UNIVERSITA' DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in Informatica

Tesi di Laurea

Flat Bloofi

Soluzioni "flat" per Bloom Filters multidimensionali

Relatore
Prof.ssa Rosalba Zizza

Candidato
Pierluigi Liguori

Matricola
0512104224

ANNO ACCADEMICO **2019/2020**

Indice

Elenco delle figure

Introduzione

La repentina evoluzione tecnologica ha delineato la struttura di sistemi informatici sempre più complessi. Il trend economico segue quelli che sono i modelli più all'avanguardia sull' elaborazione, l'acqusizione e l'archiviazione dei dati. Quest'ultimi non vengono adoperati necessariamente in contesti differenti, anzi, quasi nelle totalità delle applicazioni hi-tech, risultano essere interconnessi tra loro. Basti pensare all'**IoT (Internet of Things)**, grazie agli oggetti connessi alla rete, si riesce in modo costante a generare una enorme mole di dati. Il fenomeno **"Big Data"** è quindi diventato un argomento di interesse di molte aziende, le quali hanno posto l'attenzione sul come archiviarli ai fini di trarne il massimo profitto. Gestirli, però, è diventato sempre più complesso e costoso, naturale conseguenza è stata la nascita di ambienti cloud. Il paradigma del **Cloud Computing** si esplica su un'architettura distribuita; perciò è necessario che le comunicazioni tra nodi, tramite scambio di messaggi, siano più efficienti possibili. Questo è solo uno dei molteplici scenari in cui strutture dati di **Bloom Filters** giocano il proprio ruolo [6]. Un Bloom Filter è una struttura dati probabilistica (Il motivo della sua natura "probabilistica" verrà spiegata in modo dettagliato nei capitoli successivi) che, pur occupando una piccola quantità di memoria, sa essere molto potente soprattutto nel risolvere la classe di problemi che rispondono alla domanda: *"l'elemento x appartiene all'insieme S?*
L'efficienza dei Bloom Filters sta nel cosiddetto *"riferimento indiretto"*: durante una query di ricerca, l'oggetto da trovare in un dato insieme non viene confrontato direttamente con gli elementi della collezione. Bensì viene fatta una verifica di appartenenza ad un' astrazione di quelli che potrebbero essere i dati contenuti nell'insieme. Questa semplificazione fa sì che la computazione non sia troppo onerosa e che i costi per immagazzinare le informazioni relativi al funzionamento della query siano notevolmente ridotti. Ritornando nell'ambito distribuito, ora siamo in grado di immaginare un nodo all'opera durante un scambio di informazioni con un altro nodo. Sappiamo quanto la distanza possa notevolmente aggravare le prestazioni delle comunicazioni remote; ogni nodo potrebbe non

