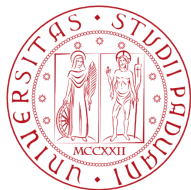


corso di laurea in Ingegneria Informatica

Matteo Piovesan

Analisi e sperimentazione di algoritmi di counting sul modello di streaming sliding window

Relatore:
Prof. Pucci Geppino

anno accademico 2024-2025

Indice

1 Introduzione **5**

2 BasicCounting **7**

3 Sum **12**

4 Generalizzazione **14**

5 Implementazione **15**

6 Conclusione **19**

Capitolo 1

Introduzione

Un modello a sliding window è un modello computazionale che, data una stream di dati, esegue una certa funzione f sugli ultimi N elementi di tale stream. In questo modo vengono ignorati gli elementi più vecchi, dando priorità agli ultimi arrivati.

Un esempio di uso di una sliding window è un qualche sensore che genera una stream di valori e, per aumentare la precisione, siamo interessati a calcolare il valore medio solo degli ultimi N valori, poichè gli altri sono ormai troppo vecchi per ricavare informazioni utili. È facile vedere che, in questo caso, l'utilizzo di memoria sia $\Theta(N)$ perché per ogni elemento in input dalla stream dobbiamo almeno registrare il tempo di arrivo, in modo da sapere quando poterlo scartare. Si può notare che, se si ignorasse la sliding window, per calcolare la media di tutti gli elementi arrivati finora si dovrebbe registrare solo la somma attuale ed il numero di elementi arrivati.

Questo approccio, pur essendo corretto, comincia a non essere praticabile con finestre molto ampie. Tornando all'esempio di prima: se si vuole mantenere una sliding-window di 10^5 elementi il sensore potrebbe non avere disponibile memoria sufficiente per gli elementi e le rispettive timestamp (oppure si vuole usare un chip di memoria meno capiente, quindi meno costoso).

Una soluzione è cercare un compromesso tra la memoria utilizzata e l'errore massimo ammissibile ϵ. Se si vorrà un'accuratezza del 50% si porrà $\epsilon = 0.5$ ed usando una struttura di dati particolare (Istogrammi Esponenziali) si garantirà che l'errore non superi ϵ utilizzando un numero di parole di memoria logaritmico rispetto a N.

In questo documento vedremo nello specifico:

BasicCounting Algoritmo che, data una stream di elementi che possono assumere come valore 0 o 1, mantenere per ogni istante di tempo il numero di 1 presenti negli ultimi N elementi.

Sum Algoritmo che, data una stream di elementi di valore compreso tra 0 e R, mantenere per ogni istante di tempo la somma degli ultimi N elementi.

Generalizzazione Generalizzazione dei problemi per cui è possibile utilizzare istogrammi esponenziali.

Implementazione Una implementazione degli algoritmi BasicCounting e Sum in C++, con statistiche di errore medio e utilizzo memoria.

Capitolo 2

BasicCounting

Il problema di BasicCounting riguarda tutti i problemi che richiedono un conteggio approssimato di certi elementi dentro la finestra attuale. Se, ad esempio, si vuole far in modo che un router conti i pacchetti provenienti da un particolare IP nell'ultimo secondo, questi ultimi genereranno un 1 nella stream della sliding-window, tutti gli altri uno 0. Per ridurre lo spazio occupato dall'algoritmo si utilizzeranno degli istogrammi di dimensioni variabili, dove ogni istogramma rappresenta una parte temporale contigua della finestra attuale. Il vantaggio principale di questo metodo rispetto al random sampling è che la presenza di 1 sparsi nella finestra non ne diminuisce la precisione, mentre l'avere istogrammi di dimensione variabile ci permette di usarne di meno rispetto all'approccio con gli istogrammi di dimensione fissa. Questa struttura dati è chiamata Istogrammi Esponenziali dal momento che, come vedremo in seguito, la loro dimensione sarà limitata a varie potenze di 2.

Prima di continuare è necessario introdurre due concetti: *bucket* e *timestamp*. Bucket è semplicemente il nome che daremo a tali istogrammi, la cui *size* sarà il numero di 1 che contiene. La timestamp è il tempo di arrivo di un elemento ma, quando riferita ai bucket, è il tempo di arrivo dell'ultimo elemento (quello più recente) che esso contiene, in questo modo quando l'elemento di questa timestamp uscirà dalla finestra implicherà che anche tutti gli elementi di quel bucket, essendo più antichi, saranno usciti. L'errore è dovuto proprio al fatto che conserviamo solo una timestamp per bucket. La timestamp può venire conservata usando un wraparound counter di $\log N$ bit, sufficiente per sapere quando scartare l'ultimo

bucket. Si può notare che, in ogni istante di tempo, solo l'ultimo bucket (quello che contiene gli elementi più antichi) potrebbe avere degli elementi non più validi. Per dimostrarlo indichiamo con B_i il bucket numero i e con B_m l'ultimo bucket. Ipotizziamo per assurdo che il bucket B_{m-1} abbia degli elementi non più validi. Dal momento che i bucket contengono elementi *contigui*, se anche un solo elemento in B_{m-1} non fosse più valido implicherebbe che tutti gli elementi dei bucket precedenti (ovvero con $i > m-1$) siano completamente fuori finestra ed abbiano la timestamp scaduta. Ma quando la timestamp del bucket è scaduta questo viene scartato, quindi B_m non può essere l'ultimo bucket (perchè scartato) e l'ultimo bucket diventerebbe B_{m-1}, che per ipotesi contiene degli elementi non validi. Questo implica anche che un bucket, per esistere, deve avere almeno un elemento ancora valido all'interno (il più recente).

