, si utilizza un array di funzioni di hash di dimensione $\lceil \ln \frac{1}{\delta} \rceil$, ognuna di esse associata ad una distinta riga della matrice. Queste funzioni sono state implementate in una classe a parte, `HashFunc`. Essendo state definite secondo il metodo MAD, tale classe presenta come variabili di stato i quattro parametri, qui definiti a, b, p e w, che caratterizzano questa tipologia di funzioni. Seguendo la definizione, esse permettono di mappare ciascun elemento i nella posizione $(a \cdot i + b)\%\ p\%\ w$. In questa classe viene implementato un metodo che permette di generare

un numero primo casuale, maggiore di w. I primi due parametri, a e b, sono interi casuali ottenuti sfruttando la classe `java.util.Random`. Inoltre, `CountMinSketch` è progettata per rispondere esclusivamente a point query. Di conseguenza, per rispondere a $Q(i)$ il metodo `estimateFrequency` ritorna il valore minimo fra tutti i valori salvati nello sketch per l'item i.
Nelle analisi effettuate si considera un data stream di tipo cash register, dove ovvero sono possibili solo inserimenti e non rimozioni di item.

Per quanto riguarda, invece, la parte di realizzazione dei grafici sull'errore medio e sul numero di match, si è usufruito della libreria `Mathplotlib` di Python. Più nel dettaglio, a partire dai file di log generati nel Main, riportanti i valori di interesse, è stata implementata una funzione, `read_log_file`, per leggere questi dati ed organizzarli in una mappa. In questo modo, sfruttando le coppie chiave-valore, è stato poi possibile realizzare i grafici.

Per maggiori dettagli, si rimanda al link indicato all'inizio del capitolo.

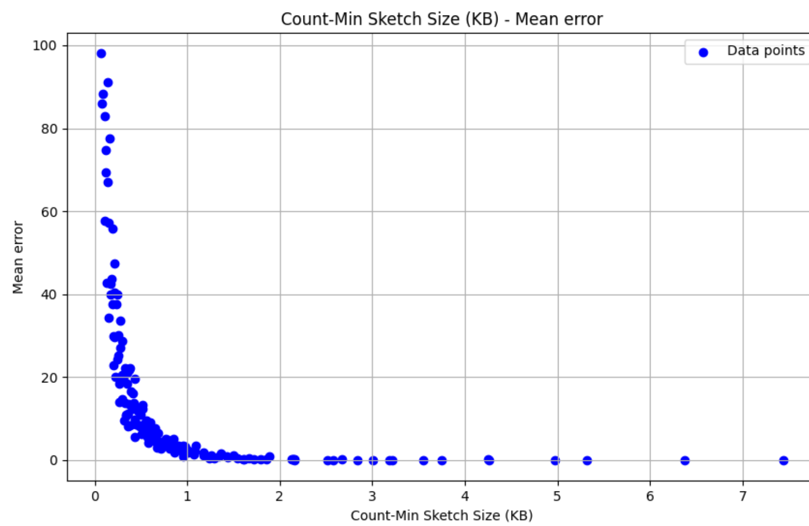
4.2 Dipendenza dell'errore medio dalle dimensioni di Count-Min sketch

In questa sezione si studia come varia l'errore medio nella risposta ad una point query, in funzione delle dimensioni (espresse in kilobyte) dello sketch. Si esamina, quindi, il comportamento di questa struttura dati con distribuzioni di Zipf di diverse grandezze.

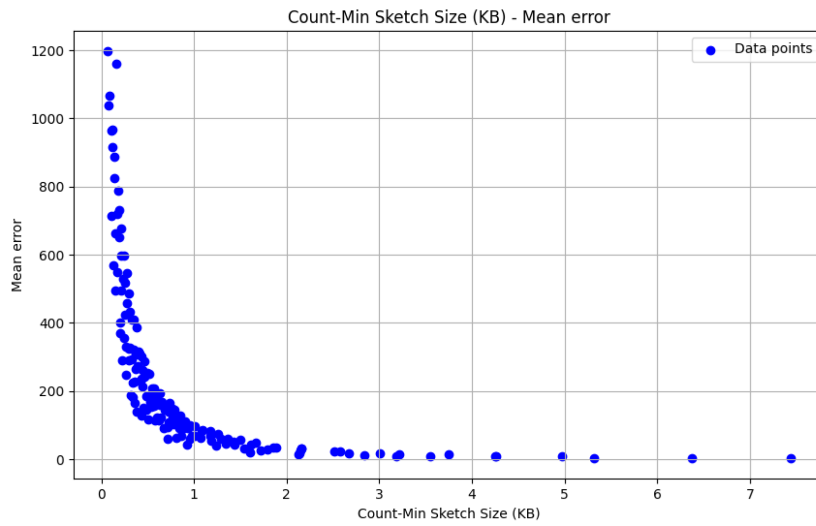
Si rappresenta graficamente la dipendenza dell'errore medio della risposta ad una point query al variare delle dimensioni di Count-Min sketch. Più nel dettaglio, nei seguenti grafici si rappresenta sull'asse delle ascisse la grandezza, espressa in KB, di Count-Min sketch. Essa si configura essere pari a $dim(CMS) = w \cdot d = \frac{\lceil \frac{e}{\varepsilon} \rceil \lceil \ln \frac{1}{\delta} \rceil \cdot 4}{1024}$, avendo supposto che ciascuna cella dello sketch memorizzi un intero, che occupa, per l'appunto, 4 Byte. Sull'asse delle ordinate è stato invece riportato l'errore medio, pari a $\frac{1}{N} \sum_{i=1}^{n} |a_i - \hat{a}_i| a_i$. Sono, quindi, state studiate diverse distribuzioni di Zipf di numeri naturali, facendo variare il numero complessivo di elementi nello stream e di item distinti. Per la precisione, sono state analizzate distribuzioni di 1000 elementi di cui 100 distinti, 10000 elementi con 1000 valori differenti, 100000 item con 10000 valori diversi ed infine 500000 elementi di cui 20000 distinti. Tali stream sono stati generati applicando la definizione di distribuzione di Zipf (1.3.4). Per maggiori dettagli implementativi, si rimanda alla classe `ZipfianGenerator` al seguente link https://github.com/giuliaberaldo2002/CMS.git.
Ogni grafico analizza quindi l'errore medio nella stima dei conteggi commesso da Count-Min sketch per una distribuzione differente. Per ognuna di esse, si sono prese in considerazione 100 dimensioni distinte della struttura dati in esame. Più nel dettaglio, si sono innanzitutto considerati degli intervalli per i possibili valori assunti da ε e δ. Per il primo parametro si è scelto l'intervallo [0.01, 0.5], mentre per il secondo il range [0.001, 0.1]. Questo significa

che le dimensioni dello sketch variano nell'intervallo [0.07, 7.4375] KB. Per determinare le misure campione, si sono, quindi, presi 100 valori tra loro equidistanti. In altre parole, si è suddiviso sia il range [0.01, 0.5] che [0.001, 0.1] in 999 sottointervalli e sono stati scelti i valori alle estremità di ciascuno di questi ultimi. I 10 valori assunti da ε sono, dunque, stati combinati con quelli di δ, ottenendo in tutto $100 \cdot 100 = 10000$ campioni, cioè 10000 grandezze differenti di Count-Min sketch. Di seguito vengono riportati i grafici così ottenuti:

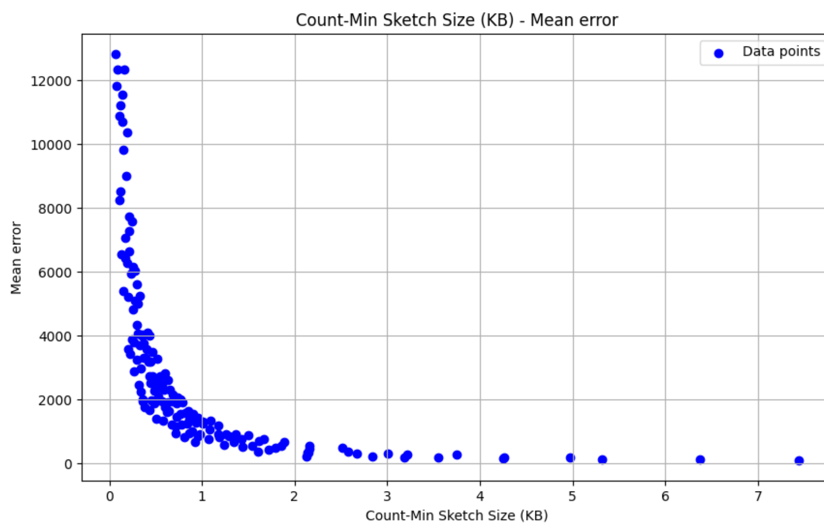
1. Distribuzione di 1000 elementi con 100 valori distinti



2. Distribuzione di 10000 elementi con 1000 valori distinti



3. Distribuzione di 100000 elementi con 10000 valori distinti



4. Distribuzione di 500000 elementi con 20000 valori distinti

Si può osservare come l'errore medio, in tutti i casi decresce con un andamento circa esponenziale con l'aumentare delle dimensioni del Count-Min sketch.
Ciò può essere dimostrato secondo il seguente ragionamento: innanzitutto si osserva che ogni contatore nel Count-Min sketch presenta un determinato errore nella stima del conteggio di un elemento, il quale è tanto maggiore quante più collisioni si verificano. Aumentando la dimensione dello sketch, la probabilità di collisioni diminuisce.
Con uno sketch di piccole dimensioni, quindi, ci sono poche posizioni disponibili, il che rende le collisioni più probabili (specie all'aumentare del numero di elementi dello stream) e l'errore più alto. Per questo motivo si osserva che a parità di dimensioni dello sketch, per distribuzioni composte da un numero più elevato di elementi, l'errore atteso è maggiore (nel caso della distribuzione con 500000 elementi, ad esempio, l'errore massimo si attesta a circa 600000, mentre in quella di 1000 elementi a 100).
Man mano che la dimensione di Count-Min sketch aumenta (al diminuire di ε e δ) lo spazio disponibile per mappare i conteggi degli elementi cresce, riducendo la probabilità di collisioni. Di conseguenza l'errore medio si riduce.
La riduzione della probabilità di collisione avviene rapidamente all'inizio, portando a una drastica riduzione dell'errore, specie per la crescita logaritmica del numero di righe in funzione dell'inverso di δ. Per Count-Min sketch di dimensioni ridotte, quindi, l'errore scende rapidamente, poi la curva tende a raggiungere un asintoto orizzontale nel momento in cui la grandezza di Count-Min sketch è sufficientemente grande. In questo caso, ulteriori aumenti nella dimensione portano a miglioramenti marginali, perché le collisioni diventano estremamente rare. Questo fa sì che l'errore si stabilizzi in corrispondenza di un valore basso, eventualmente anche nullo (come si osserva nel primo grafico).
Inoltre, l'errore di un elemento in ogni riga risulta essere indipendente dagli altri, data la pairwise independence delle funzioni di hash. Grazie a questa proprietà, quando si combinano i conteggi approssimati in ogni riga per ottenere la stima del conteggio di un elemento, la somma degli errori complessivi tende a stabilizzarsi e a ridursi man mano che

aumentano sia il numero di funzioni di hash che il numero di colonne. Si ricorda, infine, che per definizione l'errore per ogni item è limitato dal termine $||a||_1\varepsilon = \Theta\left(||a||_1\frac{e}{w}\right)$ con probabilità $1-\delta = 1-\Theta(e^{-d})$. Si nota quindi la relazione di proporzionalità inversa con il numero di colonne dello sketch.

Si prova a fornire un'analisi più formale e precisa sull'errore medio, riprendendo i teoremi e le osservazioni dimostrati in [24] (si adotta la stessa struttura). In questo caso vengono, quindi, mostrati un limite superiore e inferiore dell'errore medio, che permettono di concludere che esso decresce in modo circa inversamente proporzionale con le dimensioni dello sketch. Più precisamente, si forniscono due bound per l'errore medio di un qualsiasi sketch, considerando la casualità delle funzioni di hash. Si ricorda che $a_i \propto \frac{1}{i} = c_i$ e, per semplicità, si suppone che $c_i = \frac{1}{i}$.
Osservazione 1 : per un numero di item n distinti tale che $n > 250$ vale che

$$\ln(n+1) < \sum_{i=1}^{n} \frac{1}{i} < \ln n + 0.58$$

questo significa che una somma armonica (e quindi l'approssimazione della somma dei conteggi degli elementi) appartiene a $\Theta(\log n)$
A questo punto, si riprende il concetto di errore medio introdotto da una singola funzione di hash.
Teorema 1 : Data una funzione di hash uniforme $h : 1,...n \to [0, w-1]$, l'errore medio nella stima che viene commesso si attesta a $\Theta\left(\frac{(\log n)^2}{w}\right)$

Dimostrazione. Eliminando il fattore $\frac{1}{N}$ dalla definizione di errore medio, si ricava che

$$\begin{aligned}\mathbb{E}[C,\hat{C}] = \mathbb{E}[\sum_{i=1}^{n}(\hat{c}_i - c_i)\cdot c_i] &= \sum_{i=1}^{n}\mathbb{E}[X_{i,j}]\cdot c_i \\ &= \sum_{i=1}^{n}\Theta\left(\frac{\Theta(\log n) - c_i}{w}\right)\cdot c_i = \Theta\left(\frac{(\log n)^2}{w}\right)\end{aligned}$$

Negli ultimi due passaggi si è applicata l'osservazione 1. □

A questo punto, si suddivide l'intervallo $[0, w\ln n]$ in $m+1$ sottointervalli i cui estremi sono $\Theta\left(\frac{n}{w}\right)^{1+\gamma} = r_0 \leq \cdots \leq r_m = w\ln n$, con γ parametro da determinare in un secondo momento. Ogni estremo è tale che $r_l = \left(\ln\frac{n}{w} + \ln r_{l+1}\right)\ln(r_{l+1}^{\gamma})$.
Da ciò in [24] viene dimostrato che $\ln r_{l+1} \geq 2\ln r_l$ e da questo segue che $\forall l \geq 1$ $\sum_{j\geq i}(\ln r_j)^{-c} = O((\ln r_i)^{-c})$.
Dunque, poiché l'errore viene interpretato come una variabile aleatoria non negativa, è possibile calcolare il valore atteso dell'errore di stima per l'item i come:

$$
\begin{aligned}
\mathbb{E}[e_i] &= \int_0^{+\infty} \Pr[e_i \geq x]\, dx \\
&= \int_0^{\frac{r_0}{w}} \Pr[e_i \geq x]\, dx + \sum_{j=0}^{m-1} \int_{\frac{r_j}{w}}^{\frac{r_{j+1}}{w}} \Pr[e_i \geq x]\, dx \\
&\leq O\left(\frac{r_0}{w}\right) + \sum_{j=0}^{m-1} \int_{\frac{r_j}{w}}^{\frac{r_{j+1}}{w}} \left(\Pr[e_i \geq x | e_i \leq \frac{r_{i+1}}{w}] + \sum_{q=j+1}^{m-1} \Pr[e_i \geq \frac{r_q}{w} | e_i < \frac{r_{q+1}}{w}] \right) dx
\end{aligned}
$$

In [24] viene quindi dimostrato che $\Pr[e_i \geq x | e_i \leq \frac{r_{i+1}}{w}] = O\left(\frac{d(\ln\frac{n}{w} + \ln r_{i+1})}{w(\ln r_{i+1})^{\gamma(k-1)-1}}\right)$.
Il secondo termine dell'integrale è invece pari a $\int_{\frac{r_j}{w}}^{\frac{r_{j+1}}{w}} \sum_{q=j+1}^{m-1} \Pr[e_i \geq \frac{r_q}{w} | e_i < \frac{r_{q+1}}{w}]\, dx = O\left(\frac{d(\ln\frac{n}{w} + \ln r_{i+2})}{w(\ln r_{i+2})^{\gamma(k-1)}}\right)$.
A fronte di questi risultati, si ottiene che

$$
\begin{aligned}
\mathbb{E}[e_i] &\leq O\left(\frac{r_0}{w}\right) + \sum_{j=0}^{m-1} \int_{\frac{r_j}{w}}^{\frac{r_{j+1}}{w}} \left(\Pr\left[e_i \geq x \mid e_i \leq \frac{r_{j+1}}{w}\right] + \sum_{q=j+1}^{m-1} \Pr\left[e_i \geq \frac{r_q}{w} \mid e_i < \frac{r_{q+1}}{w}\right] \right) dx \\
&= O\left(\frac{r_0}{w}\right) + \sum_{j=0}^{m-1} O\left(\frac{d\left(\ln\frac{n}{w} + \ln r_{i+1}\right)}{w\left(\ln r_{i+1}\right)^{\gamma(d-1)-1}}\right) + O\left(\frac{d\left(\ln\frac{n}{w} + \ln r_{j+2}\right)}{w\left(\ln r_{j+2}\right)^{\gamma(d-1)}}\right) \\
&= O\left(\frac{r_0}{w}\right) + O\left(d\frac{\ln\frac{n}{w}}{w\ln r_1^{\gamma(d-1)-1}}\right) + O\left(\frac{d}{w\ln r_1^{\gamma(d-1)-2}}\right).
\end{aligned}
$$

Quest'ultima espressione richiede che $\gamma(d-1) - 2 \geq 1$, quindi è sufficiente porre $\gamma = \frac{3}{d-1}$.
Da questo si ricava quindi che, per ogni item i,

$$
\mathbb{E}[e_i] = O\left(\frac{r_0}{w}\right) = O\left(\frac{\left(\ln\frac{n}{w}\right)^{\frac{d+2}{d-1}}}{w}\right).
$$

Da queste considerazioni deriva il seguente teorema:
Teorema 2 L'errore atteso di Count-Min sketch di dimensioni dw per stimare degli elementi i cui conteggi sono distribuiti secondo una distribuzione di Zipf è pari a $O\left(\frac{\ln n(\ln(n/w))^{\frac{d+2}{d-1}}(n/w)}{w}\right)$.