condividere l'intero set di informazioni, ma solamente la sua rappresentazione probabilistica, permettendo uno scambio di dati molto più veloce.
L'uso dei Bloom Filters non si limita a questo. Nel corso del tempo hanno avuto un'applicazione in molti altri campi. Nella Bioinformatica, la scienza che studia le metodologie per la risoluzione di problemi biologici per mezzo di metodi informatici, i Bloom Filters vengono usati per il sequenziamento del DNA. Alcuni Antivirus sfruttano le potenzialità di questa struttura dati per riconoscere codice malevolo su una lista di file; vengono usati inoltre nella compressione dei file, nella crittografia, nel web caching e in moltre altre applicazioni. La piattaforma Medium utilizza i Bloom Filters per evitare di consigliare articoli che un utente ha già letto in precedenza. Il browser Web Google Chrome li utilizza per identificare URL dannosi. Qualsiasi URL viene prima verificato con un Bloom Filter locale e solo se ha restituito un risultato positivo viene eseguito un controllo completo dell'URL. E' straordinario come una semplice definizione di una struttura matematica possa avere ripercussioni su una miriade di sfaccettature della nostra vita quotidiana. La domanda che si pone questo elaborato è:
Quali sono i limiti dei Bloom Filters? Quando il numero di Bloom Filters aumenta, una sbagliata agglomerazione di essi può portare al breakdown dell'intera struttura. Inoltre, rappresentare in un unico Bloom Filter troppe collezioni di informazioni non risulterebbe anch'esso efficiente. Se precedentemente il nostro focus si soffermava sul determintare se un elemento appartenesse ad una lista, ora dobbiamo porre l'attenzione su quali tra le possibili liste è più probabile che ci sia l'elemento in questione. Questo problema viene definito come *"The multidimensional Bloom filter problem"*. Secondo [3], le possibili soluzioni sono tre. Si propone per prima una soluzione di tipo naive (ingenua); la seconda prevede la costruzione di un albero in cui le foglie rappresentano i Bloom Filters da indicizzare, mentre la radice è il Bloom Filter che rappresenta tutti gli elementi nel sistema. Questa soluzione è chiamata **Bloofi**, implementata in C dal collega Fabiano Priore [7], traendone dei risultati interessanti da confrontare con quelli di questo elaborato. L'ultima soluzione è chiamata **Flat Bloofi**: una nuova struttura che sfrutta il parallelismo bit a bit intrinseco dei processori a 64 bit. Un Flat Bloofi viene definito come un array di interi dove ogni cella raggruppa un insieme di 64 bit e, ogni i-esimo bit dell'intero j, rappresenta l'i-esimo bit del j-esimo Bloom Filter. L' elaborato si baserà principalmente sullo sviluppo e sulla valutazione di quest' ultima alternativa.
Il primo capitolo fornisce un' introduzione ai Bloom Filters, spiegandone le proprietà, le operazioni usuali ed esempi più tangibili della loro applicazioni. Si

mostra inoltre quali sono i problemi relativi a quest'ultimi e l'esigenza di implementare *Bloofi* come prima soluzione, descrivendo anche qui le operazioni classiche. Il secondo capitolo introduce la definizione del *Flat Bloofi*, punto cardine dello studio in esame. Si analizza la sua composizione, le motivazioni dello sviluppo, le strutture dati che entrano in gioco, in che modo si effettua una ricerca di un elemento, un inserimento, cancellazione e aggiornamento di un Bloom Filter. Il Capitolo 3 vede un' implementazione Java degli algoritmi trattati in precedenza e successivamente una nostra implementazione C. Il Capitolo 4 si concentra sul tesing delle implementazioni e sulla effettiva efficienza, finendo per poi fare un confronto fra le due. Il testing prosegue comparando le prestazioni del Flat Bloofi e del Bloofi in C. Nel quinto capitolo si traggono le conclusioni sui vantaggi e svantaggi messi in evidenza da Bloofi e Flat Bloofi. Nel sesto ed ultimo capitolo si propongono ulteriori ottimizzazioni e possibili sviluppi futuri.

Capitolo 1

Dai Bloom Filters verso Bloofi

1.1 Nascita del Bloom Filter

1.1.1 La proposta

Il Bloom Filter viene proposto per la prima volta nel 1970 da Burton Howard Bloom nel suo saggio *Space/Time Trade-offs in Hash Coding with Allowable Errors* [1], introducendo un nuovo metodo per codificare valori hash in modo efficiente ma con il rischio tollerabile di incappare in errori. La formula di Howard fornisce un compromesso tra percentuale falsi positivi, spazio e tempo. Successivamente la struttura venne chiamata Bloom Filter. La struttura dati, come già accennato nell'introduzione, è utile per testare se un elemento appartiene ad un insieme oppure no. Il Bloom Filter, dopo aver recepito la query, risponde con due possibili esiti: *'Sicuramente l'elemento non è presente nell'insieme'* oppure *'può darsi che l'elemento non appartenga all'insieme'*. Ovviamente, più elementi vengono aggiunti all'insieme e maggiore sarà la probabilità di ottenere un falso positivo. Più tardi venne introdotta una nuova versione in cui, oltre l'inserimento e la ricerca, fosse possibile effettuare anche una rimozione dell'elemento; questa variante è definita **Counting filters** [5].