Indicando con C_i la size del bucket B_i il numero di 1 ancora validi S è massimo

$$S_{max} = \sum_{n=1}^{m} C_n$$

(quando l'ultimo bucket è completamente nella finestra) e minimo

$$S_{min} = 1 + \sum_{n=1}^{m-1} C_n$$

(quando l'ultimo bucket ha solo un elemento valido).

Per minimizzare l'errore nel caso peggiore, all'arrivo di una query possiamo approssimare il numero di 1 validi a $(C_m/2) + \sum_{n=1}^{m-1} C_n$, rendendo l'errore assoluto massimo $C_m/2$ e, quindi, l'errore relativo massimo $\frac{C_m}{2(1+\sum_{n=1}^{m-1} C_n)}$ (dove al denominatore ipotizziamo che il numero esatto di 1 sia il minimo possibile per massimizzare l'ipotesi di errore relativo). Quindi tenendo valida la disequazione $\frac{C_m}{2(1+\sum_{n=1}^{m-1} C_n)} \leq \epsilon$ ci assicura che l'errore sia al più quello voluto. Questo ci porta alla prima invariante dell'algoritmo.

Invariante 1 In ogni istante di tempo, le size dei bucket $C_1, \ldots, C_m$ sono tali che

$$\forall j \in [1 \ldots m], \frac{C_j}{2(1 + \sum_{n=1}^{j-1} C_n)} \leq \epsilon$$

Come anticipato, gestendo le size dei bucket in modo da renderle potenze di 2 ci permette di mantenere l'invariante 1.

Definiamo $k = \lceil \frac{1}{\epsilon} \rceil$ ed assumiamo $\frac{k}{2}$ sia intero, altrimenti si approssima a $\lceil \frac{k}{2} \rceil$. Usando k ora definiamo la seconda invariante:

Invariante 2 In ogni istante di tempo:

- Le size dei bucket $C_1, \ldots, C_m$ sono non decrescenti ($C_1 \leq C_2 \leq \ldots \leq C_m$)
- Le size dei bucket $C_1, \ldots, C_m$ sono potenze di 2 ($1, 2, \ldots, 2^i$) per $i \leq \log \frac{2N}{k} + 1$
- Per ogni size diversa dalla prima e dall'ultima ci sono almeno $\frac{k}{2}$ bucket e al massimo $\frac{k}{2} + 1$. Per la size più piccola (uguale ad 1) ci sono almeno k e al massimo $k + 1$ bucket. Per la size più grande (uguale a $2^{\log \frac{2N}{k} + 1} = \frac{4N}{k}$) ci sono al massimo $\frac{k}{2}$ bucket.

Si può dimostrare come mantenendo valida la seconda invariante implica che anche la prima sia valida.

Dal momento che $k = \frac{1}{\epsilon}, \epsilon = \frac{1}{k}$ possiamo riscrivere la prima invariante come:

$$C_m \leq \frac{2}{k}(1 + \sum_{n=1}^{m-1} C_n)$$

Le size dei bucket sono limitate a potenze di 2:

$$2^m \leq \frac{2}{k}(1 + \frac{k}{2}(2^m - 1) + \frac{k}{2})$$

dove abbiamo usato la formula $\sum_{n=1}^{b} a^n = \frac{a^{n+1}-1}{a-1}$ e considerato che ci sono almeno $\frac{k}{2}$ per bucket. Usando il numero minimo di bucket ci assicura che la disequazione sia valida anche per numeri maggiori di bucket. Abbiamo aggiunto $\frac{k}{2}$ per compensare i bucket mancanti di size 1, dal momento che sono almeno k rispetto al minimo degli altri bucket $\frac{k}{2}$. Dal momento che al massimo $m = \log \frac{2N}{k} + 1$:

$$\frac{4N}{k} \leq \frac{2}{k}(1 + \frac{k}{2}(\frac{4N}{k} - 1) + \frac{k}{2})$$

che dopo aver semplificato diventerà:

$$2N \leq 2N + 1$$

che è sempre valida dal momento che N, essendo la dimensione della finestra, è numero naturale. Si può anche notare che queste size di bucket sono sufficienti

per immagazzinare il massimo di 1 della finestra, ovvero N, quando tutta la finestra è composta da 1.

$$\sum_{n=1}^{m-1} C_n \geq N$$

Seguendo i passaggi di prima ci porta a:

$$4N - 1 + \frac{8N}{k} \geq N$$

formando la catena:

$$4N - 1 + \frac{8N}{k} \geq 4N - 1 \geq N$$

valida per $N \geq \frac{1}{3}$, ovvero per ogni numero naturale maggiore di 0.