Dimostrazione. La dimostrazione si basa sulla definizione di errore atteso commesso dallo sketch e sul risultato appena ottenuto

$$
\mathbb{E}[Err(A, \hat{A}_{CM})] = \sum_{i=1}^{n} \mathbb{E}[e_i]a_i = O\left(\frac{\left(\ln\frac{n}{w}\right)^{\frac{d+2}{d-1}}}{w}\log n\right)
$$

□

Si può osservare a questo punto che, per ogni item i, $\Pr[e_i \geq \frac{1}{2\left(\frac{w}{d}\right)\ln d+1}] \geq 1 - \frac{1}{d}$ [24].

Questo implica che per un Count-Min sketch di dimensione w' vale che

$$\mathbb{E}[Err(A, \hat{A}_{CM})] = \sum_{i=1}^{n} \mathbb{E}[e_i]a_i \geq \frac{d-1}{2w' \ln d + d}$$

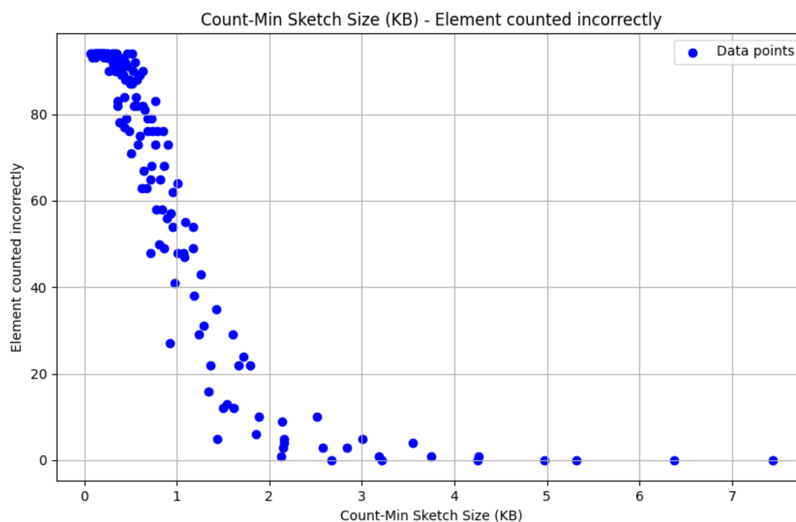
per la prima osservazione fatta. Ponendo quindi $w' = wd$, il corrispondente limite inferiore ottenuto è $\Omega\left(\frac{\ln n}{w \ln d}\right)$.

In conclusione, l'errore medio in funzione delle dimensioni dello sketch decresce circa in modo inversamente proporzionale con il numero di colonne della matrice. A ciò contribuisce, poi, l'indipendenza degli errori tra le diverse righe della tabella. Questo fa sì che la decrescita dell'errore medio nella stima dei conteggi sia molto rapida all'aumentare delle dimensioni di Count-Min sketch.

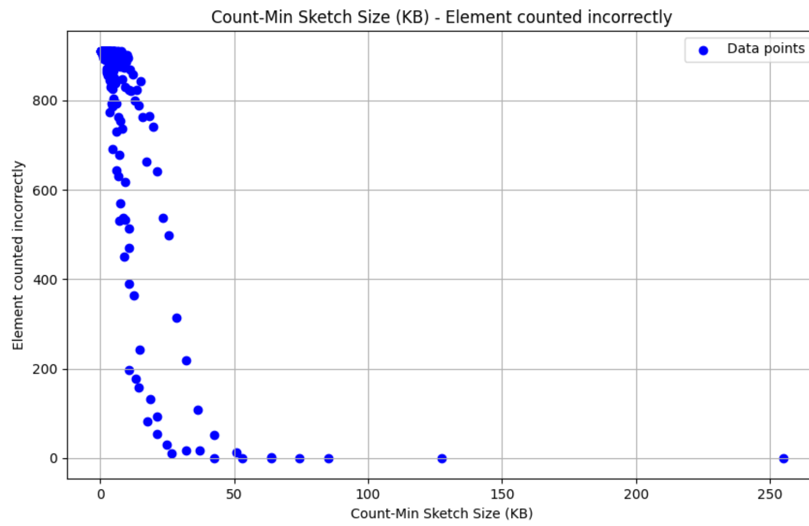
4.3 Numero di elementi contati erroneamente in funzione delle dimensioni di Count-Min sketch

Un procedimento simile e le stesse distribuzioni considerati nella sezione precedente vengono utilizzati per determinare la dipendenza del numero di elementi erroneamente contati in funzione delle dimensioni dello sketch, espressa sempre in KB. Tuttavia, per ogni distribuzione si scelgono dei range differenti entro cui far variare δ e ε, al fine di evitare che per stream con un numero elevato di item, il numero di elementi con una stima diversa dal conteggio effettivo sia costante ed eccessivamente elevato. Perciò i range di dimensioni adottati sono circa [0.07, 7.4375]KB, [0.33, 254.9]KB, [7.4, 3716.3]KB, [21.3, 9875.1]KB rispettivamente per le distribuzioni di 1000, 10000, 0000, e 500000 item.

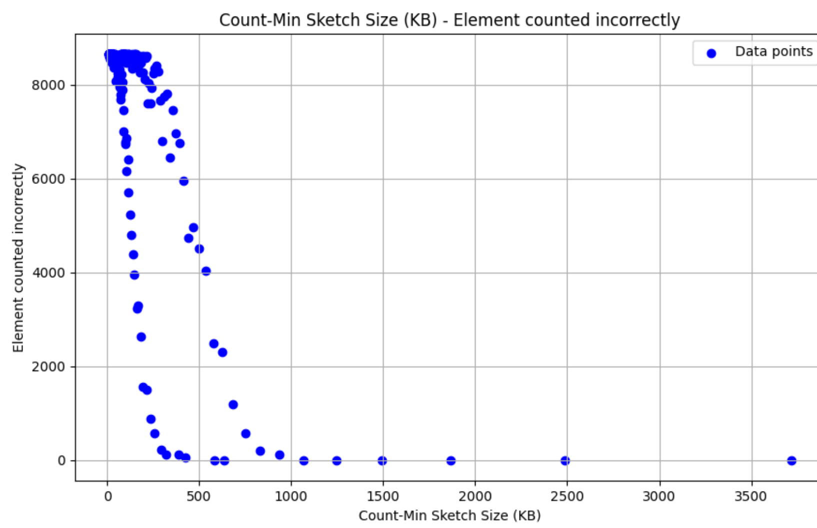
1. Distribuzione di 1000 elementi con 100 valori distinti



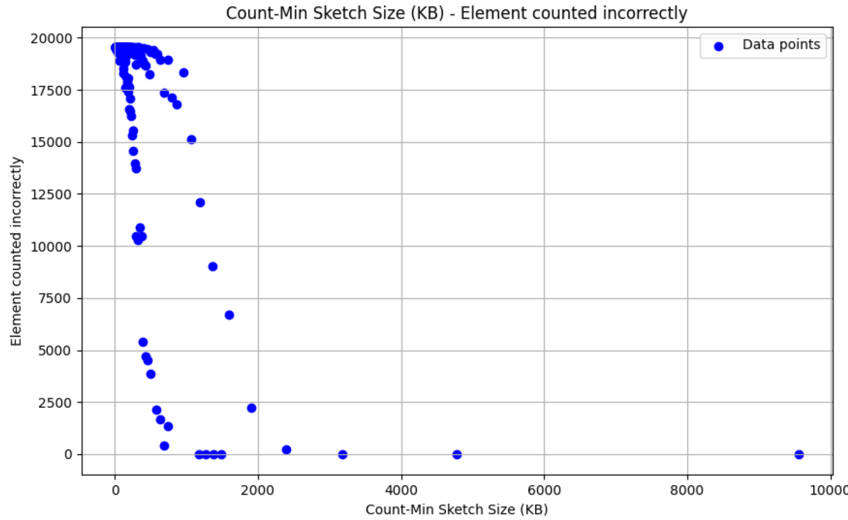
2. Distribuzione di 10000 elementi con 1000 valori distinti



3. Distribuzione di 100000 elementi con 1000 valori distinti



4. Distribuzione di 500000 elementi con 20000 valori distinti



Ciò che si osserva da questi grafici consiste nel fatto che il numero di elementi in cui la stima non coincide con il conteggio esatto risulta essere circa costante per dimensioni molto ridotte dello sketch, per poi decrescere, prima più ripidamente, poi più lentamente al diminuire dei parametri ε e δ. Le motivazioni che possono essere addotte in questo caso sono molto simili a quelle illustrate nella sezione 4.2. Infatti, all'aumentare delle colonne dello sketch, diminuisce la probabilità che si verifichino collisioni. Di conseguenza, aumenta anche la probabilità che l'elemento sia conteggiato correttamente.

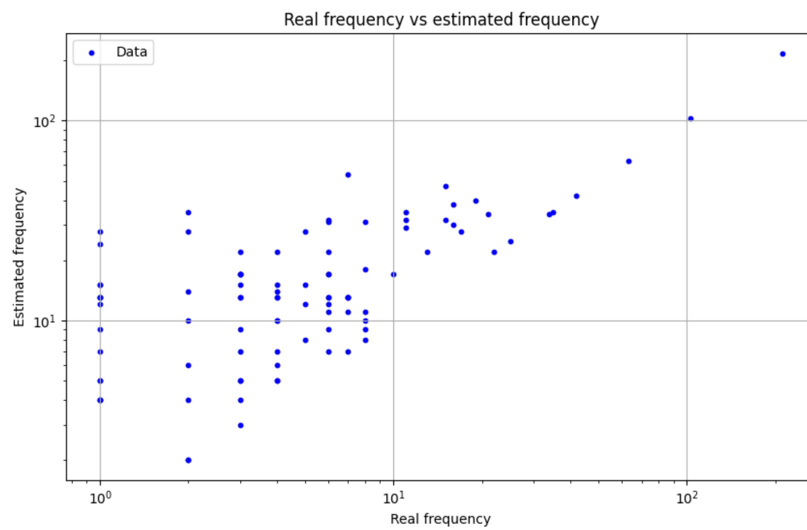
4.4 Dipendenza del numero di match e mismatch dai parametri ε e δ

Un'ultima possibile analisi grafica consiste nel visualizzare gli item del flusso di dati in un piano cartesiano in cui le coordinate (espresse in scala logaritmica) sono fissate in modo tale che nell'asse delle ascisse sia riportato l'effettivo conteggio di ciascun, mentre in corrispondenza dell'asse delle ordinate sono riportati le corrispondenti stime ottenute salvando i dati in Count-Min sketch. Se tutte le stime coincidono con i conteggi reali, allora il grafico riporta l'andamento di una retta con coefficiente angolare unitario e passante per l'origine, della famiglia, quindi, $y = x$. Come esempio illustrativo, si utilizza un data stream di dimensioni ridotte, costituito da 1000 elementi organizzati secondo un distribuzione di Zipf. Viene effettuata questa scelta in modo tale da poter studiare diverse combinazioni possibili dei valori di ε e δ, evitando una computazione eccessiva nel calcolatore. É comunque possibile generare grafici con distribuzioni di dimensioni maggiori, rappresentando poi in scala logaritmica i risultati ottenuti in modo da avere un visualizzazione più compatta.
Si fanno variare i parametri ε e δ nel seguente modo: prima si mantiene fissato ε e si modifica δ decrementandolo ad ogni iterazione (aumentando di conseguenza il numero di

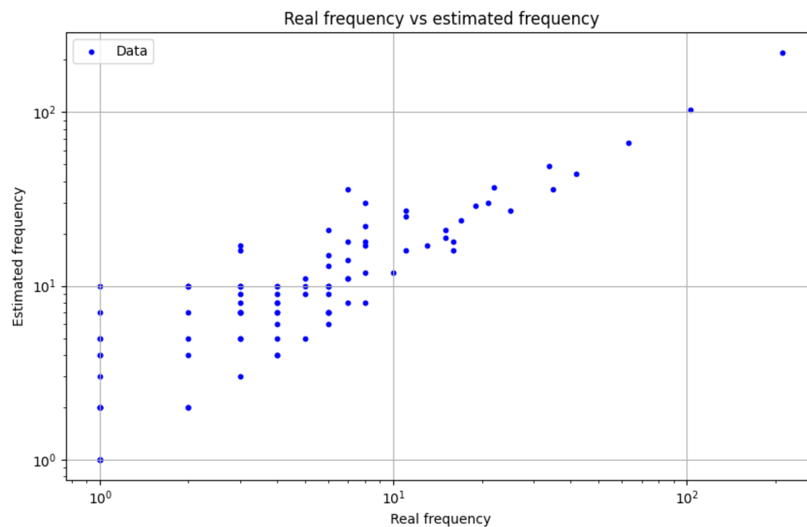
righe). In seguito, si fissa il valore di δ e si fa variare il ε. Seguendo questo procedimento, si ottengono dunque i seguenti risultati:

- Primo caso: si mantiene fissato $\varepsilon = 0.1$, quindi Count-Min sketch si compone di 28 colonne e si varia δ