1.1.2 I vantaggi

Il primo vantaggio evidente, accettando il rischio di ricevere falsi positivi, è sicuramente lo spazio occupato dall'intera struttura, il quale risulta di un peso minore rispetto ad altre strutture dati come array, liste, alberi e tabelle hash. *Il riferimento indiretto* all'elemento, fai in modo che il peso dell' oggetto in questione non gravi sul peso dell'intera struttura; perciò il Bloom Filter è indipendente

dalla dimensione degli elementi che indicizza; ne consegue che il tempo per aggiungere o verificare la presenza di un elemento è dovuta al numero k di funzioni hash all'interno del Bloom, quindi ha complessità $O(k)$. Per avere un' idea più concreta dell' effettivo vantaggio, si tenga conto che, settando una probabilità di errore all' 1% ed un valore ottimale di k funzioni hash, lo spazio richiesto per un elemento non raggiunge i 10 bit. Si noti inoltre che allocando circa 15 bit per elemento, la percentuale di errore scende allo 0.1%.
Ovviamente, adottare la scelta di un Bloom Filter non fornisce un rendimento massimo in ogni situazione. Un classico array di bit, ad esempio, richiede solo un bit per ogni potenziale elemento, quindi risulta più efficiente rispetto ad un Bloom che ha un basso numero di potenziali valori; un Bloom con numero di funzioni hash $k = 1$ assume un comportamento identico alle tabelle hash.

1.2 Operazione di inizializzazione, inserimento e ricerca

Diamo in input un insieme $A=\{a_1, a_2, ..., a_n\}$ di n elementi. Vogliamo effettuare l'inserimento di essi nel Bloom Filter: lo stato inziale prevede un array L di m bits tutti settati a 0 e la scelta di k funzioni hash in grado di mappare gli elementi in modo randomico e uniforme sull'intervallo $1...m$. Denotiamo le applicazioni delle funzioni sull'elemento 'a' con $h_i(a)$ per $1 \leq i \leq k$.
Un elemento $a \in S$ viene inserito nel Bloom Filter settando a 1 le celle dell'array L che sono indicizzate dal risultato delle applicazioni $h_i(a)$ per $1 \leq i \leq k$.

Data: x is the object key to insert into the Bloom filter.
Function: *insert(x)*
for $j : 1 \ldots k$ **do**
    /* Loop all hash functions k */
    $i \leftarrow h_j(x)$;
    if $B_i == 0$ **then**
        /* Bloom filter had zero bit at position i */
        $B_i \leftarrow 1$;
    end
end

Figura 1.1: Pseudocodice per l'inserimento di una parola del Bloom Filter.

Prima di illustrare l'algoritmo di ricerca, si tenga conto che i possibili esiti della query sono: *"l' elemento non appartiene all'insieme"* oppure *"l'elemento potrebbe appartenere all'insieme"*. Il test di appartenenza di b avviene calcolando le applicazioni $h_i(b)$ per $1 \leq i \leq k$: in questo modo vengono forniti k indici che faranno riferimento alle celle dell' array L. Se vi è anche una sola cella che ha valore 0, allora si ha la certezza che l'elemento non è presente nell'insieme A. Altrimenti, se tutti i bit controllati sono settati a 1, possiamo evincere che:

1. L'elemento b è contenuto nell'insieme A.
2. Oppure parte di tali bit risultano settati a 1 per mezzo di altri elementi inseriti in precedenza, questo ne consegue la presenza un falso positivo.

Data: x is the object key for which membership is tested.
Function: *ismember*(x) returns true or false to the membership test

$m \leftarrow 1$;
$j \leftarrow 1$;
while $m == 1$ *and* $j \leq k$ **do**
  $i \leftarrow h_j(x)$;
  if $B_i == 0$ **then**
    $m \leftarrow 0$;
  end
  $j \leftarrow j + 1$;
end
return m;

Figura 1.2: Pseudocodice per la ricerca di un elemento in un Bloom Filter.

Ovviamente, se si incrementa il numero di elementi contenuti dall'insieme, la probabilità di falsi positivi cresce. Aumentare il numero di funzioni hash implicherebbe un aumento del costo della computazione ed una conseguente diminuzione della probabilità di falsa positività. L' aumento invece della dimensione del Bloom Filter invece, porta ad abbassare la percentuale di falsi positivi.