Per mantenere l'invariante 2, appena si hanno troppi bucket di una size, si prendono i due bucket più antichi e si crea un bucket di dimensione doppia a quella del livello e come timestamp la più recente tra i due. Questo implica anche che i bucket di dimensione maggiore contengono gli elementi più antichi e quelli con dimensione minore quelli più recenti.

Con questo metodo, per ogni bucket, si conservano $\log N$ bit per la timestamp e $\log \log N$ bit per l'esponente (gli esponenti sono logaritmici rispetto alla finestra) per un totale $(\log N + \log \log N) \in O(\log N)$ bit per bucket. Al massimo ci sono $k + 1$ di size 1, $\frac{k}{2}$ per la size più grande, $(\frac{k}{2} + 1)(\log \frac{2N}{k} + 1)$ per le altre size ed 1 perl bucket che eventualmente causerà il merge, per un totale di $(\frac{k}{2} + 1)(\log \frac{2N}{k} + 2) + \frac{k}{2}$ bucket, quindi $O(\frac{1}{\epsilon} \log N)$ bucket. Quindi si ha un totale di $O(\frac{1}{\epsilon} \log^2 N)$ bit di memoria occupati dall'algoritmo. Mantenendo dei contatori per la size dell'ultimo bucket C_m e la somma delle size di tutti gli altri C_{tot} ci permette di rispondere alla query di conteggio in $O(1)$ tempo risolvendo l'equazione $S = (C_m/2) + C_{tot}$. Questi contatori verranno aggiornati quando l'ultimo bucket scadrà, con $C_{tot} = C_{tot} - C_{m-1}$ e $C_m = C_{m-1}$. Per garantire l'invariante durante l'inserimento di un nuovo elemento si eseguono le seguenti operazioni in questo ordine:

- All'arrivo di un nuovo elemento, a prescindere che sia 0 o 1, si aggiorna la timestamp e, se necessario, si aggiorna l'ultimo bucket ed i contatori C_m e C_{tot}.
- Se l'elemento è 0 questo viene ignorato e l'inserimento finisce qui.

- Se l’elemento è 1 viene inserito un nuovo bucket di size 1 con timestamp associata.
- Partendo da $i = 1$, se ci sono troppi bucket di size i questi vengono fusi in un nuovo bucket di size $2i$ e di timestamp quella più recente tra i due bucket in questione. Si ripete questo procedimento per il prossimo $i' = 2i$ fino a $i = \log \frac{2N}{k}$.
- Aggiornare, se necessario, il puntatore all’ultimo bucket ed i contatori C_m e C_{tot}.

È importante far notare che, nel caso peggiore, l’inserimento causerà un merge su ogni size di bucket, portando l’inserimento a $O(\log N)$ tempo.

Capitolo 3

Sum

Per tenere traccia della somma degli ultimi N elementi limitati tra $[0, \dots, R]$, anziché il semplice conteggio di alcuni elementi, instintivamente possiamo riutilizzare parte del codice usato per BasicCounting, il quale tiene la somma dei primi N elementi limitati a $[0, 1]$. Quindi l'algoritmo di Sum può essere visto come una generalizzazione di BasicCounting. Sempre usando gli Istogrammi Esponenziali, definiamo come size dei bucket, anzichè il numero di 1 in un'area contigua, la somma di elementi in un'area contigua. Le timestamp dei bucket rappresentano sempre la timestamp dell'elemento più recente che contengono. Definiamo, inoltre, con C_m* la somma di tutte le size dei bucket successivi a B_m (quindi $C_m* = \sum_{i=1}^{m-1} C_i$) e con $C_{i,i-1}$ la size che avrebbe il bucket creato dal merge di B_i e B_{i-1} (quindi $C_{i,i-1} = C_i + C_{i-1}$).

Definiamo l'invariante 3 come:

Invariante 3 Per ogni bucket B_i vale che $\frac{1}{2\epsilon} C_i \leq C_i*$

Analogamente a BasicCounting questa invariante ci permette di rendere l'errore relativo a ϵ. Si definisce anche l'invariante 4:

Invariante 4 Per ogni bucket B_i vale che $\frac{1}{2\epsilon} C_{i,i-1} > C_{i-1}*$

Questa invariante ci permette di ridurre i bucket utilizzati. Se il merge di due bucket non rispetta l'invariante 4 questi verranno uniti. Si noti come il bucket prodotto dal merge rispetti di conseguenza l'invariante 3.