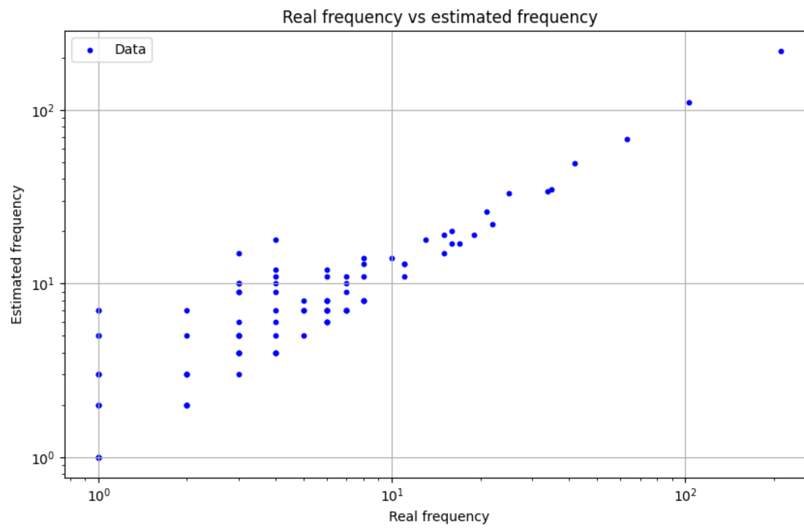
  1. $\delta = 0.01$, 5 righe



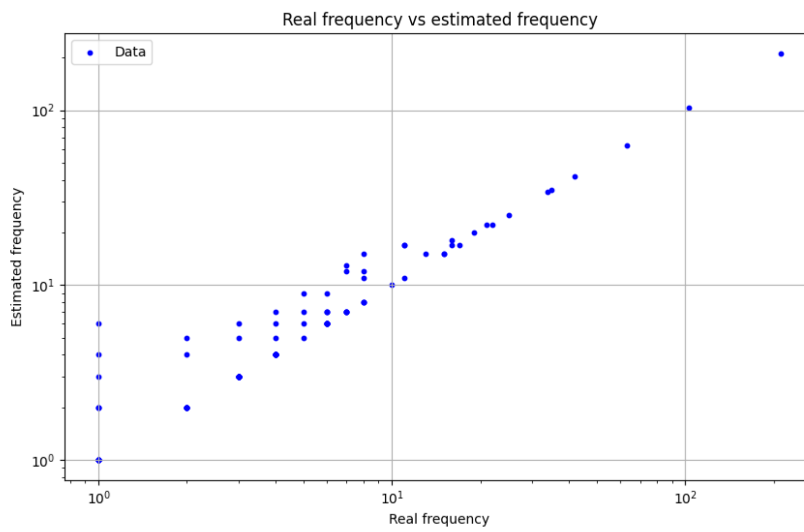
  2. $\delta = 0.001$, 7 righe



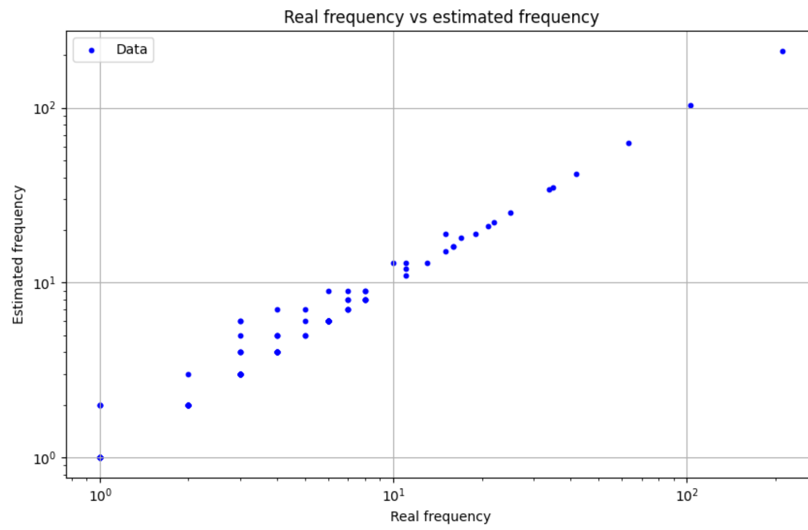
  3. $\delta = 10^{-5}$, 12 righe



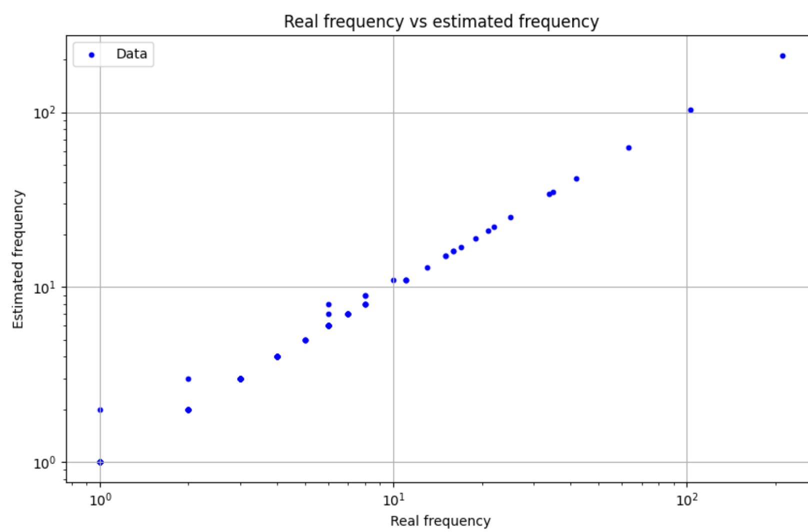
4. $\delta = 10^{-10}$, 24 righe



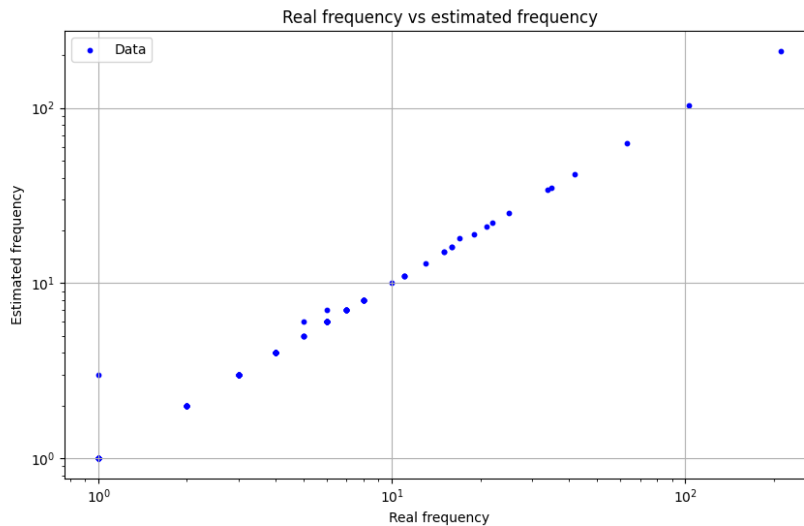
5. $\delta = 10^{-20}$, 47 righe



6. $\delta = 10^{-40}$, 93 righe



7. $\delta = 10^{-50}$, 116 righe

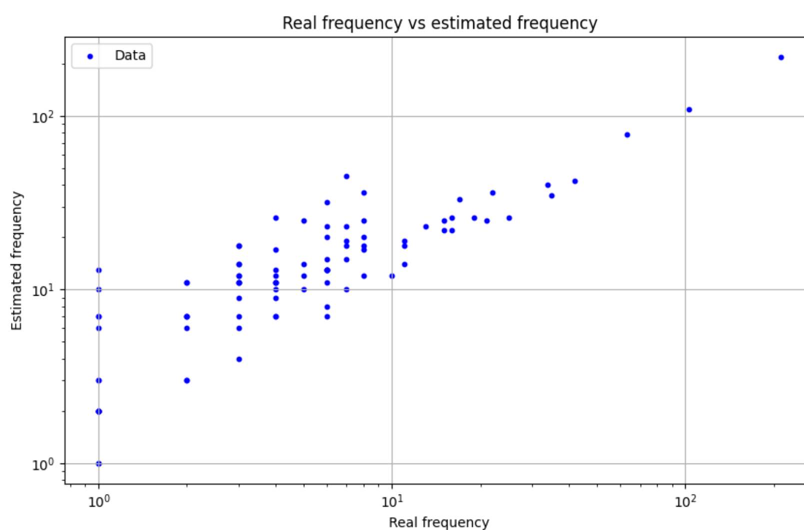


L'analisi si ferma a $\varepsilon = 10^{-50}$ per limitazioni di memoria del calcolatore.

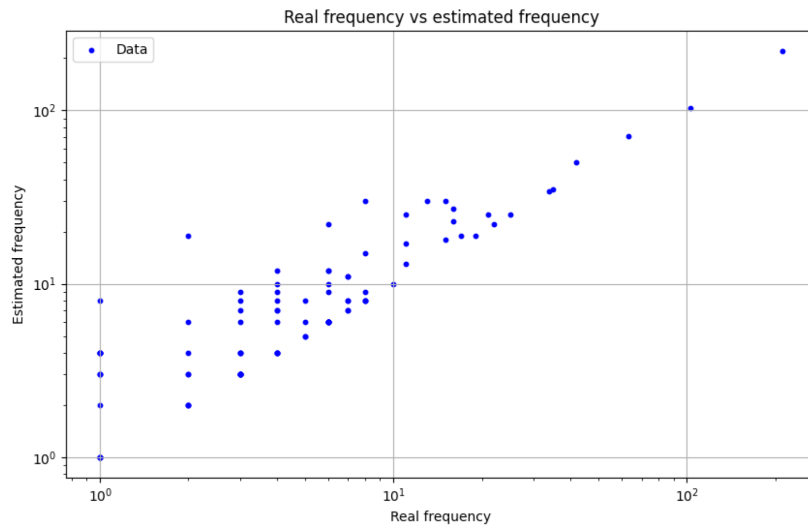
Si può notare, comunque, che al diminuire di δ (e quindi all'aumentare del numero di righe della matrice) il numero di elementi contati correttamente dal Count-Min sketch cresce, come testimoniato dal fatto che il grafico, per valori decrescenti di δ, approssima la retta $y = x$. Al diminuire del parametro considerato infatti, la distribuzione dei punti si appiattisce maggiormente, fino ad assumere un andamento circa lineare. Nell'ultimo caso, per esempio, l'andamento è prossimo a quello di una retta.

- Secondo caso: si mantiene $\delta = 0.1$ (si fissa il numero di righe a 3) e si fa variare ε:

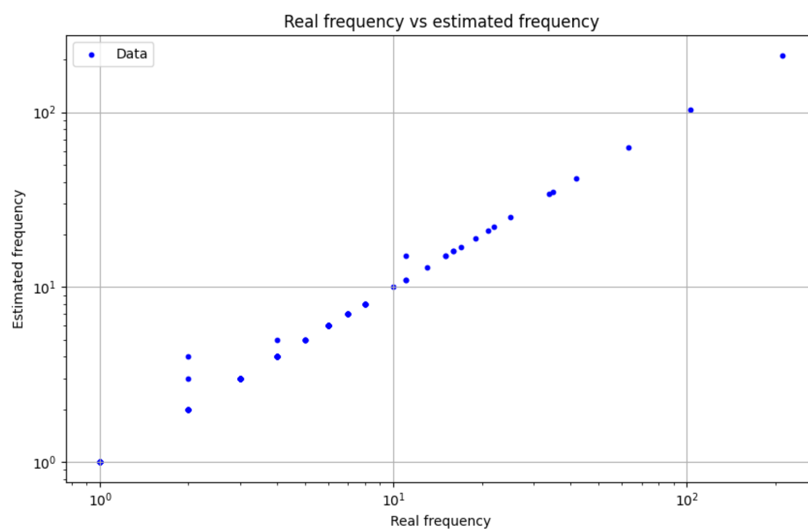
    1. $\varepsilon = 0.1$, 28 colonne



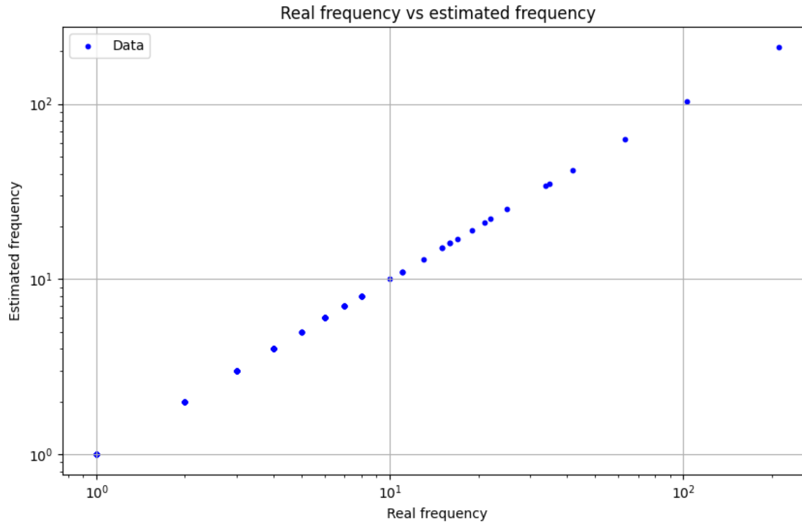
    2. $\varepsilon = 0.05$, 55 colonne



3. $\varepsilon = 0.01$, 280 colonne



4. $\varepsilon = 0.008$, 340 colonne



In questo caso si nota che mantenendo δ fissato a 0.1 (ottenendo quindi una matrice con 28 colonne) per valori di ε inferiori circa a 0.008, tutti gli elementi sono correttamente conteggiati. In questo caso infatti, in ogni riga ciascun conteggio può essere mappato in 340 posizioni differenti. Di conseguenza, le collisioni sono rare, se non nulle. Riprendendo i risultati ottenuti e la notazione utilizzata nella sezione 2.1.1, si ottiene che con 100 valori distinti, l'errore di stima medio commesso è pari a $\frac{100}{w} \approx 0.29$, che risulta quindi trascurabile.

Coerentemente con la teoria, inoltre, si ottengono eventualmente solo sovrastime, come testimoniato dal fatto che i vari "punti" nel grafico sono collocati sopra la retta con coefficiente angolare unitario e passante per l'origine. Per di più, le sovrastime sono più marcate in corrispondenza di elementi poco frequenti nello stream (che per le proprietà della distribuzione di Zipf costituiscono la maggioranza del flusso stesso). Qualora si verifichi, infatti, una collisione tra un heavy hitter e un item a minor frequenza, la stima di quest'ultimo si discosterà notevolmente dal conteggio effettivo. Per risolvere questo problema è stato, quindi, introdotto l'uso di un oracolo o di un predittore nella sezione 5.2. Le osservazioni che possono essere fatte a riguardo sono quindi le seguenti: l'accuratezza delle stime nel Count-Min Sketch migliora con l'aumento sia del numero di colonne w che del numero di righe (e quindi di funzioni di hash) d. Tuttavia, come osservato dai grafici, la velocità con cui l'accuratezza migliora differisce tra questi due parametri. Per chiarire la relazione, si studiano le caratteristiche di ε e δ.

- Influenza di ε: Si ricorda innanzitutto che il parametro ε rappresenta l'errore massimo ammesso nelle stime e, inoltre, determina il numero di colonne della matrice, pari a $w = \lceil \frac{e}{\varepsilon} \rceil$. Si nota quindi che w ed ε sono legati da una relazione di proporzione inversa. Di conseguenza, riducendo ε, si riduce l'errore si aumenta il numero di colonne, il che riduce l'errore additivo massimo nelle stime. La riduzione di ε migliora l'accuratezza delle stime riducendo l'errore massimo additivo in modo lineare. É

stato poi mostrato che se il numero di colonne è superiore del numero di elementi distinti nel flusso, le stime coincidono con i conteggi effettivi.

Perciò, all'aumentare di w, tale parametro diminuisce e l'accuratezza migliora significativamente. Dimezzando ε, per esempio, il numero di colonne aumenta e si riduce la possibilità di collisioni tra elementi distinti.

- Influenza di δ: Innanzitutto, si ricorda che δ si configura come la probabilità di errore accettabile. Il numero di righe (e dunque di funzioni di hash) è legato a tale parametro dalla seguente relazione: $d = \lceil \ln \frac{1}{\delta} \rceil$, che si configura come relazione logaritmica inversa. Di conseguenza, aumentando il numero di righe, l'accuratezza delle stime migliora più rapidamente inizialmente, poiché una piccola riduzione del parametro ivi considerato può tradursi in un aumento significativo del numero di righe a causa della relazione logaritmica. L'aggiunta di righe nella matrice comporta dunque un miglioramento dell'accuratezza perché ogni nuova funzione di hash fornisce un'ulteriore dimensione su cui effettuare stime più precise, riducendo il rischio di collisioni e quindi di errori. Tuttavia questa crescita si configura come logaritmica, quindi a parità di diminuzione del valore dei parametri ε e δ, quest'ultimo parametro contribuisce in modo meno significativo al miglioramento nell'accuratezza della stima. Inoltre, avendo in questo esempio fissato ε a 0.1, per un totale di 28 colonne, le collisioni (anche elevate) in ciascun bucket sono inevitabili. Inoltre, come mostrato dal grafico, esse sono più evidenti per elementi poco frequenti, piuttosto che per gli heavy hitter. In conclusione, la riduzione di δ migliora l'accuratezza delle stime riducendo la probabilità di ottenere un errore significativo. Tuttavia, la probabilità di errore diminuisce in modo meno significativo rispetto alla stessa riduzione del parametro ε.

In conclusione, in un Count-Min sketch, l'accuratezza delle stime è influenzata sia dal parametro ε che dal parametro δ. La riduzione del solo parametro ε porta a un miglioramento dell'accuratezza più rapido rispetto alla riduzione del solo parametro δ, per i motivi illustrati. Questo comportamento può essere attribuito al fatto che l'aggiunta di funzioni di hash (una per riga) ha un impatto minore sulla riduzione degli errori rispetto all'aumento della larghezza dell'array. Aumentando il numero di colonne, diminuisce di conseguenza l'errore massimo ammesso che presenta una probabilità inversamente proporzionale al numero di righe dello sketch.

Capitolo 5

Miglioramento delle prestazioni di Count-Min Sketch con tecniche di predizione

In questo capitolo vengono presentate tecniche predittive, che, affiancate a Count-Min sketch, permettono di migliorare l'accuratezza della stima dei conteggi degli elementi del data stream. Questi algoritmi aggiuntivi individuano, in modo esatto o approssimato, gli heavy hitter, il cui conteggio viene salvato in un bucket distinto, evitando, così, collisioni costose. In particolar modo, si espongono innanzitutto i risultati ottenuti in [24], in cui vengono sfruttati algoritmi predittivi di machine learning. In seguito, si mostra come, utilizzando un oracolo, vale a dire un algoritmo che identifica esattamente gli elementi heavy hitter, o un predittore, che cerca di effettuare una predizione sull'identità degli elementi più frequenti, le prestazioni di Count-Min sketch nella stima dei conteggi degli elementi migliora. Questo è dimostrato attraverso dei grafici che confrontano l'errore medio e il numero di corrispondenze nel caso in cui si usa solo Count-Min sketch, l'oracolo e il predittore. Infine, si riportano tecniche alternative per migliorare la stima nei conteggi, quali Augmented sketch o k-ary sketch, gSketch e Frequency-Aware Counting sketch, che si applicano in contesti particolari.

Per quanto descritto fino a questo punto, Count-Min sketch si configura come uno "streaming algorithm" (che, come visto, è una classe di algoritmi che utilizzano uno spazio di memoria ristretto rispetto alle dimensioni dei dati che elaborano, calcolando in questo modo una stima del conteggio per ciascuno di essi), che presuppone l'assenza di pattern nei dati processati. Tuttavia, i data stream reali, come per esempio flussi di traffico rete o file di testo, generalmente possiedono correlazioni intrinseche tra i dati contenuti: ad esempio, in un testo la frequenza delle parole è inversamente correlata alla loro lunghezza, mentre in una rete, alcuni applicazioni tendono a generare più traffico di altre ([24]). Sfruttando queste proprietà, dunque, è possibile ottenere algoritmi per rispondere a point query più efficienti.
Seguendo questa intuizione, gli autori in [24] suggeriscono di associare all'algoritmo un

"learning model" (che non si configura come un oracolo perfetto) che permette di sfruttare le proprietà del flusso di dati senza essere specifico ad un particolare pattern od avere delle conoscenze a priori sulle proprietà del flusso. Altre applicazioni simili, come suggerito in [31], riguardano i problemi di ski rental, scheduling, bloom filter, bin packing, queueing, caching, page migration. Per esempio, si potrebbe utilizzare un algoritmo in grado di riconoscere i pattern delle collisioni, inevitabili negli "hashing-based algorithm", di cui Count-Min sketch fa parte. L'errore complessivo dipende infatti da tale pattern: collisioni tra elementi più frequenti comportano un errore di stima più elevato, che invece dovrebbe essere minimizzato. Questo è dovuto al fatto che gli heavy hitter hanno un'alta frequenza, quindi l'incremento nelle celle della matrice dovuto a loro è significativo. Quando un altro elemento (non heavy hitter) condivide una cella con un questo, la sua stima di frequenza sarà sovrastimata di una quantità notevole.

5.1 Introduzione di algoritmi di Machine Learning e modifiche apportate a Count-Min sketch

In questa sezione vengono innanzitutto introdotte brevemente le caratteristiche principali dei learning algorithm, proposti in [24], che analizzano il flusso, estrapolandone le proprietà principali. Successivamente, si illustrano le modifiche apportate a Count-Min sketch e la procedura seguita per migliorare le prestazioni in termini di accuratezza nella risposta a point query. Viene, dunque, riportata la dimostrazione in [24] che mostra un miglioramento logaritmico nell'errore di stima dello sketch.