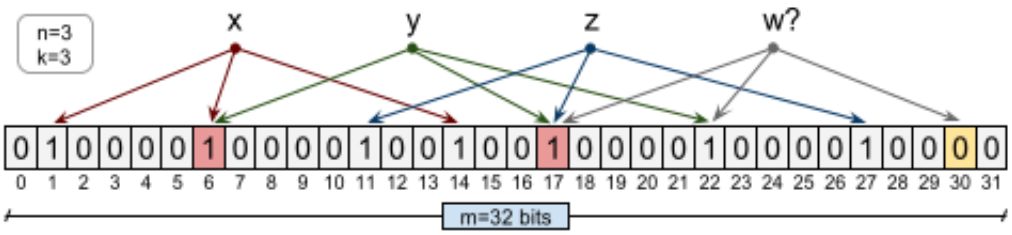

Figura 1.3: La ricerca di un elemento in un Bloom Filter.

1.3 Probabilità del falso positivo

Non esiste quindi un modo per distinguere un esito positivo da un falso positivo, possiamo però calcolare la probabilità che l'errore accada. Considerando m come il numero bit di un dato Bloom Filter, la probabilità che un bit qualsiasi abbia valore 1 è banalmente:

$$\frac{1}{m}.$$

La probabilità contraria, ossia, la possibilità che un bit sia impostato a 0 è perciò:

$$1 - \frac{1}{m}.$$

Se considerassimo tale risultato per tutte le k funzioni hash applicate al singolo elemento:

$$\left(1 - \frac{1}{m}\right)^k$$

Ed estendendo per tutti gli elmenti n inseriti, otteniamo:

$$\left(1 - \frac{1}{m}\right)^{kn}$$

Infine, la probabilità che un generico bit, dopo le k applicazioni per n elementi, sia 1 è:

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

Quando viene eseguita una query di ricerca, l'elemento viene dato in input alle k funzioni hash, quindi il risultato precedente deve essere considerato per tutti

gli indici restituiti da $h_i(a)$ per $1 \leq i \leq k$. Perciò la probabilità che tutti i bit del nuovo elemento sono impostati su 1 è:

$$P = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

P è la probabilità di falsa positività che l'elemento appartenga all'insieme, approssimabile al risultato di fianco. Si può notare come la probabilità di falso positivo diminuisce all'aumentare di m, ossia la dimensione del Bloom Filter, aumenta invece aumentando il numero elementi inseriti (ossia n). Lo step successivo è quello di ottimizzare la percentuale minimizzandola rispetto a k. Se si uguaglia la derivata a 0, otteniamo il valore ottimo di k:

$$k_{opt} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}$$

Sostituendo k_{opt} nell'equazione precedente, si ottiene:

$$P = \left(\frac{1}{2}\right)^k \approx 0.6185^{m/n}$$

Per convenzione, un Bloom Filter viene inizializzato con P prefissato, con il quale si ricavano il resto dei parametri. Fissando anche il numero di elementi n, m sarà:

$$m = -\frac{n \ln P}{(\ln 2)^2}$$

1.4 Cenni di applicazioni

Come già accennato nell' introduzione, i Bloom Filters hanno un ruolo centrale negli ambienti cloud. Nel particolare, sono utili per condividere informazioni *web cache* [10] . Si consideri un sistema distribuito formato da un insieme di Proxy; ogni nodo condivide un proprio Bloom Filter che rappresenta dati cache locali in modo compatto. Se un Proxy non trova un determinato dato cache, cerca se il dato è contenuto in un altro nodo consultando una *"Bloom Filter bank"*, inoltrando una richiesta al medesimo se possiede tale risorsa.
Un discorso analogo viene applicato per Rintracciare IP [9], magari per intercettare il mittente di pacchetti dannosi. L'obiettivo è quindi delineare il percorso intrapreso dal pacchetto su vari router nella rete. E' necessario consultare su