Usando l'invariante 3 possiamo vedere come le size massime dei bucket crescono esponenzialmente. Assumendo che la size del primo bucket sia R, la size

del secondo bucket è al massimo $2\epsilon R$, del terzo $(4\epsilon^2 + 2\epsilon)R$ ecc. Notando che vale la regola $C_i = 2\epsilon C_{i-1} + C_{i-1}$ si può ricavare l'equazione $C_i = 2\epsilon(2\epsilon + 1)^{i-2}$ con $i > 1$. È importante far notare come per alcuni bucket l'invariante non potrebbe valere. Ad esempio se si inserisce due volte il valore a ed avessimo $\epsilon = 0.25$ si dovrebbe avere $a \leq 2\epsilon a$ e di conseguenza $\epsilon > \frac{1}{2}$ che, per ipotesi, non è vera. Questo non ci causa problemi perchè tali bucket hanno un solo elemento al loro interno (se fossero il risultato di un merge varrebbe l'invariante 3) e quindi, nel caso tali bucket diventassero ultimi bucket, sapremmo con esattezza la timestamp dell'unico elemento al loro interno, quindi non ci sarebbe errore.

Si può intuire come il numero di bucket necessari sia $O(\frac{1}{\epsilon}(\log N + \log R))$, dal momento che i bucket crescono esponenzialmente. Questa formula torna se consideriamo l'algoritmo BasicCounting come un algoritmo Sum con $R = 1$: si avrebbero $O(\frac{1}{\epsilon}(\log N + \log 1)) = O(\frac{1}{\epsilon}\log N)$ bucket, che torna come numero di bucket utilizzati dall'algoritmo BasicCounting. Ogni bucket occupa almeno $\log N$ bit per la timestamp e $\log R$ bit per la size per un totale di $\log N + \log R$ bit per bucket. L'utilizzo totale di memoria dell'algoritmo è quindo $O(\frac{1}{\epsilon}(\log N + \log R)^2)$ bit. Analogamente per l'algoritmo di BasicCounting si tiene un contatore per l'ultimo bucket C_m ed uno separato per la somma delle size degli altri C_m*. Per ogni inserimento si effettuano le seguenti operazioni in questo ordine:

- All'arrivo di un nuovo elemento, a prescindere dal valore, si aggiorna la timestamp e, se necessario, si aggiorna l'ultimo bucket ed i contatori C_m e C_m*.
- Se l'elemento è 0 questo viene ignorato e l'inserimento finisce qui.
- Se l'elemento è maggiore di 0 viene creato un nuovo bucket con timestamp associata e size come l'elemento appena arrivato.
- Per $i > 2$, se $C_{i,i-1} \leq 2\epsilon C_{i-1}*$ si effettua il merge sui bucket B_i e B_{i-1}. Si noti come il prossimo valore $C_{i-1}*$ può essere calcolato come $C_i* = C_{i-1} + C_{i-1}*$.

Dato che teniamo i contatori di C_m e C_m* la query può essere eseguita in $O(1)$ tempo. L'inserimento comporta uno sweep di tutti i bucket presenti, quindi $O(\frac{1}{\epsilon}(\log N + \log R)$ tempo.

Capitolo 4

Generalizzazione

Abbiamo visto come, utilizzando gli istogrammi esponenziali, possiamo risolvere in spazio logaritmico problemi di conteggio in modello sliding-window. Generalmente, il modello in sliding-window richiede di mantenere una funzione f lungo la finestra di elementi attivi. Per esempio, nel problema BasicCounting la funzione f era semplicemente il conto di 1 attivi. Per utilizzare gli istogrammi esponenziali in questi casi si definiscono vari punti:

Size dei bucket La size dei bucket rappresenta il valore della funzione f sulla porzione di finestra coperta dal bucket B.

Operazione di merge Se si deve effettuare un merge di due o più bucket viene creato un nuovo bucket con timestamp pari a quella del bucket più recente. La size di questo bucket verrà calcolata considerando le size dei bucket su cui facciamo il merge più eventuali informazioni situazionali. Per definizione, la size del nuovo bucket è la somma delle size dei bucket su cui effettuiamo il merge.

Stima Per stimare il valore della funzione $f(N)$, dal momento che è solo l'ultimo bucket che potrebbe avere elementi non più attivi, si interpola il risultato $f(C_m)$ e lo si somma al calcolo della funzione sugli altri bucket. Questo approccio funziona se f è debolmente additiva, ovvero se $f(A) + f(B) = f(A + B)$ per A e B disgiunti.

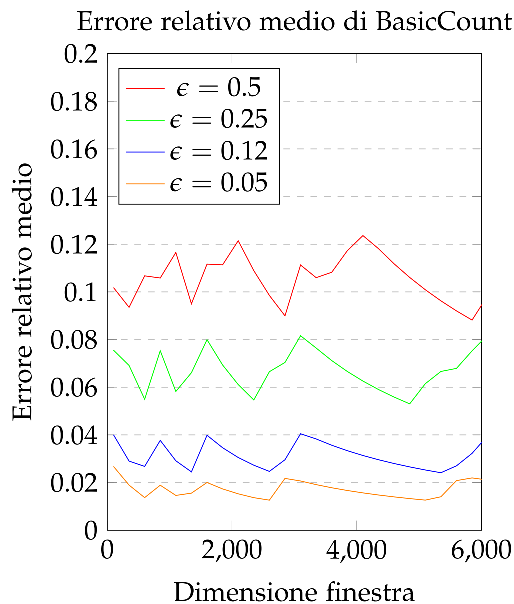
Eliminazione del bucket più antico A prescindere dalla funzione f, quando la timestamp del bucket più antico raggiunge $N + 1$, tutti gli elementi di quel