Si considera, innanzitutto, il vettore che rappresenta il flusso di dati, a. Si utilizza, quindi, un sottoinsieme di esso, indicato come $a^{'}$, per effettuare un training dell'algoritmo predittivo che permette di estrapolare le proprietà degli heavy hitter. Infatti, gli algoritmi utilizzati non mirano a identificare l'identità di questi, quanto le proprietà particolari e i pattern che soddisfano. Per esempio, nel caso di un testo, questi algoritmi individuano come heavy hitter le parole più corte, che probabilisticamente sono le più frequenti. In questo modo, l'algoritmo può identificare come heavy hitter anche word che non sono presenti nel training set $a^{'}$ [24]. Alla matrice utilizzata per Count-Min Sketch si apportano alcune modifiche, al fine di ottimizzare le prestazioni di stima dei conteggi. A questo punto, si esamina nuovamente l'intero data stream. Per ogni elemento che soddisfa le proprietà di un heavy hitter si associa il proprio bucket distinto, che contiene, di conseguenza, il conteggio esatto, in modo tale da evitare collisioni costose. Il conteggio degli elementi poco frequenti, invece, è semplicemente salvato in Count-Min sketch. Questa idea è simile al concetto di cache, utilizzata nei processori, che si configura come memoria di dimensioni ridotte che mantiene gli elementi più recentemente acceduti: essa sfrutta, cioè, il principio di località temporale per migliorare il tempo di accesso medio. In genere, infatti, tra gli elementi più recenti sono inclusi quelli che si "presentano" più volte. Si ripete, poi,

che tutti gli algoritmi "hashing-based" comportano errori di stima a causa delle collisioni che si possono verificare: quando i conteggi di due elementi distinti sono mappati nello stesso bucket, essi influenzano a vicenda la loro stima. Infine, gli algoritmi utilizzati usano funzioni di hash casuali, quindi le collisioni sono controllate solo probabilisticamente. Per questo motivo agli elementi classificati come heavy hitter si associa un proprio bucket, distinto dalla matrice usata per il conteggio di tutti gli altri elementi. Questa probabilità può essere influenzata dalla scelta delle funzioni hash e dalle dimensioni della struttura dati. Tuttavia, anche con una gestione probabilistica ottimale, non è possibile garantire che tutte le collisioni vengano evitate, ma si può solo minimizzare la loro frequenza e l'impatto sull'accuratezza delle stime fornite dall'algoritmo. Ricapitolando, in una prima lettura il learning algorithm estrae (esaminando il sottoinsieme $a^{'}$) le proprietà caratteristiche dello stream (come pattern tra collisioni e heavy hitter). A questo punto si mantiene un insieme di B_r bucket e un Count-Min sketch di dimensione $B - B_r = dw$. Effettuando, quindi, una passata completa del flusso, il conteggio di ogni elemento classificato come heavy hitter viene salvato (e quindi aggiornato) in uno dei B_r bucket, mentre per i restanti elementi si mantiene la stima in Count-Min sketch. Per rispondere, perciò, ad una point query, si adotta una procedura simile al caso, già presentato, in cui si fa uso solo dello sketch: ai fini di calcolare il conteggio a_i dell'item i, l'algoritmo prima verifica se i è heavy hitter; in tal caso il dato cercato è reperito nel corrispondente bucket e, quindi, viene restituito il conteggio esatto (vale cioè che $a_i = \hat{a}_i$). Altrimenti, il conteggio si configura pari al minimo delle d stime in Count-Min sketch. Viene riportata, cioè, una stima (sovrastima) del conteggio effettivo.
Questa idea è simile alla struttura *Skimmed Sketch*, proposta in [19], la quale comporta la rimozione degli elementi ad alta frequenza (che superano, cioè, una certa soglia), nella fase principale di elaborazione dei dati, con l'aiuto di un heap. Esso viene utilizzato nell'ottimizzazione di operazioni nei database, specialmente per la stima del join size di due stream, ma anche per multi-join query.
In conclusione, quindi, si utilizza un algoritmo di machine learning in grado di identificare le proprietà degli heavy hitter, in modo tale da salvare i conteggi esatti di questi ultimi in bucket distinti e separati, evitando che l'errore di stima sia elevato. Questo permette un miglioramento logaritmico delle prestazioni (accuratezza della stima) rispetto al solo utilizzo di Count-Min sketch (la dimostrazione è riportata in 5.1.1).

5.1.1 Analisi

In questa sezione si riportano le dimostrazioni e i lemmi, descritti in [24], che mostrano come l'utilizzo coadiuvato di Count-Min sketch con un algoritmo di machine learning, per l'identificazione degli heavy hitter, migliori le prestazioni, in termini di accuratezza nel conteggio degli item, della prima struttura dati. Si riprendono, per prima cosa, le limitazioni (inferiori e superiori) nell'errore di stima dei conteggi usando Count-Min sketch dimostrate in 4.2. A questo punto, si mostra come, affiancando a tale struttura un

algoritmo predittivo di machine learning, l'errore nella stima dei conteggi migliora di un fattore logaritmico; se ideale, si dimostra, inoltre, che esso raggiunge lo stesso bound asintotico di un Count-Min sketch in cui le funzioni di hash sono ottimizzate e, dunque, non è possibile ottenere un ulteriore miglioramento nelle prestazioni di stima (sotto l'ipotesi che $n/w' > e^{4.2}$, cioè per flussi di dimensioni molto maggiori a quelle dello sketch). Viceversa, se l'algoritmo ammette errori di predizioni, si mostra che questa tecnica permette di mantenere lo stesso limite del caso ideale, purchè l'errore di predizione sia $\delta = O(1/\ln n)$.

- Innanzitutto, si ricorda che l'errore medio nella stima dei conteggi degli elementi di un data stream, mediante l'utilizzo di Count-Min sketch di dimensioni $d\frac{w}{d}$, è limitato inferiormente e superiormente. I due limiti sono i seguenti:

  - $\mathbb{E}[Err(A, \hat{A}_{CM})] \in \Omega\left(\frac{\ln n}{w \ln d}\right)$
  - $\mathbb{E}[Err(A, \hat{A}_{CM})] \in O\left(\frac{d \ln n \ln \frac{dn}{w}^{\frac{d+2}{d-1}}}{w}\right)$

  A questo punto, si introduce il seguente teorema, che permette di dimostrare come, affiancando a Count-Min sketch, di dimensione $d\frac{w'-B_r}{d}$ un algoritmo di machine learning predittivo, l'errore atteso dell'accuratezza della stima migliore di un fattore logaritmico. Per gli heavy hitter si mantengono B_r bucket distinti. Si ricorda che il seguente risultato vale se gli item sono distribuiti secondo una distribuzione di Zipf.

  Teorema 3: "L'errore atteso ottimale di un Count-Min sketch affiancato ad un algoritmo predittivo di machine learning di parametri (wd, B_r) è al più pari a $\frac{(\ln n/B_r + 0.58)^2}{wd - B_r} \leq \Theta\left(\frac{\ln (n/w)^2}{w}\right)$. Se $B_r = \Theta(w) = \Theta(n)$, allora $\mathbb{E}[Err(C, \hat{C}_{L-CM})] = O\left(\frac{1}{n}\right)$".

  Si riporta, quindi, la dimostrazione di [24]:

  Dimostrazione. Si ricorda, per prima cosa, che un heavy hitter è un elemento il cui conteggio è almeno pari a $\phi||a||_1$ e si osserva che nel caso particolare di una distribuzione di Zipf il numero massimo di heavy hitter è $\frac{1}{\phi \ln n}$, quindi $B_r \in \Theta(\phi \ln n)^{-1}$.

  A questo punto si calcola l'errore medio di Count-Min sketch così modificato

  $$\mathbb{E}[Err(C, \hat{C}_{L-CM})] = \sum_{j=1}^{n} \mathbb{E}[e_j] \cdot c_j$$

  I B_r elementi più frequenti sono salvati ciascuno in un bucket distinto, quindi $e_j = 0$ $\forall j$ heavy hitter. Da ciò ne consegue che

  $$= \sum_{j \notin heavy\ hitter} \mathbb{E}[e_j] < \frac{(\ln(n/B_r) + 0.58)^2}{w' - B_r}$$

  Quest'ultima disequazione segue dalla seguente osservazione 1, considerando $d = 1$ (vale infatti solo per una singola funzione di hash, 4.2). □

Come già enunciato nei soprastanti teoremi, il miglioramento è massimizzato quando w è dello stesso ordine di n, vale a dire $w = \Theta(n)$.

- In presenza di un algoritmo perfetto, tale cioè da identificare con esattezza gli elementi heavy hitter, l'errore atteso si attesta essere pari a $\mathbb{E}[Err(C, \hat{C}_{L-CM})] \in \Theta\left(\frac{(\ln n/B)^2}{B}\right)$. Questo bound coincide con quello ottenuto in un Count-Min sketch ideale, tale cioè da ottimizzare le d funzioni di hash usate. Tramite il seguente teorema enunciato, si dimostra che il limite inferiore dell'errore atteso di stima per uno sketch con funzioni di hash ottimizzate è pari a $\Omega\left(\frac{\ln n/w^2}{w}\right)$([24]):

  Teorema 4: "Se $n/w^{'} \geq e^{4.2}$, allora l'errore di stima di un qualsiasi sketch che mappa una distribuzione Zipfiana di n item in $w^{'}$ bucket è pari a $\Omega\left(\frac{\left(\ln \frac{n}{w^{'}}\right)^2}{w^{'}}\right)$".

  Dimostrazione. La dimostrazione è costruttiva: essa considera un Count-Min sketch con una singola funzione di hash $h^* : 1, \ldots, n \rightarrow 0, \ldots, w^{'} - 1$ costruita a partire da d funzioni di hash, tali per cui una di esse è pari $h_j : 0, \ldots, n \rightarrow [w^{'}/d]$. L'errore nella stima di questa nuova struttura è minore o uguale al corrispondente errore dello sketch originario (con d righe). Definendo $C'[i]$ come il bucket che memorizza il valore del conteggio dell'item i restituito dallo sketch originario, si ricava che $|C'[i] : i \in 1, \ldots, n| \leq w^{'}$, dal momento che lo sketch possiede al più $w^{'}$ bucket. Vengono, quindi, considerati solo i bucket che riportano le stime effettive dei conteggi. Si definisce quindi $h^*(i) = (j^*, h_{j^*}(i))$, $\forall i$ tale per cui $j^* = \text{argmin}_{j \in 1,\ldots,d} C[j, h_j(i)]$. Da questo ne consegue che h^* mappa ogni item in esattamente un unico bucket, quindi $C'[h^*(i)] \leq C[h^*(i)] = \hat{c}_i$, dove $\hat{c}_i$ è la stima del conteggio effettivo restituito da Count-Min sketch. Da questo si evince che $C'[h^*(i)] \geq c_i$, quindi $Err(C, \hat{C}_{h^*}) \leq Err(C, \hat{C}_{CM})$.

  Una volta dimostrato questo, per la validità del seguente teorema ([24]): "Se $n/w^{'} \geq e^{4.2}$, allora l'errore di stima di una qualsiasi funzione di hash che mappa una distribuzione Zipfiana di n item in $w^{'}$ bucket è pari a $\Omega\left(\frac{\ln n/w^{'2}}{w^{'}}\right)$",

  si ricava che la stima dell'errore di h^* è pari a $\Omega\left(\frac{(\ln n/w^{'})^2}{w^{'}}\right)$, da cui ne consegue che l'errore di Count-Min sketch con d funzioni di hash è lo stesso. □

- Infine, si dimostra che tale risultato continua a rimanere valido anche se l'oracolo ammette errori nella predizione con una probabilità δ, sotto il vincolo che $\delta = O(\frac{1}{\ln n})$. La dimostrazione presentata in [24] è basata sul seguente teorema, di cui poi si riporta la dimostrazione ([24]):

  "In un Count-Min sketch ottimo di parametri $(w^{'}, B_r)$ con un algoritmo predittivo soggetto a rumori, HH_δ, $\mathbb{E}[Err(A, \hat{A}_{(L-CM,\delta)})] \in O\left(\frac{\delta^2 B_r + (\ln(n/B_r))^2}{w^{'} - B_r}\right)$".

Dimostrazione. L'osservazione chiave consiste nell'osservare che ogni item, può essere classificato in modo scorretto con una probabilità δ. Se l'elemento non rientra invece in questa categoria, allora si ottiene che l'errore medio di stima è pari a

$$\mathbb{E}[e_j] \leq \delta \cdot \left(\frac{\ln B_r + 1}{w' - B_r}\right) + \left(\frac{\ln(n/B_r) + 1}{w' - B_r}\right) = O\left(\frac{\delta \ln B_r + \ln(n/B_r)}{w' - B_r}\right).$$

In questa disequazione, il primo termine dell'addizione costituisce il contributo atteso di un falso negativo (cioè di un heavy hitter non classificato come tale), mentre il secondo termine denota il contributo di un elemento non heavy hitter (vale a dire falso positivo).

A questo punto, si sfrutta questo risultato per calcolare l'errore atteso tra la frequenza stimata e quella effettiva:

$$\begin{aligned}
\mathbb{E}[Err(A, \hat{A}_{(L-CM,\delta)})] &\approx \sum_{j \in 1,\dots,n]} \mathbb{E}[e_j] \cdot c_j \\
&< \sum_{j \leq B_r} \mathbb{E}[e_j] \cdot c_j + \sum_{j > B_r} \mathbb{E}[e_j] \cdot c_j \\
&\leq O(\delta \cdot B_r + \ln(n/B_r)) \cdot O\left(\frac{\delta \ln B_r + \ln(n/B_r)}{w' - B_r}\right) \\
&\leq O\left(\frac{\delta^2 (\ln B_r)^2 + (\ln(n/B_r))^2}{w' - B_r}\right)
\end{aligned}$$

□

In conclusione, l'utilizzo coadiuvato di Count-Min sketch con un algoritmo predittivo, che permette di riconoscere le proprietà di un data stream su un suo sottoinsieme, permette di migliorare le prestazioni in termini di accuratezza nella stima di un fattore logaritmico. Infatti, sia nel caso ideale che nel caso soggetto a rumore, le prestazioni dello sketch migliorano di un fattore logaritmico, circa pari a $\Theta\left(\frac{(\log n/w')^2}{w'}\right) = \Theta\left(\frac{(\log n/wd)^2}{wd}\right)$.

5.2 Miglioramento di Count-Min sketch tramite l'utilizzo di un oracolo o di un predittore

In questa sezione si studiano le prestazioni di Count-Min sketch nella risposta a point query qualora venga affiancato ad esso o un oracolo, che determina con esattezza gli elementi heavy hitter, oppure un predittore, in cui l'individuazione degli elementi più frequenti è soggetta ad errore. Mediante un'analisi grafica, si confrontano qualitativamente, quindi, le prestazioni, in termini di errore medio e match tra conteggi effettivi e stime nei tre casi considerati. Viene dimostrato, quindi, che l'utilizzo di un oracolo o di un predittore permette di diminuire l'errore medio di stima.

Nel primo caso, l'oracolo è un algoritmo (nel codice implementato come una classe, `Oracle`) in grado di distinguere con esattezza tra elementi heavy hitter e non heavy hitter. Come è possibile osservare dall'implementazione della classe citata nella repository GitHub, viene letto innanzitutto un flusso campione. Utilizzando, dunque, l'algoritmo di Misra-Gries si determinano i B_r elementi più frequenti. Data la percentuale ϕ, come suggerito nella dimostrazione nella sezione precedente, si sceglie B_r in modo tale che approssimi la quantità $\frac{1}{\phi \ln n}$, con ϕ da determinare ed n, numero di elementi distinti, noto. Si presuppone che il data stream elaborato per l'identificazione degli heavy hitter sia noto a priori. Per semplicità, viene quindi letto una seconda volta lo stesso flusso, considerando un elemento alla volta. Il conteggio di un elemento classificato come heavy hitter viene salvato in un bucket distinto. Nel caso di elementi meno frequenti, invece, si memorizza il corrispondente conteggio in un Count-Min sketch, per cui, alla fine, è possibile per questi ultimi ottenere solo una stima della sua effettiva frequenza.

Nel caso del predittore, invece, nella prima lettura dello stream viene utilizzato un Count-Min sketch di dimensioni ridotte (cioè molto inferiori a quelle che caratterizzano lo sketch principale, vale a dire quello in cui vengono salvati i conteggi degli elementi non heavy hitter) per memorizzare approssimativamente le frequenze dei vari elementi dello sketch. A questo punto, in base ai conteggi approssimati ottenuti nella prima lettura, il predittore è in grado di identificare (in modo non perfetto) gli elementi heavy hitter. In particolare, si sceglie una quantità fissata (per esempio $\frac{1}{\phi \ln n}$, in modo simile a quanto visto nel caso precedente), che coincide con i B_r elementi più frequenti. Si osserva che, a causa delle possibili collisioni tra elementi maggiormente frequenti ed item più sporadici, è ammessa la presenza di falsi positivi (che non si possono verificare, invece, nel caso dell'oracolo). Successivamente, i corrispondenti conteggi di questi elementi vengono salvati in bucket distinti. Il conteggio degli elementi non heavy hitter è, al contrario, ordinariamente salvato in Count-Min sketch. Per ulteriori dettagli implementativi, si consiglia di leggere il file `Predictor`, reperibile nella repository GitHub precedentemente specificata.

Si osserva, dunque, che non vengono utilizzate tecniche di machine learning, quanto algoritmi più primitivi, che si fondano sul concetto di predizione per ottenere una stima più accurata nei conteggi degli elementi di un data stream, ma che non sfruttano eventuali dipendenze dai dati del flusso. Inoltre, avendo utilizzato, per semplicità, in entrambi i casi lo stesso flusso sia per la determinazione degli heavy hitter che per il conteggio (per lo più stimato) di ogni item, gli algoritmi proposti non si configurano propriamente come online algorithm ([31]), di cui Count-Min sketch è, per l'appunto, esempio. Tuttavia, l'accuratezza nella stima, specialmente nel caso dell'oracolo, migliora notevolmente. La complessità spaziale rimane pressoché inalterata. Si può considerare, infatti, che lo spazio riservato ai B_r bucket e al Count-Min sketch di dimensioni ridotte se si utilizza il predittore, sia $O(1)$ rispetto alla taglia della matrice per lo sketch.

In tutti e due i casi, poi, per rispondere ad una point query si adotta la procedura seguita

nella sezione 5.1. Di conseguenza, nel momento in cui viene posta una query per il conteggio di un elemento, si verifica se esso è un heavy hitter. In tal caso, il corrispondente conteggio (esatto) è reperito nel corrispettivo bucket tra i B_r disponibili. In caso negativo, invece, il valore cercato coincide con il minimo dei d conteggi contenuti in Count-Min sketch. In quest'ultimo caso si ottiene quindi solo una stima.

5.2.1 Oracolo

In questa sezione si analizza, più nel dettaglio, il funzionamento dell'oracolo. Quest'ultimo si configura come un algoritmo in grado di identificare gli elementi che soddisfano la proprietà di essere heavy hitter. Il conteggio di ciascuno di essi viene in seguito memorizzato in un bucket distinto, anziché in Count-Min sketch, in modo tale da evitare collisioni più importanti tra elementi a maggiore frequenza ed elementi caratterizzati da un conteggio molto basso. Mediante analisi sperimentali, si è evinto che il numero di elementi da considerare come heavy hitter, in modo tale da ottenere un buon trade-off tra spazio di memoria e accuratezza, si attesta intorno al 2% delle dimensioni totali del flusso. Questo significa che se la distribuzione è costituita da 1000 elementi, contando anche eventuali duplicati, 20 di questi verranno considerati come heavy hitter. Una stima più esatta, come suggerito in [24], consiste nel considerare $B_r = \frac{1}{\phi \ln n}$. In questo caso, tuttavia, sarebbe necessaria avere una conoscenza a priori del flusso, in quanto dovrebbe essere noto il numero di elementi distinti n.
Per identificare gli item più frequenti si è utilizzato l'algoritmo di Misra-Gries ([33]), illustrato nel seguito.