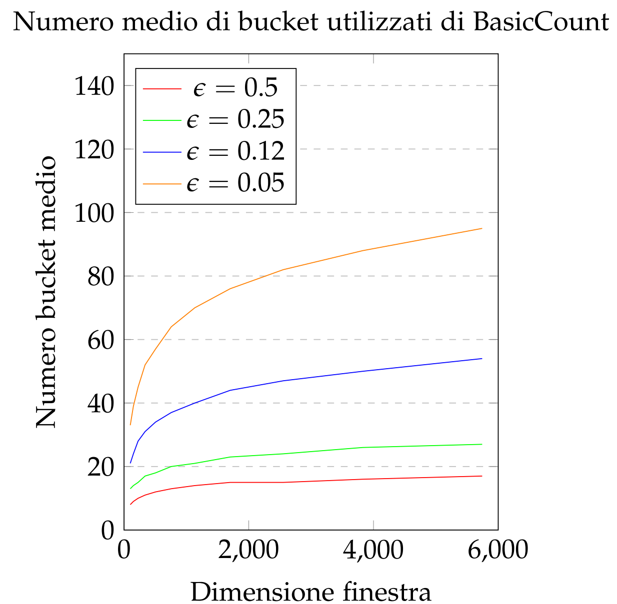
bucket sono scaduti. Il bucket quindi deve essere eliminato per conservare memoria.

Capitolo 5

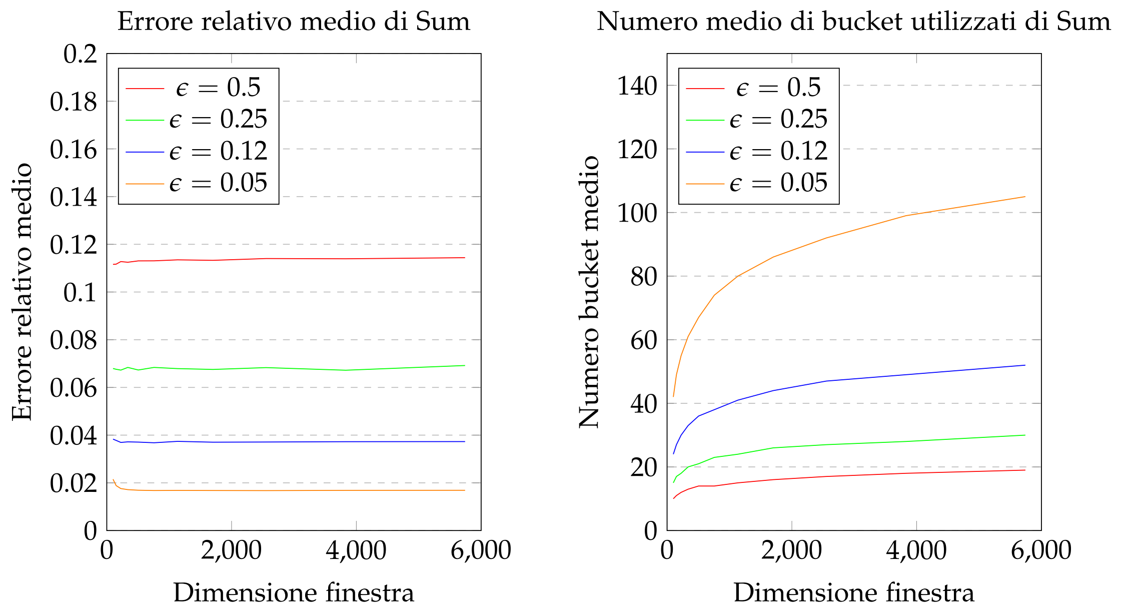
Implementazione

Si propone un'implementazione in C++ degli algoritmi di BasicCounting e Sum. Come stream di prova per BasicCounting si è utilizzato un file di 10^6 elementi con una densità di 1 del 50%, mentre per Sum un file da 10^6 di elementi compresi tra 0 e 250. Entrambi i file sono stati ottenuti usando la funzione rand() della libreria <cstdlib>. Il codice di entrambi gli algoritmi può essere trovato in fondo al documento.

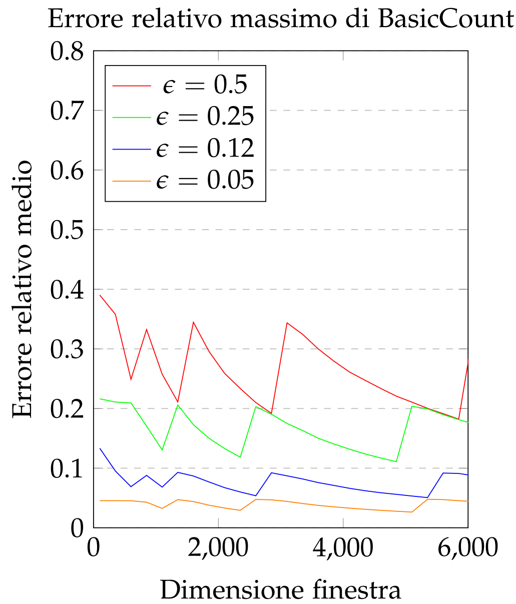


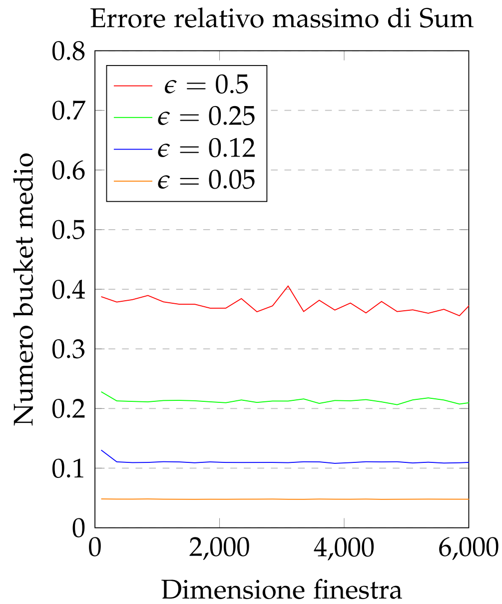


Si può notare come l'errore medio sia minore dell'errore massimo dato e di come il numero di bucket sia logaritmico in funzione della dimensione della finestra N, come previsto.

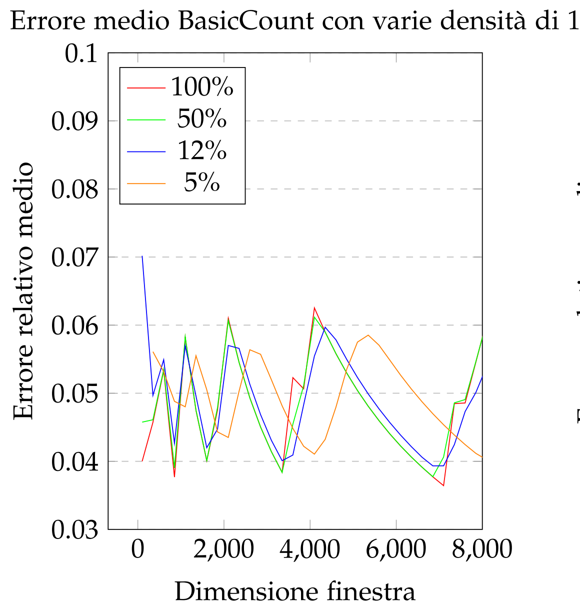


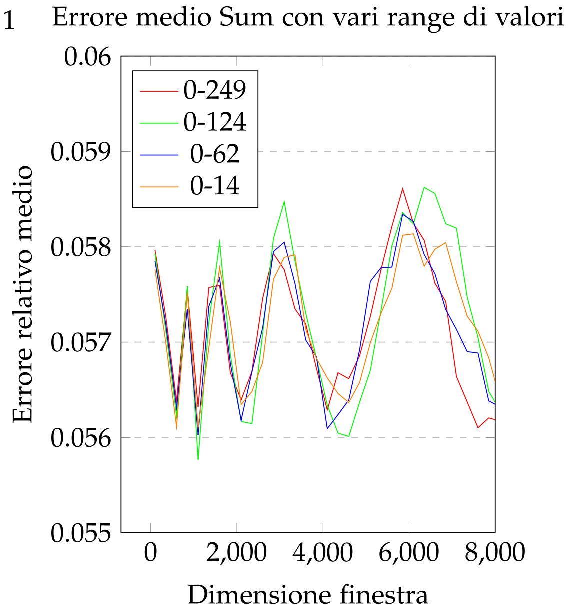
Per Sum si noti che l'errore medio varia molto poco. Questo è dovuto al range di valori più ampio, invece di essere solo un massimo e un minimo. Come per BasicCount, il numero di bucket utilizzati è logaritmico rispetto alla dimensione della finestra N.





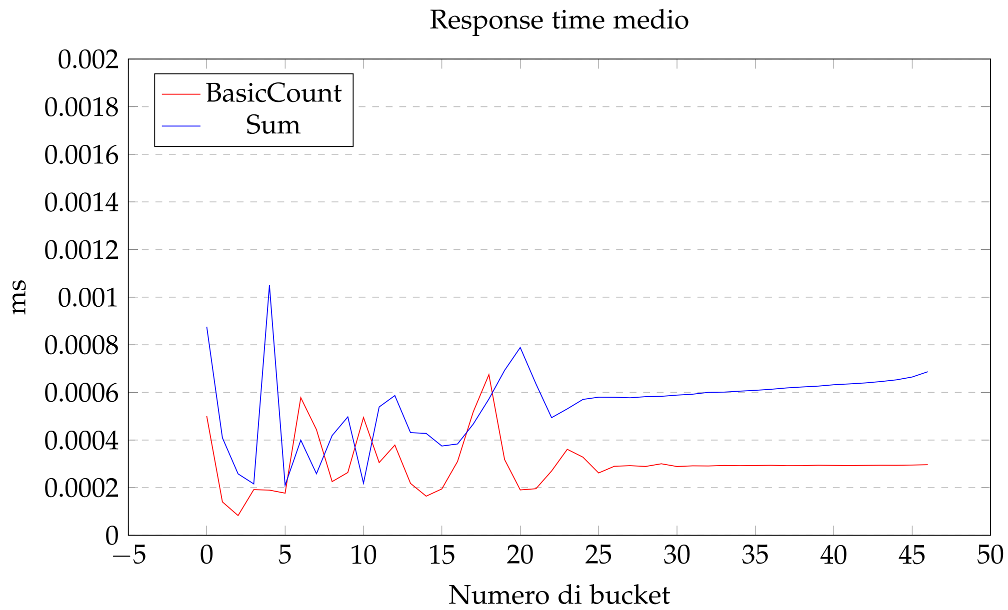
Si può vedere che, anche contando solo il caso peggiore per dimensione della finestra, l'errore è inferiore rispetto a ϵ.