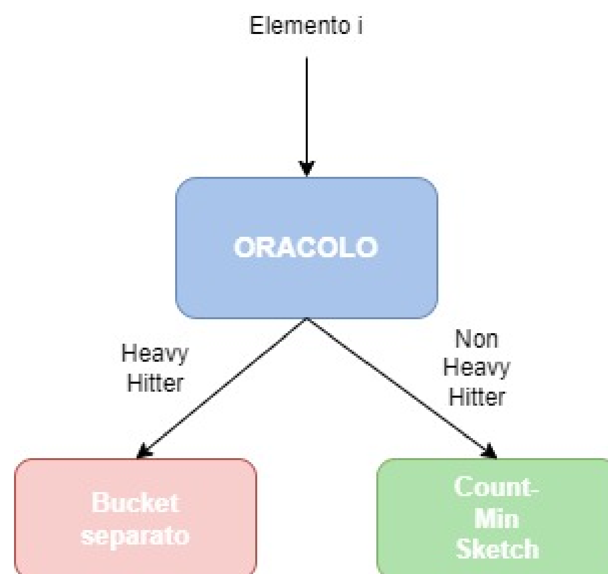


Figura 5.1: Schematizzazione del funzionamento oracolo

Come già descritto, l'oracolo (per maggiori dettagli consultare il file `Oracle` nella repository GitHub) si configura come un algoritmo che permette di identificare con esattezza gli elementi che soddisfano la proprietà di essere heavy hitter (i B_r item più frequenti, per

la precisione). Per operare questa selezione, si è sfruttato l'algoritmo deterministico di Misra-Gries, introdotto dagli omonimi autori in [33]. Esso permette di determinare, in una sola lettura del data stream, quali elementi sono heavy hitter. Questi ultimi sono poi salvati in una struttura dati (come suggerito in [33] quella più adeguata è un AVL Tree, dal momento che evita un'implementazione randomizzata di un algoritmo deterministico, come si otterrebbe utilizzando invece un Hash Map, [11], che porterebbe a inesattezze nell'analisi della complessità temporale dell'algoritmo) ausiliaria di dimensioni pari a $\Theta(B_r)$, tale per cui ogni operazione di aggiornamento in essa richiede un tempo pari a $O(\log B_r)$. L'algoritmo mostrato è descritto dal seguente pseudocodice:

Algorithm 1 Algoritmo di Misra-Gries

```
 1: procedure FIND HEAVY HITTER
 2:     Inizializza ciascuno dei B_r bucket a (null, 0)
 3:     for x ∈ stream do
 4:         if x ∈ bucket b then
 5:             b.counter ← b.counter + 1
 6:         else if ∃ bucket b′ : b′.counter = 0 then
 7:             b′.element ← x
 8:             b′.counter ← 1
 9:         else
10:             for b′ ∈ bucket do
11:                 b′.counter ← b′.counter − 1
12:             end for
13:         end if
14:     end for
15: end procedure
```

L'algoritmo quindi, a seguito di un'iniziale instanziazione di ogni bucket al valore di default (pari a `(null, 0)`, dove il primo termine indica l'item, mentre il secondo il conteggio), esamina ogni elemento del flusso: se riscontra che un bucket contiene già il conteggio per questo elemento, allora incrementa di un'unità il corrispondente contatore. Altrimenti, se esiste almeno un bucket inizializzato al valore di default, quest'ultimo viene modificato al valore (item, 1). Se invece tutti i bucket contengono il conteggio di un elemento, allora, per ciascuno di questi il contatore viene decrementato di un'unità, in modo che, se un elemento dovesse presentare come nuovo conteggio 0, esso viene sostituito con il successivo item nel flusso. Come nota conclusiva, si nota che questo algoritmo presuppone che ad ogni iterazione il conteggio venga aumentato di un'unità (vale a dire $c_t = 1$) e, di conseguenza, viene considerato solo il caso di flusso di tipo cash register, cioè ($c_t > 0$).

Adottando questo approccio, da un punto di vista asintotico lo spazio aggiuntivo utilizzato si attesta essere pari a $O(B_r)$ (infatti si necessita di una mappa di B_r elementi per determinare gli item più frequenti e un'altra mappa per memorizzare i conteggi di questi ultimi), lineare, quindi, nel numero di elementi classificati come heavy hitter. Lo spazio complessivo è dunque uguale a $O\left(\frac{e}{\varepsilon} \ln \frac{1}{\delta} + B_r\right)$. Si presuppone, come appunto, che le

dimensioni di Count-Min sketch siano pari a $\Theta(B - B_r)$. In altre parole si riducono le dimensioni dello sketch affiancato all'oracolo in modo che lo spazio complessivo occupato sia uguale a quello richiesto dall'utilizzo di Count-Min sketch, con B bucket.

Per quanto riguarda, invece, la complessità temporale, si osserva innanzitutto che l'oracolo richiede una prima lettura del data stream per determinare quali elementi si configurano come heavy hitter. A questo punto, è necessario "scorrere" nuovamente il flusso in modo tale da salvare i conteggi o nella mappa o nel Count-min sketch. Dunque, il costo di lettura raddoppia rispetto al caso non ottimizzato. Più nel dettaglio, considerando in un primo momento solo l'algoritmo di Misra-Gries si possono fare le seguenti considerazioni: ogni elemento nel flusso viene processato in tempo costante se l'elemento è già presente nei candidati o se c'è spazio per un nuovo candidato. Se, invece, l'elemento non è presente nella mappa e non c'è spazio, è necessario ridurre i contatori degli elementi memorizzati nella mappa. Questo richiede tempo $O(B_r)$. In totale, per un flusso di N elementi, la complessità temporale di Misra-Gries si attesta a $O(NB_r)$. Dal momento che B_r è una costante determinata a priori, è possibile considerare tale parametro costante, rendendo la complessità temporale uguale a $O(N)$. Nella fase, poi, di conteggio degli item, se l'elemento è un heavy hitter l'operazione considerata richiede un accesso ad un singolo bucket, operazione eseguita in un tempo costante. Se invece l'elemento non fa parte di questa categoria, il conteggio è salvato in d bucket distinti. La complessità totale per leggere l'intero stream si attesta a $O(\sum_{i \in heavy\ hitter} a_i + (N - \sum_{i \in heavy\ hitter} a_i)d)$. Il tempo di risposta ad una point query è, invece, pari a $\Theta(1 + d) = \Theta(d = \log(1/\delta))$.

5.2.2 Predittore

In questa sezione viene analizzata la strategia che affianca a Count-Min sketch un predittore. Quest'ultimo si configura come un algoritmo in grado di identificare, in modo approssimato, gli elementi che soddisfano la proprietà di essere heavy hitter. Per fare questo, il predittore scorre il data stream e salva i conteggi di ciascun item in un Count-Min sketch di dimensioni ridotte, cioè molto minori rispetto a quelle dello sketch "principale". Vengono classificati come heavy hitter i B_r elementi più frequenti, vale a dire gli item che presentano i conteggi approssimati più elevati. Per implementare questo, viene mantenuto un dizionario di triplette <elemento, conteggio stimato, conteggio effettivo>, in cui memorizzare i B_r heavy hitter. In questo modo viene ridotto il numero di letture del flusso complessive, a scapito, tuttavia, di una minore atomicità del predittore. Inizialmente le varie entry sono inizializzate al valore di default, cioè `(null, 0, 0)`. Si scorre, nuovamente, il flusso: dato l'item i, se esiste almeno una posizione non inizializzata nella mappa allora l'elemento viene salvato con il valore `(i, CMS.estimate(i), 1)`. Se, invece, è già presente una entry associata all'elemento considerato, si aggiorna il conteggio effettivo di un'unità. Nel caso in cui non figuri tra gli elementi salvati, si confronta il conteggio stimato con la stima minima dei bucket. Se il primo si configura essere maggiore del secondo, allora viene

salvato il valore corrispondente, mentre l'altro è salvato in Count-Min sketch `CMS2`, con un aggiornamento del conteggio di c_t. Altrimenti, l'elemento è salvato direttamente in Count-Min sketch, in quanto sicuramente non è uno degli elementi più frequenti. Dopo aver letto tutto il flusso, nella mappa sono mantenuti i B_r heavy hitter. Per maggiore chiarezza, si riporta lo pseudocodice della procedura appena descritta.

Algorithm 2 Algoritmo per la determinazione degli heavy hitter

```
 1: procedure ALGORITMO PER LA DETERMINAZIONE DEGLI HEAVY HITTER
 2:     Inizializza ciascuno dei B_r bucket a (null, 0, 0)
 3:     for x ∈ stream do
 4:         if ∃bucket b' : b'.key = null then
 5:             b'.element ← x
 6:             b'.estimate ← CMS.estimate(x)
 7:             b'.count ← 1
 8:         else
 9:             c ← getValue(x)
10:             if c = null then
11:                 b' ← argmin bucket.estimate
12:                 if CMS.estimate(x) > min b'.estimate then
13:                     CMS2.update(x, b'.count)
14:                     b' ← (x, CMS.estimate(x), 1)
15:                 else
16:                     CMS2.update(x, 1)
17:                 end if
18:             else
19:                 b'.count ← b'.count + 1
20:             end if
21:         end if
22:     end for
23: end procedure
```

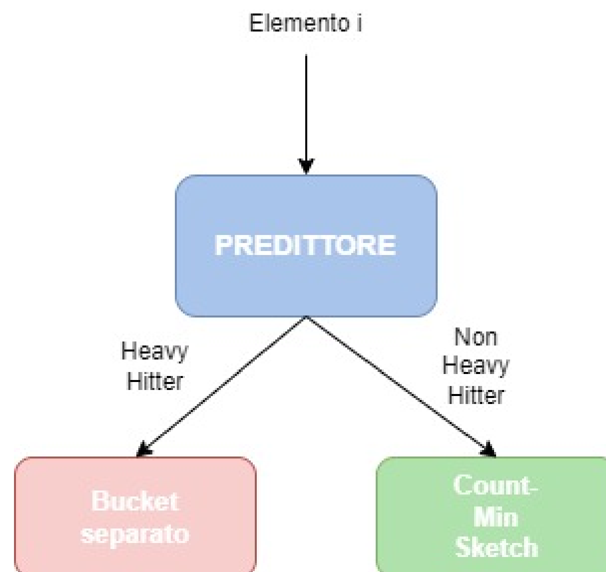


Figura 5.2: Schematizzazione del funzionamento del predittore

Procedure più avanzate per la stima dei conteggi, sempre probabilistiche e approssimate, sono invece presentate nell'articolo [6]. In esso gli autori descrivono, per esempio, l'algoritmo *Sampling*, il quale consiste nel considerare un campione casuale uniforme di elementi del flusso, organizzato secondo una lista, in cui ad ogni elemento distinto è associato un contatore. In base a questo algoritmo, se il campione contiene x elementi, allora per assicurarsi che un elemento di frequenza a_k sia incluso nel campione, si necessita che la probabilità di essere compreso in esso, $\frac{x}{n}$, sia $\frac{x}{n} > O((\log k))/a_k)$. Questo permette di includere nel campione i k elementi più frequenti, che possono quindi essere considerati come heavy hitter. Un ulteriore miglioramento, è stato proposto in [14]: in esso gli heavy hitter vengono tracciati dinamicamente, basandosi sull'idea di *Group Testing*, che consiste nel progettare un numero di test che raggruppano insiemi di m item in modo tale da determinare, a partire da essi, i B_r elementi più frequenti ([14]).

Si denominano $\varepsilon^{'}$ e $\delta^{'}$ i parametri di Count-Min sketch usato dal predittore. Dunque, per quanto riguarda la complessità spaziale di questa soluzione, l'aggiunta di un predittore presuppone l'utilizzo di uno spazio aggiuntivo di memoria pari a 160 "celle" (determinato sperimentalmente, come giusto compromesso tra memoria ed accuratezza, ponendo $\varepsilon^{'} = 0.1$ e $\delta^{'} = 0.01$) per salvare il Count-Min sketch di dimensioni minori, più una mappa per salvare i conteggi degli heavy hitter. Se quest'ultima struttura dati viene implementata in modo da essere sufficientemente piccola rispetto alle dimensioni del flusso, allora il suo contributo può essere considerato costante. Quindi la complessità spaziale si attesta ad essere pari a $\Theta\left(\frac{e}{\varepsilon}\log\frac{1}{\delta} + \frac{e}{\varepsilon'}\log\frac{1}{\delta'}\right)$.

Nel caso dello studio della complessità temporale, invece, le operazioni aggiuntive consistono nel salvataggio dei conteggi approssimati di ogni elemento, la cui complessità asintotica si attesta pari a $\Theta\left(N\log\frac{1}{\delta'}\right)$, che comporta una prima lettura dell'intero flusso. Successivamente, viene effettuata una seconda passata del data stream, in cui vengono determinati gli elementi heavy hitter. Per ogni item, si controlla innanzitutto se esiste una entry corrispondente nella mappa che memorizza gli item più frequenti. Se il numero di elementi distinti è minore di B_r vengono effettuati al più $O(N)$ accessi. Se l'elemento considerato non è presente in essa, allora si legge il corrispondente conteggio nello sketch di dimensioni ridotte. Un limite superiore al numero di accessi coincide con $O\left(N\log\frac{1}{\delta'}\right)$. Gli elementi non heavy hitter, in tutto $n - B_r$, richiedono, poi, $O\left(\log\frac{1}{\delta}\right)$ accessi in Count-Min sketch. Una stima della complessità temporale in questa operazione è dunque $O\left(N\log\frac{1}{\delta'} + (N - B_r\sum_{j\in heavy\ hitter} a_j)\log\frac{1}{\delta}\right)$. Complessivamente, se $n > B_r$, il tempo nella fase di inizializzazione è limitato da $O\left(N\log\frac{1}{\delta'} + N\log\frac{1}{\delta'} + (N - B_r\sum_{j\in heavy\ hitter} a_j)\log\frac{1}{\delta}\right)$
$= O\left(N\log\frac{1}{\delta'} + (N - B_r\sum_{j\in heavy\ hitter} a_j)\log\frac{1}{\delta}\right)$.
Il procedimento seguito per rispondere ad una point query è lo stesso dell'oracolo. Di conseguenza la complessità temporale si attesta in questo caso essere pari a $\Theta(d) = \Theta(\log(1/\delta))$.

5.2.3 Analisi Sperimentali ed Osservazioni

In questa sezione vengono studiate le prestazioni di Count-Min sketch, in termini di errore nella stima e di accuratezza, quando si associa o un oracolo o un predittore. Più nel dettaglio, si confrontano i risultati ottenuti considerando quattro distribuzioni distinte, composte da 1000, 10000, 100000 e 500000 elementi, studiando per ciascuna di esse gli opportuni parametri con cui instanziare le strutture dati di supporto usate dall'oracolo e dal predittore. Viene, inoltre, fatto in modo che lo spazio utilizzato in ciascuna di queste tre strategie sia circa lo stesso. Vengono, quindi, riportati i grafici al variare dei parametri di ε e δ di Count-Min sketch e delle dimensioni dello stream.

Innanzitutto, per semplicità si considera solo la distribuzione con 1000 elementi. Si confrontano, quindi, i risultati ottenuti, al variare delle dimensioni di Count-Min sketch, sfruttando l'oracolo, il predittore oppure usando il solo sketch. In blu sono riportati i match e mismatch che si ottengono sfruttando solo Count-Min sketch, in rosso ('x') quelli ricavati associando allo sketch l'oracolo e in verde quelli ottenuti aggiungendo invece il predittore. Per maggiore chiarezza, si utilizza per entrambe gli assi la scala logaritmica.