Fissando $\epsilon = 0.2$, si nota che l'errore non viene influenzato significativamente dalla densità di 1 per BasicCounting o dal range di valori per Sum e l'errore oscilla con frequenza sempre più bassa. L'errore comincia ad aumentare in prossimità delle dimensioni di finestra abbastanza grandi da aumentare l'espo-

nente massimo per BasicCounting, e cresce poichè un bucket finale più grande, se presente, corrisponde ad un'errore più alto, per poi comminciare a scendere quando la finestra è abbastanza grande da contenere sufficienti 1 per abbassare l'errore relativo. Analogamente per Sum le window size dove l'errore comincia ad aumentare richiedono un bucket in più che aumenterà l'errore. La frequenza di questa oscillazione rallenta con la dimensione della finestra grazie al fatto che i bucket richiesti dall'algoritmo sono logaritmici rispetto alla dimensione della finestra.



Il response time cresce con l'aumentare dei bucket per via dei merge più frequenti. Sum ha un response time molto più alto di BasicCount per via del controllo sequenziale che deve fare dopo ogni inserimento per mantenere l'invariante 4.

Capitolo 6

Conclusione

In questo documento è stato visto il ragionamento, il funzionamento e una possibile implementazione di alcuni algoriti di counting in modello sliding window, che risolvono problemi di conteggio di elementi specifici in una finestra molto rapidamente e con poca memoria a costo di non essere completamente esatti. Algoritmi di questo tipo sono molto usati in situazioni dove la velocità è più importante della precisione, ad esempio un'approssimazione del risultato di un'operazione di Join di un database. L'implementazione di entrambi gli algoritmi è stata fatta utilizzando una linked list di classe Node (utilizzata per immagazzinare le informazioni dei bucket) con, nello specifico di BasicCount, vari punti di inserimento (per i vari esponenti). Per tenere traccia del tempo di arrivo dei bucket si è utilizzato un wrap-around counter resettato quando supera la dimensione della finestra. La sperimentazione ha dimostrato l'esattezza dei limiti superiori dell'utilizzo di memoria ed errore relativo.

```
//BasicCount.h
class BasicCount {
private:
    struct node {
        int value;
        int exp;
        node* next;
        node* prev;
    };
    int k, k_half, max_exponent, time, sum, window_size;
    node** array_last;
    node* start;
    node* stop;
    int* array_count;
public:
    BasicCount(float e, int n);
    ~BasicCount();
    void printall();
    void insert(int value);
    int eval();
private:
    void push(int exp, int value);
    int pop(int exp);
    void update();
};
```

```
//BasicCount.cpp
#include "BasicCount.h"
#include<cmath>
BasicCount::BasicCount(float e, int n) {
    k = ceil(1 / e);
    k_half = ceil((float)k / 2);
    max_exponent = ceil(log2(n / k_half))+1 ;
    array_count = new int[max_exponent + 1];
    array_last = new node * [max_exponent + 1];
    start = new node;
    stop = new node;

    start->next = stop;
    start->prev = nullptr;
    stop->next = nullptr;
    stop->prev = start;

    start->exp = -1;
    stop->exp = -1;
    time = 0;
    sum = 0;
    window_size = n;
    for (int i = 0;i < max_exponent + 1;i++) {
        array_count[i] = 0;
        array_last[i] = nullptr;
    }
}
BasicCount::~BasicCount() {

    node* iter = start->next;
    while (iter != stop) {
        node* temp = iter->next;
        delete iter;
        iter = temp;
    }
    delete start;
```

```
    delete stop;
    delete[] array_count;
    delete[] array_last;
}
void BasicCount::push(int exp, int value) {
    node* temp;
    //se exp e' maggiore di 0 esiste almeno un bucket di (exp-1)
    //il merge si attiva quando il numero di bucket per una size
    // raggiunge (k/2)+2, ma
    //(k/2)+2-2=0 -> k/2=0 -> k=0, il che e' impossibile
    if (exp == 0) temp = start;
    else temp = array_last[exp - 1];
    node* ins = new node;

    ins->exp = exp;
    ins->value = value;

    ins->next = temp->next;
    ins->prev = temp;
    temp->next->prev = ins;
    temp->next = ins;

    if (array_last[exp] == nullptr)array_last[exp] = ins;
    array_count[exp]++;
    sum += pow(2, exp);
}
int BasicCount::pop(int exp) {
    array_last[exp]->next->prev = array_last[exp]->prev;
    array_last[exp]->prev->next = array_last[exp]->next;
    int ret = array_last[exp]->value;