- Distribuzione con 1000 elementi

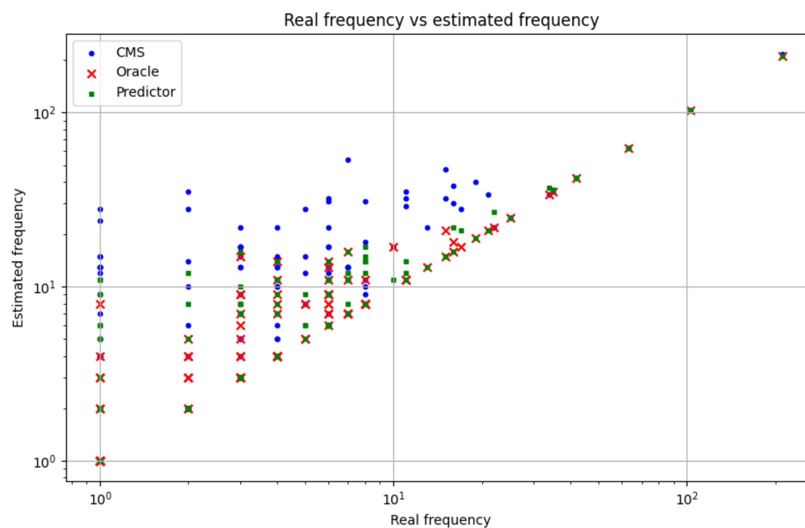


Figura 5.3: $\varepsilon = 0.1, \delta = 0.01$

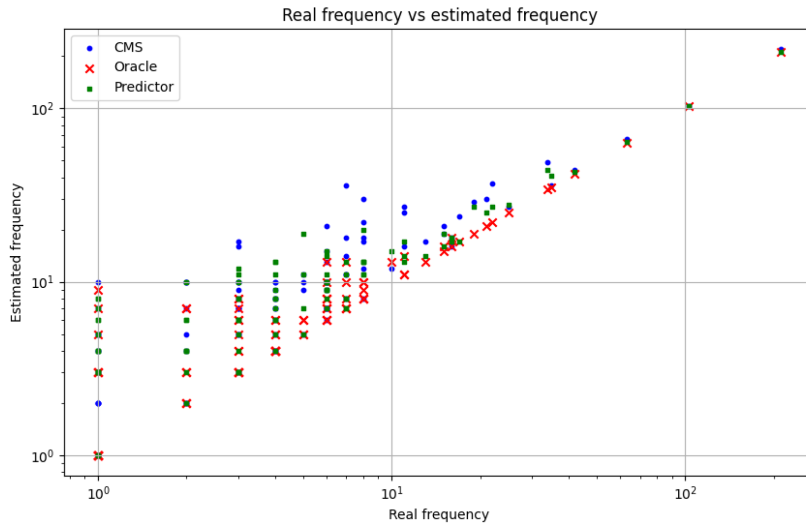


Figura 5.4: $\varepsilon = 0.1, \delta = 0.001$

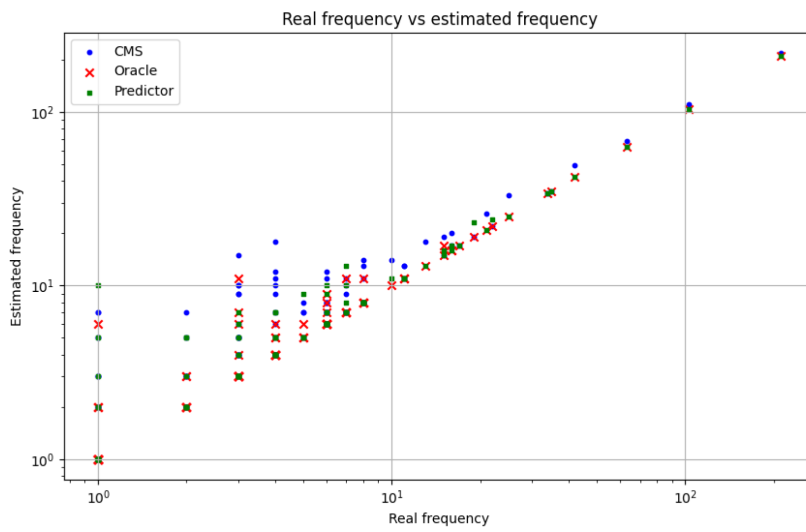


Figura 5.5: $\varepsilon = 0.1, \delta = 10^{-5}$)

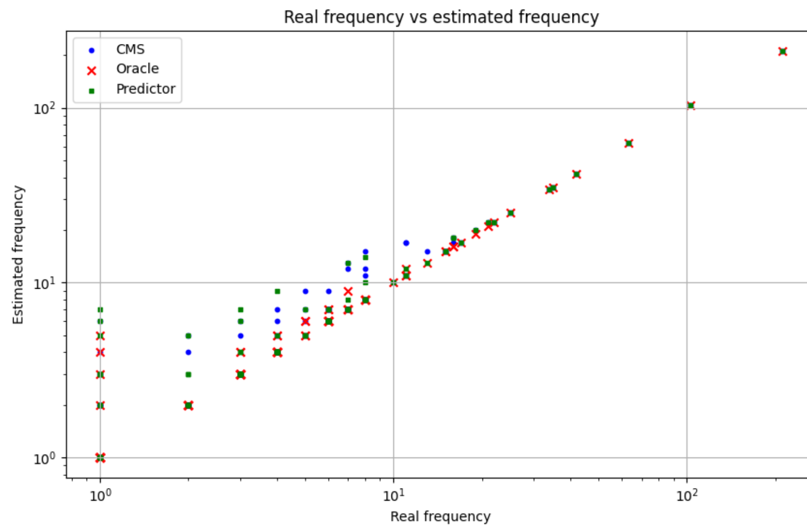


Figura 5.6: $\varepsilon = 0.1, \delta = 10^{-10}$

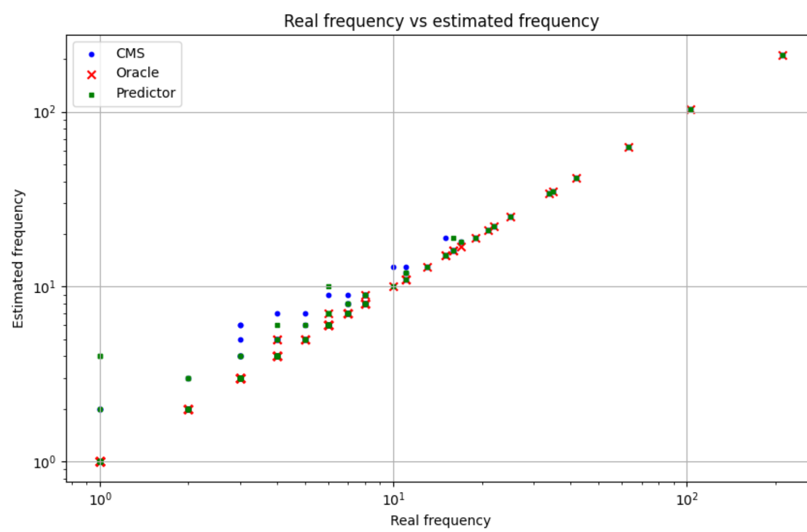


Figura 5.7: $\varepsilon = 0.1, \delta = 10^{-20}$

Fissando, invece, $\delta = 0.1$, si osservano i seguenti comportamenti al variare di ε:

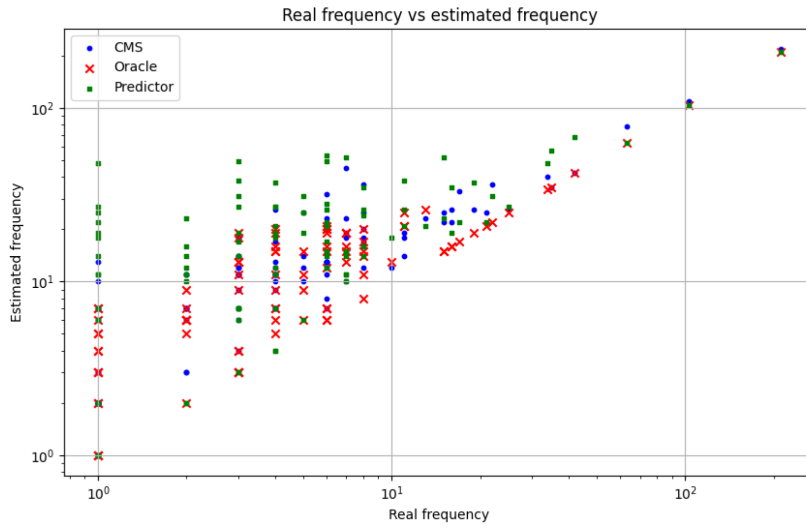


Figura 5.8: $\varepsilon = 0.1, \delta = 0.1$

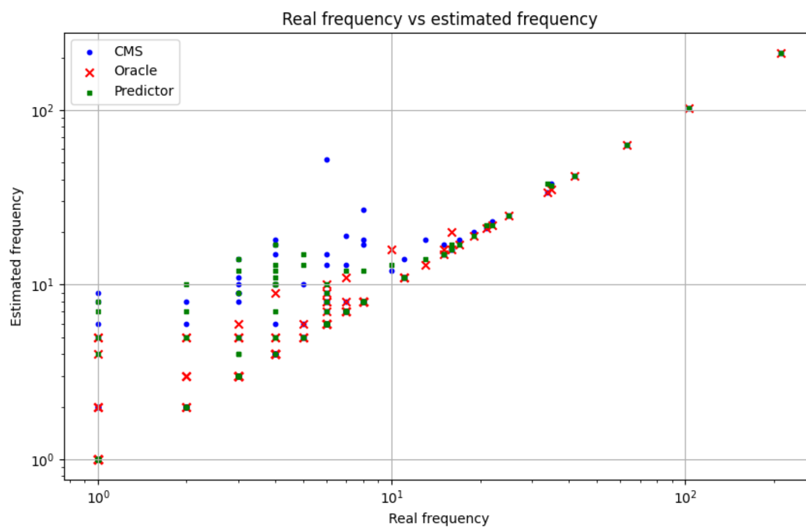


Figura 5.9: $\varepsilon = 0.05, \delta = 0.1$

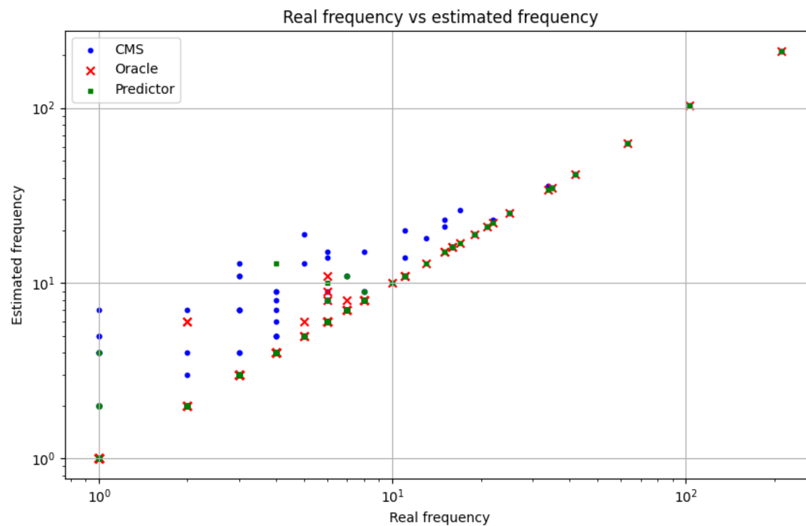


Figura 5.10: $\varepsilon = 0.03, \delta = 0.1$

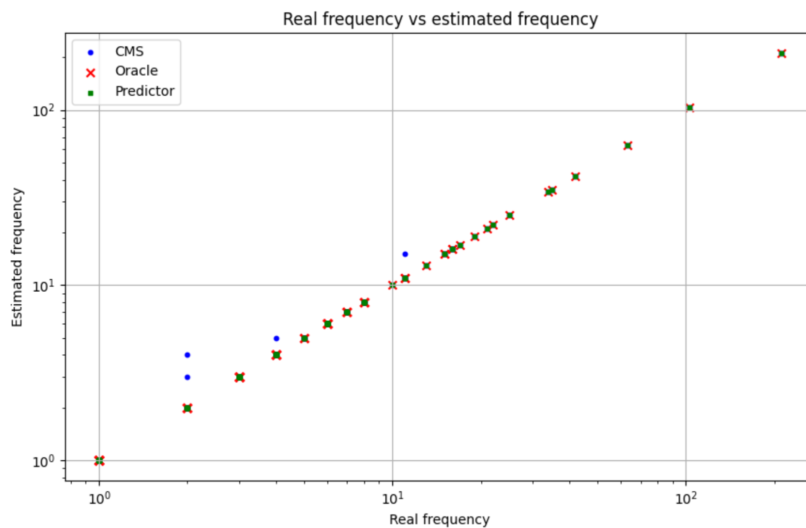


Figura 5.11: $\varepsilon = 0.01, \delta = 0.1$

Dai seguenti grafici si può notare come le migliori prestazioni si ottengono quando a Count-Min sketch viene associato l'oracolo, che permette di distinguere tra elementi heavy hitter (circa il 2 % del numero complessivo di item che costituisce la distribuzione): ciò, infatti, può essere evinto dall'osservazione dei grafici, che mostrano come le 'x' rosse, meglio approssimano l'andamento di una retta, passante per l'origine e con coefficiente unitario, specialmente per sketch di dimensioni maggiori (come quello ottenuto con $\delta = 10^{-20}$, in cui l'andamento è effettivamente lineare). Le prestazioni, sempre in termini di accuratezza nella stima, ottenute coadiuvando il Count-Min sketch con il predittore risultano leggermente peggiori rispetto al caso dell'oracolo, ma comunque le stime approssimano meglio l'andamento di una retta, rispetto al solo utilizzo dello sketch. Utilizzare quindi

l'oracolo, a parità di spazio, si rivela essere la strategia più efficace dal punto di vista dell'accuratezza.

Un'altra possibile analisi consiste nel confrontare l'accuratezza nella stima dei conteggi che si ottiene usando Count-Min sketch e quella ottenuta affiancando a Count-Min sketch un oracolo in modo tale che lo spazio complessivo occupato dallo sketch e dall'oracolo sia esattamente lo stesso di quello usato dal solo sketch. Quindi, se per esempio lo spazio complessivo nel primo caso è di 120, allora anche nel secondo, il numero di celle è lo stesso. Viene, quindi, confrontata la soluzione peggiore (vale a dire quella ottenuta con il solo Count-Min sketch) con quella migliore (cioè quella in cui si affianca l'oracolo). Di seguito vengono riportati i grafici ottenuti per dimensioni differenti.

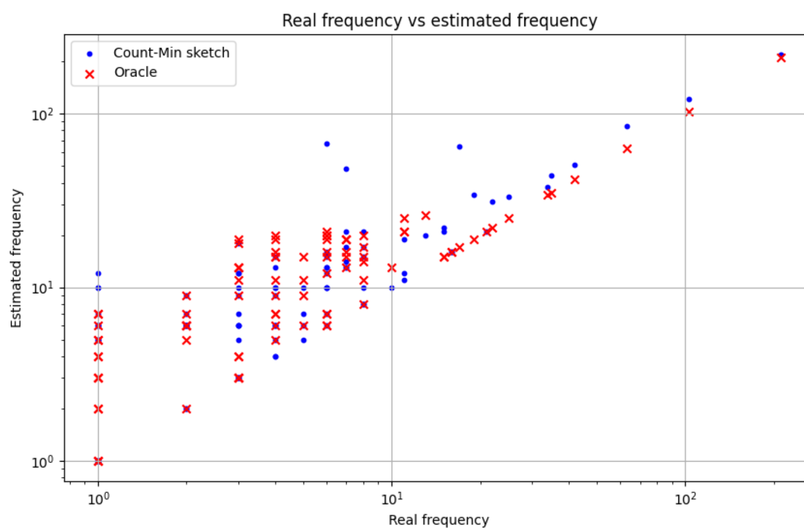


Figura 5.12: Numero complessivo di entry pari a 84

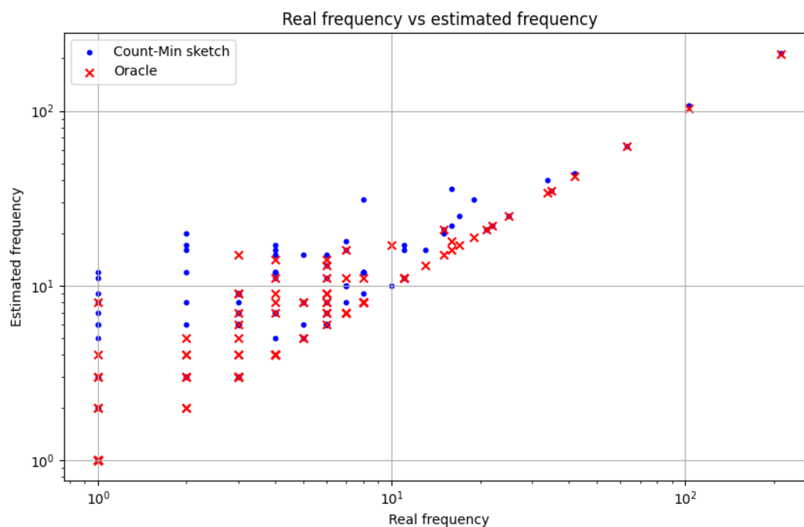


Figura 5.13: Numero complessivo di entry pari a 170

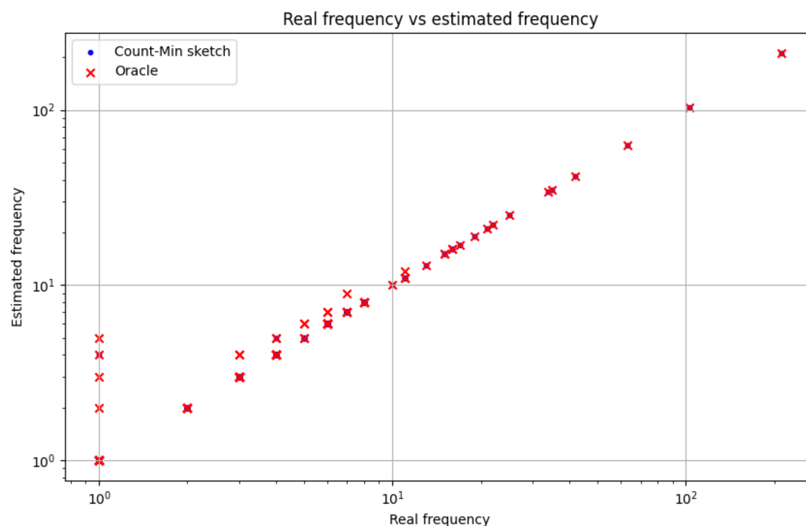


Figura 5.14: Numero complessivo di entry pari a 702

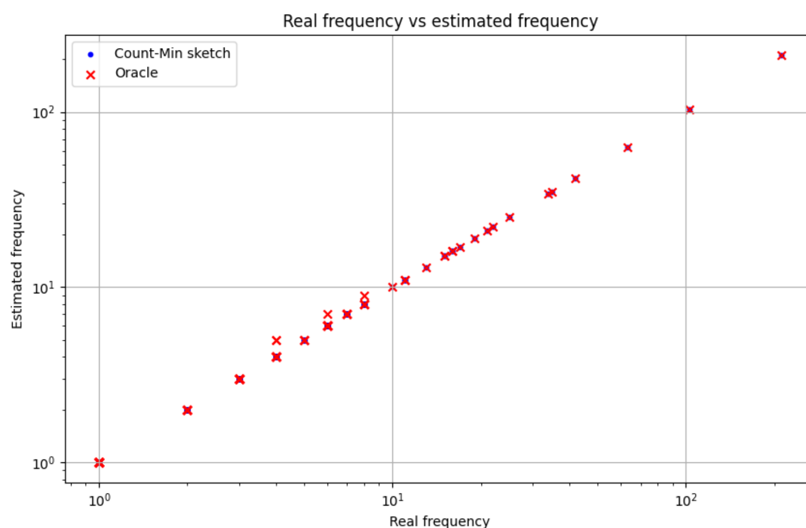


Figura 5.15: Numero complessivo di entry pari a 1346

Si osserva, dunque, che a parità di entry memorizzate, le prestazioni in termini di accuratezza dell'uso coadiuvato di Count-Min sketch con l'oracolo sono migliori rispetto al solo utilizzo dello sketch. Infatti, da un'analisi qualitativa, si nota che il numero di 'x' rosse, che rappresentano le stime ottenuto tramite oracolo, approssimano meglio l'andamento lineare rispetto al numero di pallini blu. Tuttavia, si nota che negli ultimi due grafici le prestazioni del solo Count-Min sketch sono circa le stesse di quelle ottenute con l'oracolo. Questo può essere spiegato col fatto che, nel tentativo di costruire uno sketch che occupasse lo stesso spazio in memoria dello stesso sketch con l'oracolo, si è utilizzato un numero di righe elevato (circa 50): questo diminuisce il numero di possibili collisioni e quindi di mismatch nel conteggio degli elementi. In altre parole, il Count-Min sketch associato all'oracolo presenta un numero differente di righe e colonne rispetto al singolo sketch. Se

nel secondo caso il numero di colonne (o di righe) è maggiore, il pattern di collisioni risulta essere diverso e ciò comporta un errore di stima nei conteggi dei singoli elementi differente.

A questo punto, si esamina graficamente l'errore medio ottenuto o utilizzando il solo Count-Min sketch oppure associando a questo o un oracolo o un predittore. Si analizza questo parametro con diverse distribuzioni, facendo variare uniformemente i valori di ε e δ rispettivamente negli intervalli $[0.01, 0.5]$ e $[0.001, 0.1]$. In altre parole, per ognuno dei due parametri si campionano 100 valori ad intervalli regolari, che poi vengono combinati con i valori assunti dall'altro parametro, per un totale di 10000 campioni (10000 dimensioni differenti di Count-Min sketch), adottando quindi una procedura simile a quella della sezione 4.2. Di seguito, vengono riportati i risultati ricavati nei tre casi. Più precisamente, in blu vengono riportati i risultati ottenuti con Count-Min sketch, in rosso quelli ottenuti usando anche l'oracolo ed in verde quelli del predittore. Le dimensioni vengono riportate in kilobyte e variano, dunque, nell'intervallo [0.07, 7.4]KB

- Distribuzione di Zipf con 1000 elementi

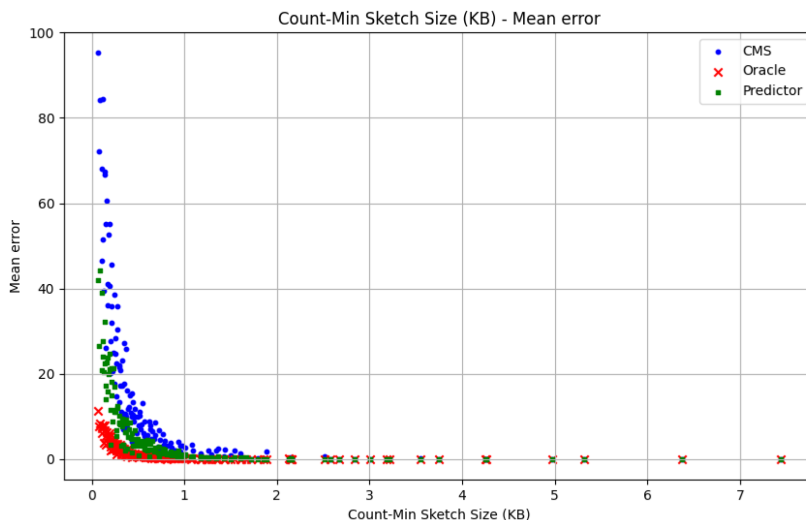


Figura 5.16: Errore medio con una distribuzione di 1000 elementi

- Distribuzione di Zipf con 10000 elementi

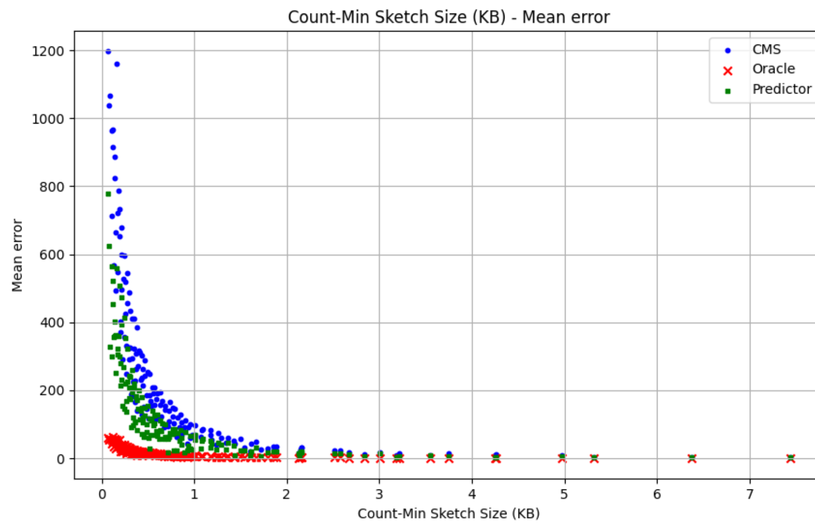


Figura 5.17: Errore medio con una distribuzione di 10000 elementi

- Distribuzione di Zipf con 100000 elementi

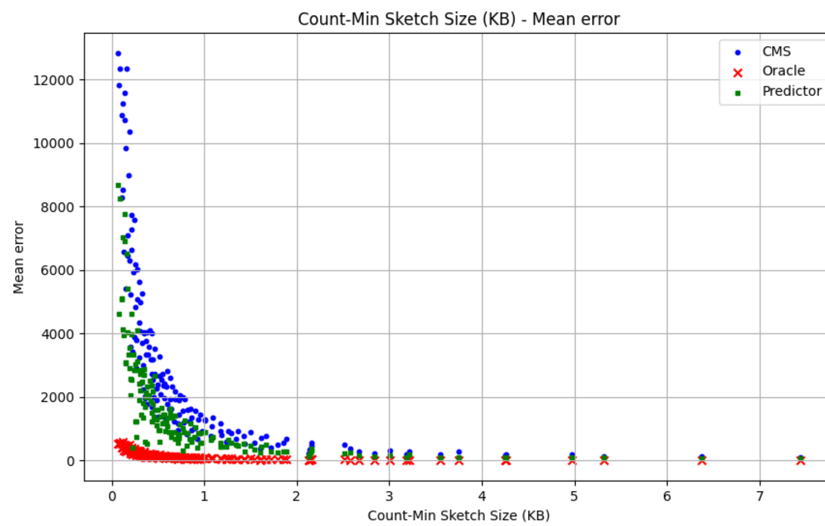


Figura 5.18: Errore medio con una distribuzione di 100000 elementi

- Distribuzione di Zipf con 500000 elementi

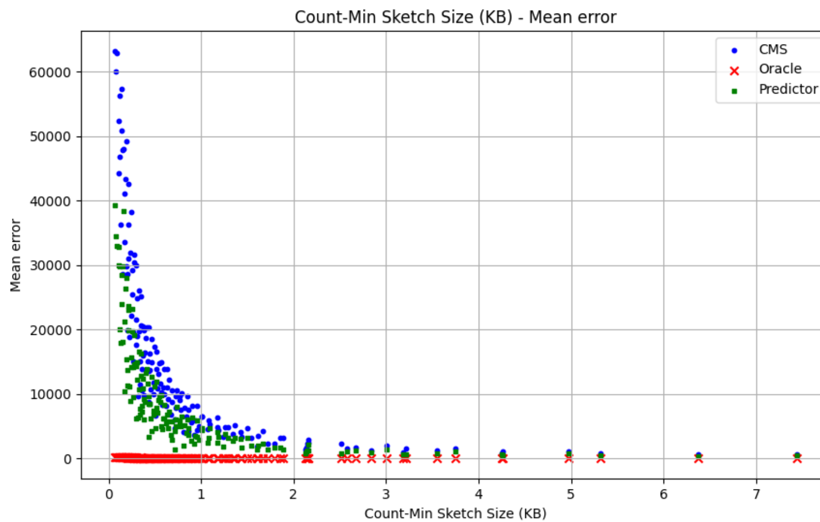


Figura 5.19: Errore medio con una distribuzione di 500000 elementi

Da questi grafici si evince che affiancando a Count-Min sketch un oracolo si ottengono prestazioni migliori in termini di accuratezza nella stima. In quest'ultimo caso, ottime sono le performance per distribuzioni di dimensioni elevate, dove la decrescita è molto "appiattita", apparendo quasi come una retta. Usufruendo invece del predittore, l'errore si mantiene leggermente inferiore rispetto a quello ottenuto utilizzando solo Count-Min sketch. Inoltre, si nota come in tutti i casi esaminati l'errore massimo nella stima dei conteggi utilizzando l'oracolo sia minore rispetto ai corrispondenti valori ottenuti con il predittore (comunque minore del solo utilizzo dello sketch) e con il solo Count-Min sketch. In altre parole, definendo E_O l'errore massimo nella stima dei conteggi associando al Count-Min sketch un oracolo, E_P l'errore massimo ottenuto aggiungendo il predittore, e E_C l'errore nella stima dei conteggi usando unicamente Count-Min sketch, dai dati sperimentali si osserva quindi che $E_O < E_P < E_C$.

Aumentando il numero di elementi da considerare come heavy hitter (e quindi le dimensioni dell'heap/mappa nell'oracolo per mantenere il conteggio di questi ultimi), sicuramente diminuisce l'errore medio, al costo tuttavia di una maggiore occupazione di memoria: per questo motivo il giusto compromesso è stato ottenuto considerando circa il 2% degli elementi come heavy hitter.

Per memorizzare gli heavy hitter nell'oracolo si è usufruito di una mappa, fornita nella libreria `java.util`. Tuttavia, per ottenere un'analisi temporale più corretta, sarebbe opportuno, come suggerito dagli autori in [33], utilizzare un AVL Tree o un heap. Infatti, utilizzando `HashMap`, la complessità attesa nell'operazione di aggiornamento è pari a $O(1)$. Questo valore è sicuramente ottenuto sfruttando un hardware adeguato e una memoria associativa. Invece, in assenza di tale hardware si necessita di una struttura a dizionario dinamica adeguata, quale un heap, per evitare un'implementazione randomizzata di un algoritmo deterministico.

In conclusione, in tale capitolo sono state presentate delle tecniche alternative ai learning algorithm per migliorare l'accuratezza nella risposta stimata a point query di Count-Min sketch. Sono stati descritti, quindi, gli algoritmi di oracolo e predittore che possono essere utilizzati congiuntamente allo sketch. Tramite un'analisi grafica, si sono confrontate le prestazioni in termini di errore medio e di corrispondenze tra stime e conteggi effettivi delle tre strategie. Si è osservato che la soluzione migliore consiste nell'uso dell'oracolo. Il predittore, invece, presenta un leggero miglioramento rispetto a Count-Min sketch, ma non significativo come per il caso di `Oracle`. Si deve tenere comunque conto che, nonostante l'accuratezza presenti un miglioramento, si rende necessario effettuare due letture dello stream nelle alternative proposte, rivelandosi quindi meno efficiente di Count-Min sketch da un punto di vista di complessità temporale.

5.3 Soluzioni alternative per aumentare l'accuratezza di Count-Min Sketch

In questa sezione vengono presentate delle strategie alternative per migliorare l'accuratezza nella stima dei conteggi di Count-Min sketch. Tra di esse figurano Augmented sketch, che permette di migliorare anche la complessità temporale di risposta ad una point query, k-ary sketch, frequency aware Counting e gSketch, spiegate nel seguito.

5.3.1 Augmented Sketch

Una possibile soluzione, proposta in [35], consiste nell'affiancare un filtro a Count-Min sketch. Più nel dettaglio, similmente a quanto visto per l'oracolo e il predittore, se il filtro occupa uno spazio pari a s_f, generalmente di dimensioni molto ridotte, il numero di bucket $B^{'}$ utilizzato per Count-Min sketch coincide con lo spazio complessivo occupato dal solo Count-Min sketch originario, B. Per ottenere lo sketch di dimensioni $B^{'}$ si riduce il numero di funzioni di hash utilizzate (e quindi il numero di righe della matrice), oppure si diminuisce il numero di colonne (conseguentemente il range di valori in cui le funzioni di hash mappano i conteggi degli item). Considerando dunque t_f e t_s i tempi di aggiornamento del filtro e dello sketch rispettivamente, è possibile affermare che il tempo di aggiornamento di Augmented Sketch si configura essere pari a $t_f + \text{filter}_{selectivity} \cdot t_s$, dove il termine $\text{filter}_{selectivity}$ coincide con il rapporto tra il conteggio degli item che superano il filtro e il conteggio totale di tutti gli elementi ([35]). In questo modo il tempo di elaborazione aumenta di t_f ma, al contempo, si riduce la complessità temporale nello sketch di un fattore pari a $\text{filter}_{selectivity}$. Con questa struttura dati proposta, inoltre, l'errore massimo di stima si riduce a $\left(\frac{e}{d-\frac{s_f}{w}} N_2 \left(\frac{N_2}{||a||_1}\right)\right)$, con probabilità e^{-w}, dove $N_2 = ||a||_1 - N_1$; N_1 si configura essere la dimensione del flusso elaborato nel filtro, quindi N_2 coincide con la quantità del flusso elaborato dallo sketch. Con Count-Min sketch si ricorda invece che l'errore massimo si attesta a $\Theta\left(\frac{1}{w}||a||_1\right)$, con probabilità $\Theta(e^{-d})$. Siccome $N_2 \frac{N_2}{||a||_1} << ||a||_1$,

l'errore di stima atteso per Augmented sketch è notevolmente inferiore all'errore ottenuto con Count-Min sketch.
Per quanto riguarda l'algoritmo, innanzitutto si spiega che il filtro memorizza item ad alta frequenza (che superano cioè una certa soglia) e due conteggi, definiti `new_count`, che denota la stima della frequenza dell'item, ed `old_count`, che indica il conteggio dell'elemento nell'istante in cui è stato inserito nel filtro. La differenza tra questi due contatori rappresenta la frequenza aggregata esatta che si è accumulata durante il tempo che il conteggio dell'item è stato memorizzato all'interno di questa struttura. Count-Min sketch viene utilizzato per salvare invece i conteggi dei restanti elementi, meno frequenti, similmente a quanto visto negli algoritmi già analizzati. A questo punto, ogni item i nel flusso viene esaminato: se il conteggio di esso è già presente nel filtro, allora il corrispondente contatore è aumentato. Altrimenti, se in questo è presente almeno una cella vuota, viene inserita una nuova tupla (i, c_i), con c_i conteggio dell'elemento i. Se invece il filtro fosse "pieno" allora l'elemento con la relativa frequenza vengono mappati in Count-Min sketch. Si tiene poi traccia del conteggio minimo `new_count`; se questo valore è inferiore alla frequenza dell'ultimo item inserito nello sketch si effettua uno scambio tra questi due elementi. Nello sketch si mappa, dunque, la quantità `new_count - old_count` per l'elemento scambiato, inizialmente contenuto nel filtro. Questo è dovuto al fatto che `old_count` può essere solamente o 0 (il che indica che l'elemento non è mai stato inserito nello sketch) o, altrimenti, presentare una frequenza già contenuta nello sketch. Nel filtro vengono quindi memorizzati i $\lfloor \frac{s_f}{dim(item)} \rfloor$ elementi più frequenti, paragonabili agli heavy hitter, similmente al caso già visto mediante l'uso di predittore ed oracolo. Per questi elementi l'errore di stima è dunque nullo. Inoltre, come precedentemente osservato, questo fa sì che venga ridotta la probabilità di collisioni con gli elementi ad alta frequenza, comportando conseguentemente la riduzione dell'errore di falsi positivi. L'aumento nell'errore di stima del conteggio di elementi non heavy hitter a causa della riduzione delle dimensioni dello sketch risulta nella pratica trascurabile poiché l'accuratezza della stima è critica principalmente per elementi ad alta frequenza. Come dimostrato in [35], l'incremento di questo errore, ΔE, è tale da soddisfare la seguente disequazione:

$$\Delta E \leq \frac{es_f}{wd\left(w - \frac{s_f}{d}\right)}||a||_1$$

In accordo a quanto osservato, tale limite superiore è ragionevolmente piccolo, anche per data stream di dimensioni significative.

5.3.2 k-ary sketch

Questa struttura dati, è stata introdotta in [27], come alternativa allo sketch classico con lo scopo identificare cambiamenti significativi nel data stream. Essa si propone di utilizzare una piccola quantità di memoria e presenta un costo di aggiornamento e di ricostruzione costante. Similmente ad uno sketch, k-ary sketch consiste in una tabella (in altre parole,

una matrice) di wd contatori, con w colonne e d righe. Ciascuna riga è associata una funzione di hash. Ciò in cui differisce dallo sketch studiato consiste nell'eseguire operazioni più semplici ma più efficienti. Ad esempio, l'aggiornamento di un conteggio di un item segue la stessa procedura dello sketch, mentre per la stima viene preso il valore mediano della seguente quantità $\frac{a_i - \sum_{j \in w} a_{ij}/w}{1-1/w}$. Un'altra operazione ammessa è quella di combinazione di sketch differenti: considerando l strutture di sketch $S_1 \dots S_i \dots S_l$ e i rispettivi conteggi complessivi (somma dei singoli conteggi per ogni elemento) $c_1 \dots c_i \dots c_l$, la combinazione di questi è pari a $S = \sum_{k=1}^{l} c_k \cdot S_k$, tale per cui ogni entry dello sketch risultante, S, è data da $c_{ij} = \sum_{k=1}^{l} c_k \cdot c_{S_{ijk}}$. Grazie alle sue proprietà di linearità (caratteristiche degli sketch), gli autori in [27] suggeriscono di utilizzare tale struttura dati per riassumere traffici a diversi livelli. A questo punto, sopra di questi "riassunti" vengono implementati una serie di modelli di predizione temporale che permettono di identificare cambiamenti significativi cercando nei flussi che presentano errori di predizione elevati.