    sum -= pow(2, exp);
    array_count[exp]--;
    node* temp = array_last[exp];
    if (array_count[exp] == 0)array_last[exp] = nullptr;
    else array_last[exp] = array_last[exp]->prev;
```

```
    delete temp;
    return ret;
}
void BasicCount::insert(int value) {
    if (value != 0) push(0, time);
    time++;
    if (time > window_size) time = 0;
    update();
}
void BasicCount::update() {
    //merge phase
    if (array_count[0] > k + 1) {
        //L'elemento a sinistra contiene sicuramente l'elemento piu' recent
        //quindi lo prendiamo come timestamp
        pop(0);
        int a = pop(0);
        push(1, a);
        for (int i = 1;i <= max_exponent;i++) {
                if (array_count[i] > k_half + 1) {
                    pop(i);
                    int a = pop(i);
                    push(i + 1, a);
                }
              //Dal momento che il numero di bucket per esponente cresce
              //solo se e' avvenuto un merge tra bucket di esponente minore
              //Possiamo uscire se, in un esponente, non sono avvenuti merg
                else { break; }
            }
    }
    if (stop->prev->value == time) pop(stop->prev->exp);
}
int BasicCount::eval() {
    if (stop->prev->exp == 0) return sum;
    return sum - pow(2, (stop->prev->exp) - 1);
    //(2^x)/2=2^(x-1)
}
```

```
//Sum.h
class Sum {
private:
    struct node {
        int timest;
        int size;
        bool merged;
        node* next;
        node* prev;
    };
    int time, somma, window_size,lenght;
    float k;
    node* start;
    node* stop;

public:
    Sum(float e, int n);
    ~Sum();
    void insert(int value);
    int eval();
private:
    node* push(int size, int value, node* insert, bool merged);
    int pop(node* buck);
    void update();
};
```

```
//Sum.cpp
#include "Sum.h"
Sum::Sum(float e, int n) {
    k = 1 / (2 * e);
    start = new node;
    stop = new node;

    start->next = stop;
    start->prev = nullptr;
    stop->next = nullptr;
    stop->prev = start;
    start->merged = false;
    stop->merged = false;

    time = 0;
    somma = 0;
    window_size = n;
}
Sum::~Sum() {
    node* iter = start->next;
    while (iter != stop) {
        node* del = iter;
        iter = iter->next;
        delete del;
    }
    delete stop;
    delete start;
}
//Aggiunge un bucket e ritorna il suo puntatore
Sum::node* Sum::push(int size, int timest, node* insert, bool merged) {
    node* ins = new node;
    lenght++;
    ins->size = size;
    ins->timest = timest;
    ins->merged = merged;
```

```
    ins->next = insert->next;
    ins->prev = insert;
    insert->next->prev = ins;
    insert->next = ins;

    somma += size;
    return ins;
}
//Elimina un bucket e ritorna la sua timestamp
int Sum::pop(node* buck) {
    buck->prev->next = buck->next;
    buck->next->prev = buck->prev;
    somma -= buck->size;
    int ret = buck->timest;
    delete buck;
    lenght--;
    return ret;
}
void Sum::insert(int value) {
    if (value != 0) push(value, time, start, false);
    //wrap-around counter (dal momento che parte da 0,
    //viene resettato quando supera window_size)
    time++;
    if (time >window_size)time = 0;
    update();
}
void Sum::update() {
    //merge phase
    node* iter = start->next;
    int i = 1;
    //Early return se ci sono meno di 3 elementi
    while (iter != stop && i < 3) {
        iter = iter->next;
        i++;
    }
    if (iter == stop)return;
```

```
    int iter_sum = start->next->size;
    while (iter != stop) {
        if (k * (iter->size + iter->prev->size) <= iter_sum) {
            node* ins = iter->prev->prev;
            int newval = iter->size + iter->prev->size;
            //L'elemento a sinistra e' sicuramente piu' recente
            int a = pop(iter->prev);
            pop(iter);
            iter = push(newval, a, ins, true);
        }
        else iter_sum += iter->prev->size;
        iter = iter->next;
    }
    //Scarto dell'ultimo bucket se uscito dalla finestra
    if (stop->prev->timest == time) pop(stop->prev);
}
int Sum::eval() {
    //Viene tenuta la somma attuale di tutti i bucket attuali e poi
    //sottratta la meta' dell'ultimo. Se l'ultimo bucket non e'
    // frutto di un merge non viene effettuata l'approssimazione
    if (!stop->prev->merged) return somma;
    return somma - ((stop->prev->size) / 2);
}
```