5.3.3 gSketch

Questa struttura dati, presentata in [43], è utilizzata principalmente per migliorare la stima in risposta ad edge query quando viene elaborato un graph stream, vale a dire un flusso di archi (i, j) di cui un esempio possono essere le reti sociali e di comunicazione. Per questa tipologia di flussi l'utilizzo di Count-Min sketch per rispondere ad una edge query (il cui obiettivo è quello di determinare il conteggio di un particolare arco) si rivela inefficiente. Infatti, l'errore di stima relativo per un singolo item è limitato superiormente da $N/(w \cdot a_{(i,j)})$, con N numero di vertici del grafo e $a_{(i,j)}$ conteggio dell'arco (i, j). L'errore è quindi elevato per elementi poco frequenti e per flussi di dimensioni significative. Tuttavia la distribuzione delle query può essere tale da richiedere ripetutamente il conteggio di questi archi, risultando in un errore elevato. Gli autori dell'articolo citato propongono, quindi, di sfruttare proprietà e relazioni intrinseche di questi stream, come l'asimmetria e la similarità locale, per cui i conteggi degli archi localmente vicini sono correlate fra loro. Per questo motivo, si partiziona uno sketch, che memorizza i conteggi dell'intero flusso, in r sketch di dimensioni ridotte e localizzati, tali per cui $w_l = w/r$ e $d_l = d$, dove w e d sono il numero di colonne e righe dello sketch di partenza. In ognuno di essi vengono memorizzati i conteggi di archi in modo tale che archi appartenenti a due regioni distinte vengano memorizzati in due sketch differenti. In questo modo l'errore relativo medio si riduce. Per di più, vengono sfruttate le proprietà strutturali del graph stream in modo da approssimare efficientemente le relazioni tra conteggi, che vengono poi sfruttate per partizionare i vari item. L'errore relativo per l'arco (i, j) per uno di questi "sottosketch", S_l è quindi pari a $\frac{C(S_l) - a_{(i,j)}}{w_l}$. Il criterio di suddivisione dello sketch consiste nel minimizzare l'errore di stima relativo, combinando la partizione dei vertici con quella degli archi. Successivamente, si utilizza una funzione di hash, $H : V \to S_l, \forall l : 1 \leq l \leq r$, tale per cui il generico arco (i, j) viene mappato nello sketch $H(i) = S_l$.

5.3.4 Frequency-Aware Counting

Gli autori in [41], presentano un miglioramento di Count-Min sketch in caso di architettura multicore (per la precisione si fa riferimento al processore Cell). In questo caso, quindi, sono presenti m processori distinti, ciascuno con una propria memoria locale privata e tali per cui la comunicazione avviene tramite messaggi oppure strutture dati condivisi. Ciascuno di questi elabora una porzione dello stream in input, di cui mantiene i conteggi grazie ad uno sketch locale. Per rispondere ad una point query, dunque, tutti i processori elaborano una propria stima. La risposta finale è costituita dalla combinazione di queste ultime. In una prima fase viene, dunque, partizionato il flusso, o sfruttando le relazioni tra conteggi degli elementi (Hash-based partitioning) o considerando m blocchi (Block-based partitioning). I conteggi degli elementi di ciascun "sottoflusso" vengono salvati in processore in una struttura simile a Count-Min sketch. In particolar modo, ciascuno sketch è costituito da d righe (e quindi funzioni di hash) e w colonne. Tuttavia, il numero di funzioni di hash che mappano il conteggio di ogni elemento dipende dalla frequenza dell'item stesso. Per gli elementi heavy hitter, infatti, si utilizza un numero di funzioni di hash inferiore rispetto agli item a bassa frequenza, in modo tale da ridurre il numero di collisioni costose. Il sottoinsieme di funzioni di hash si configura come valore di hash dell'item, determinato usando due funzioni aggiuntive, per calcolare un offset iniziale nelle righe e per determinare un gap tra righe consecutive. Questo sottoinsieme, inoltre, è calcolato seguendo una modalità round-robin. Per identificare gli heavy hitter si utilizza Misra-Gries, come già applicato nel caso dell'oracolo (5.2.1). Questo algoritmo è particolarmente vantaggioso, infatti, quando il flusso (e quindi l'identità degli heavy hitter) evolve dinamicamente. Successivamente, il procedimento di risposta ad una point query è simile a quello di Count-Min sketch. Infatti, viene ritornato il valore minimo tra quelli presenti. Per la struttura di Frequency-Aware Counting, tuttavia, il numero di confronti tra tutti i conteggi stimati per ottenere il minimo è inferiore per un elemento heavy hitter rispetto ad uno meno frequente.

In conclusione, in questa sezione si sono presentate delle modifiche alternative ad algoritmi di machine learning, o tecniche predittive come oracolo e predittore per migliorare l'accuratezza nella stima. Si sono esaminate, quindi, Augmented sketch, k-ary sketch, gSketch e Frequency-Aware Counting. Similmente a quanto visto nei casi esaminati nella sezione precedente, l'idea alla base di tutte queste tecniche consiste nello sfruttare le proprietà intrinseche e le dipendenze tra i dati del flusso.

Capitolo 6

Conclusioni

In questo elaborato è stato, innanzitutto, presentato Count-Min sketch, che si configura come una struttura dati probabilistica facente parte dei "counting sketch" e "streaming algorithm", per memorizzare gli elementi di un flusso di dati. Esso è costituito da un array bidimensionale, con $w = \lceil \frac{e}{\varepsilon} \rceil$ colonne e $d = \lceil \ln \frac{1}{\delta} \rceil$ righe, tale per cui ad ogni riga è associata una funzione di hash differente, con ε e δ parametri caratteristici di tale struttura. Ciascuna delle funzioni di hash, dunque, mappa l'item in una colonna distinta, quindi per ogni elemento del data stream, il corrispondente conteggio è mappato d volte.

Si sono quindi introdotte le principali tipologie di query a cui si può rispondere usufruendo di Count-Min sketch. Queste sono le point query, le inner product query ed infine le range query. Sono state, dunque, mostrate le operazioni per rispondere a ciascuna classe di query, concentrandosi in particolar modo sulle point query, di cui si è riportata la dimostrazione sui limiti inferiori e superiori dell'errore di stima.

Successivamente, è stato fatto un confronto in termini di accuratezza nella stima e complessità temporale e spaziale, tra Count-Min sketch e altre strutture di sketch, quali tug-of-war sketch, Fast-Count sketch, Count Sketch e random subset sums. Si è dimostrato, quindi, che la struttura dati analizzata presenta diversi vantaggi sotto i punti di vista sopracitati rispetto agli altri sketch.

A questo punto, si sono illustrate le diverse applicazioni di Count-Min sketch. Prima si sono prese in considerazione casi più astratti, come i $\phi - Quantiles$ e gli *Heavy Hitter* (di cui comunque sono state riportate anche delle applicazioni pratiche), per poi passare ad ambiti più concreti: tale sketch può essere, infatti, utilizzato nel Natural Language Processing, nel controllo di flusso di reti (per determinare gli URL o gli indirizzi IP più frequenti), nei database, in biologia computazionale e nel controllo delle password. In seguito, si sono mostrate possibili future applicazioni in cui Count-Min sketch si rivelerebbe efficace.

Dopo aver implementato Count-Min sketch in Java, sono stati realizzati dei grafici che mostrano le prestazioni in termini di accuratezza nella stima del conteggio e di errore

medio commesso, variando i parametri ε e δ e le dimensioni del flusso di dati esaminato. Più precisamente, si sono innanzitutto si sono realizzati grafici in cui viene mostrato l'errore medio nella stima dei conteggi al variare delle dimensioni dello sketch e delle dimensioni del flusso di dati (modellato come una distribuzione di Zipf). Per ogni stream si è osservata una decrescita inversamente proporzionale all'aumentare delle righe e delle colonne di Count-Min sketch. Successivamente, si è studiato graficamente il numero di elementi la cui stima non coincide con il conteggio effettivo all'aumentare delle dimensioni dello sketch. Si è osservato che per valori dello sketch relativamente piccoli rispetto alle dimensioni del flusso il numero di elementi contati inesattamente è molto elevato e circa costante. All'aumentare della grandezza della matrice, tuttavia, questo numero tende a diminuire, molto rapidamente, raggiungendo alla fine un valore prossimo a 0. Infine, si sono costruiti dei grafici che, al variare di ε e δ, mostrano come varia il numero di elementi contati correttamente. Idealmente, per valori molto piccoli di questi parametri, l'andamento dovrebbe essere quello di una retta passante per l'origine e con coefficiente angolare unitario. É stato fatto notare come tale obiettivo si ottiene più facilmente diminuendo, anche di poco, ε, in quanto dalla costruzione dello sketch, il numero di colonne aumenta linearmente al diminuire di tale parametro, riducendo di conseguenza il numero di collisioni tra elementi distinti. Nel caso di δ, invece, il numero di righe incrementa in modo logaritmico rispetto alla diminuzione di δ stesso. Di conseguenza, per ottenere lo stesso andamento lineare, δ deve assumere valori molto più piccoli rispetto a quelli di ε.

Successivamente, sono state introdotte tecniche che permettono di migliorare le prestazioni in termini di accuratezza della stima del conteggio. Esse permettono di identificare proprietà del flusso e relazioni tra item. In particolar modo, gli algoritmi esaminati distinguono tra elementi heavy hitter e non heavy hitter. Con questa informazione, è possibile salvare il conteggio di ciascuno dei primi in un bucket distinto, in modo da avere il valore esatto ed evitare, di conseguenza, collisioni importanti con elementi poco frequenti. Ciò permette di ridurre l'errore medio nella stima del conteggio e, di conseguenza, aumentare l'accuratezza. Per identificare gli elementi più comuni, si è quindi in un primo momento introdotta l'idea di utilizzare algoritmi di machine learning che identificano gli heavy hitter in base alle proprietà che soddisfano (in un testo per esempio, a parole più corte è associata maggiore frequenza). Si sono, poi, riportate i teoremi, con le corrispettive dimostrazioni, riprese da [24], che illustrano formalmente i vantaggi, in termini di accuratezza, di questo approccio.

In seguito, sono stati introdotti i concetti di oracolo e di predittore, come alternativa agli algoritmi sopraindicati. Il primo ha la funzione di determinare con esattezza quali elementi sono heavy hitter. Per fare questo, si è ripreso l'algoritmo di Misra-Gries, fissando il numero di elementi più frequenti ad una determinata percentuale delle dimensioni complessive del flusso esaminato. I conteggi di questi elementi, come già accennato, sono salvati in bucket distinti. Per quanto riguarda il predittore, invece, si utilizzato un Count-Min sketch di dimensioni ridotte per salvare, in modo approssimato, i conteggi dei vari elementi del

data stream. A questo punto, vengono classificati come heavy hitter tutti gli item il cui conteggio stimato supera una certa soglia. Per questi, il conteggio viene salvato in un bucket distinto, mentre per i restanti, si utilizza un Count-Min sketch di dimensioni maggiori.
Effettuando, dunque, degli esperimenti con l'oracolo e il predittore, si è ottenuto che il primo presenta le prestazioni migliori (in termini di elementi contati correttamente ed errore medio, in particolare per distribuzioni di dimensioni notevoli), mentre quelle ottenute con il predittore sono simili a quelle ricavate mediante il solo utilizzo di Count-Min sketch. in questi casi si considera che le dimensioni delle strutture utilizzate per salvare i conteggi degli heavy hitter siano trascurabili rispetto alle dimensioni dello sketch. In seguito, si sono confrontati i risultati riguardo al numero di item contati correttamente (match) utilizzando lo sketch con l'oracolo e lo sketch singolo in modo tale che, asintoticamente, lo spazio in memoria complessivo sia lo stesso. Ancora una volta, l'oracolo si configura come l'opzione migliore.
Si è quindi svolta un'analisi asintotica sulla complessità temporale e spaziale nei tre casi (Count-Min sketch singolo, oracolo e predittore).

Nell'ultima sezione, infine, sono state introdotte delle alternative agli algoritmi predittivi, che permettono di migliorare l'accuratezza nella stima dei conteggi di Count-Min sketch. Alcune di esse vengono applicate in particolari contesti; ad esempio, gSketch fornisce una risposta approssimata ad edge query, mentre Frequency-Aware Counting viene applicato in caso di architettura di processore multicore.

Bibliografia

[1] Rakesh Agrawal, Tomasz Imieliński e Arun Swami. “Mining association rules between sets of items in large databases”. In: *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (1993), pp. 207–216.

[2] Noga Alon, Yossi Matias e Mario Szegedy. “The Space Complexity of Approximating the Frequency Moments”. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (1996), pp. 20–29.

[3] Carlo Bonferroni. “Teoria statistica delle classi e calcolo delle probabilita”. In: *Pubblicazioni del R istituto superiore di scienze economiche e commericiali di firenze* 8 (1936), pp. 3–62.

[4] Andrei Broder e Michael Mitzenmacher. “Network applications of bloom filters: A survey”. In: *Internet mathematics* 1.4 (2004), pp. 485–509.

[5] Nicolò Cesa-Bianchi. “Conteggio approssimato”. In: *Complementi di Algoritmi e Strutture Dati* (2023).

[6] Moses Charikar, Kevin Chen e Martin Farach-Colton. “Finding frequent items in data stream”. In: *International Colloquium on Automata, Languages, and Programming* (2002), pp. 693–703.

[7] Jiecao Chen e Qin Zhang. “Bias-aware sketches”. In: *arXiv preprint arXiv:1610.07718* (2016).

[8] Saar Cohen e Yossi Matias. “Spectral Bloom Filter”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003).

[9] Graham Cormode. “Count-Min Sketch.” In: *AT&T Labs-Research* (2009).

[10] Graham Cormode. “Sketch techniques for approximate query processing”. In: *Foundations and Trends in Databases. NOW publishers* 15 (2011).

[11] Graham Cormode e Marios Hadjieleftheriou. “Finding frequent items in data streams”. In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1530–1541.

[12] Graham Cormode e Marios Hadjieleftheriou. “Finding the frequent items in streams of data”. In: *Communications of the ACM* 52.10 (2009), pp. 97–105.

[13] Graham Cormode e Shan Muthukrishnan. “An improved data stream summary: the count-min sketch and its applications”. In: *Journal of Algorithms* 55.1 (2005), pp. 58–75.

[14] Graham Cormode e Shan Muthukrishnan. “What’s hot and what’s not: tracking most frequent items dynamically”. In: *ACM Transactions on Database Systems (TODS)* 30.1 (2005), pp. 249–278.

[15] Graham Cormode e Shanmugavelayutham Muthukrishnan. “What’s New: Finding Significant Differences in Network Data Streams”. In: *IEEE/ACM Transactions on Networking* 13.6 (2005), pp. 1219–1232.

[16] Graham Cormode et al. “Finding hierarchical heavy hitters in data streams”. In: *Proceedings 2003 VLDB Conference* (2003), pp. 464–475.

[17] Cristian Estan e George Varghese. “New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring Mice”. In: *ACM Transactions on Computer Systems (TOCS)* 21.3 (2003), pp. 270–313.

[18] Min Fang et al. “Computing Iceberg Queries Efficiently”. In: *International Conference on Very Large Databases* (1999).

[19] Sumit Ganguly, Minos Garofalakis e Rajeev Rastogi. “Processing data-stream join aggregates using skimmed sketches”. In: *Advances in Database Technology-EDBT 2004: 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004 9* (2004), pp. 569–586.

[20] Anna C Gilbert et al. “How to summarize the universe: Dynamic maintenance of quantiles”. In: *VLDB’02: Proceedings of the 28th International Conference on Very Large Databases* (2002), pp. 454–465.

[21] Anna C Gilbert et al. “Quicksand: Quick summary and analysis of network data”. In: (2001).

[22] Amit Goyal e Hal Daumé III. “Approximate scalable bounded space sketch for large data nlp”. In: *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing* (2011), pp. 250–261.

[23] S. Muthukrishnan Graham Cormode. “An Improved Data Stream Summary: The Count-Min Sketch and its Application”. In: *Elsevier Computer Science* (2003).

[24] Chen-Yu Hsu et al. “Learning-Based Frequency Estimation Algorithms.” In: *International Conference on Learning Representations* (2019).

[25] Mark Huber. “Halving the bounds for the Markov, Chebyshev, and Chernoff inequalities using smoothing”. In: *The American Mathematical Monthly* 126.10 (2019), pp. 915–927.

[26] Michael Huerta et al. “NIH working definition of bioinformatics and computational biology”. In: *US National Institute of Health* (2000), pp. 1–1.

[27] Balachander Krishnamurthy et al. “Sketch-based change detection: Methods, evaluation, and applications”. In: *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 234–247.

[28] Wentian Li. "Zipf's Law everywhere." In: *Glottometrics* 5.2002 (2002), pp. 14–21.

[29] Edo Liberty. "Simple and deterministic matrix sketching". In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (2013), pp. 581–588.

[30] Elizabeth D Liddy. "Natural language processing". In: (2001).

[31] Thodoris Lykouris e Sergei Vassilvitskii. "Competitive caching with machine learned advice". In: *Journal of the ACM (JACM)* 68.4 (2021), pp. 1–25.

[32] Guillaume Marçais e Carl Kingsford. "A fast, lock-free approach for efficient parallel counting of occurrences of k-mers". In: *Bioinformatics* 27.6 (2011), pp. 764–770.

[33] Jayadev Misra e David Gries. "Finding repeated elements". In: *Science of computer programming* 2.2 (1982), pp. 143–152.

[34] Shanmugavelayutham Muthukrishnan et al. "Data streams: Algorithms and applications". In: *Foundations and Trends® in Theoretical Computer Science* 1.2 (2005), pp. 117–236.

[35] Pratanu Roy, Arijit Khan e Gustavo Alonso. "Augmented sketch: Faster and more accurate stream processing". In: *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1449–1463.

[36] Russel Russel. *Genetica*. Edises, 2002.

[37] Florin Rusu e Alin Dobra. "Statistical analysis of sketch estimators". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (2007), pp. 187–198.

[38] Stuart Schechter, Cormac Herley e Michael Mitzenmacher. "Popularity is everything: A new approach to protecting passwords from {Statistical-Guessing} attacks". In: *5th USENIX Workshop on Hot Topics in Security (HotSec 10)* (2010).

[39] Benedikt Sigurleifsson, Aravindan Anbarasu e Karl Kangur. "An overview of count-min sketch and its applications". In: (2019).

[40] Ramakrishnan Srikant e Rakesh Agrawal. "Mining quantitative association rules in large relational tables". In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data* (1996), pp. 1–12.

[41] Dina Thomas et al. "On efficient query processing of stream counts on the cell processor". In: *2009 IEEE 25th International Conference on Data Engineering* (2009), pp. 748–759.

[42] Qingpeng Zhang et al. "These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure". In: *PloS one* 9.7 (2014), e101271.

[43] Peixiang Zhao, Charu C Aggarwal e Min Wang. "gsketch: On query estimation in graph streams". In: *arXiv preprint arXiv:1111.7167* (2011).