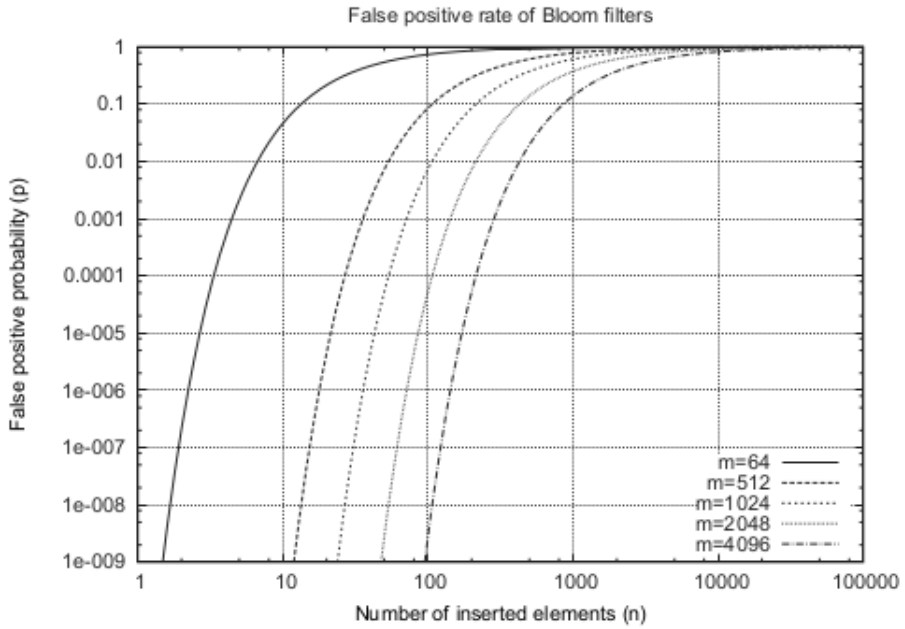

Figura 1.4: rappresentazione di P in funzione di n ed m.

ciascun router per dedurne le tracce, purtroppo è uno sforzo oneroso, ma, se la consultazione si limitasse ad un Bloom Filter, sarebbe possibile eseguire una query di ricerca in un tempo accettabile. Ovviamente questo non evita l'incappare di un falso positivo, il quale significherebbe immettere un router sbagliato nel percorso e quindi una generazione di falsi cammini. Limitare P ad una percentuale bassa aumenterebbe il grado di affidabilità.

1.5 Bloom Filters multidimensionali

1.5.1 Panoramica

L'applicazione costante del paradigma distribuito negli ultimi tempi fa spesso uso di Bloom Filter; abbiamo visto diversi esempi e sembra che il trend sia in continua crescita. A pari passo con lo sviluppo di nuove infrastrutture, cresce esponenzialmente il numero di dati da immagazzinare. Supportare tecnologie basate su Bloom Filters diventa via via sempre più insostenibile: se inizialmente il problema fosse cercare una determinata informazione tra nodi, Proxy, database che contenevano insiemi di dati, ora la criticità è nel cercare la macchina che contiene l'insieme che ospita l'oggetto in questione, una sorta di "meta-problema". Questo dilemma è definito come *"The multidimensional Bloom Filter problem"*.

Si pensi a grandi applicazioni web finanziarie, dove una miriade di dati vengono creati, tracciati, modificati, inoltrati. In un contesto cloud, ogni postazione dovrebbe custodire il proprio banco di dati, astrarlo con un Bloom Filter e condividerlo con una sede centrale. Deve esistere un metodo per poter rappresentare tutte le astrazioni di ogni postazione con una struttura matematica rigorosa, in modo che la sede centrale possa avere una consultazione veloce, senza dover cercare ingenuamente su ogni singolo Bloom Filter di ogni postazione.
Indicizzare diversi Bloom Filter non è come indicizzare un generico oggetto a causa del *'riferimento indiretto'* tra gli elementi del set, poiché ciò che rimane è 'solo' una rappresentazione compatta di quello che è l'insieme. Ciò che si sta cercando è quindi un link al Bloom Filter che potrà contenere l'elemento interessato. Sono ancora rari i casi in cui un sistema distribuito non regga l'infrastruttura creata con approccio probabilistico ma considerando che l'adozione di tale euristica è in continuo aumento, diviene di grande importanza affrontare eventuali colli di bottiglia. Per questo motivo è stato necessario introdurre una nuova struttura dati che indicizzi i Bloom Filters in modo rigoroso, chiamato **Bloofi**.

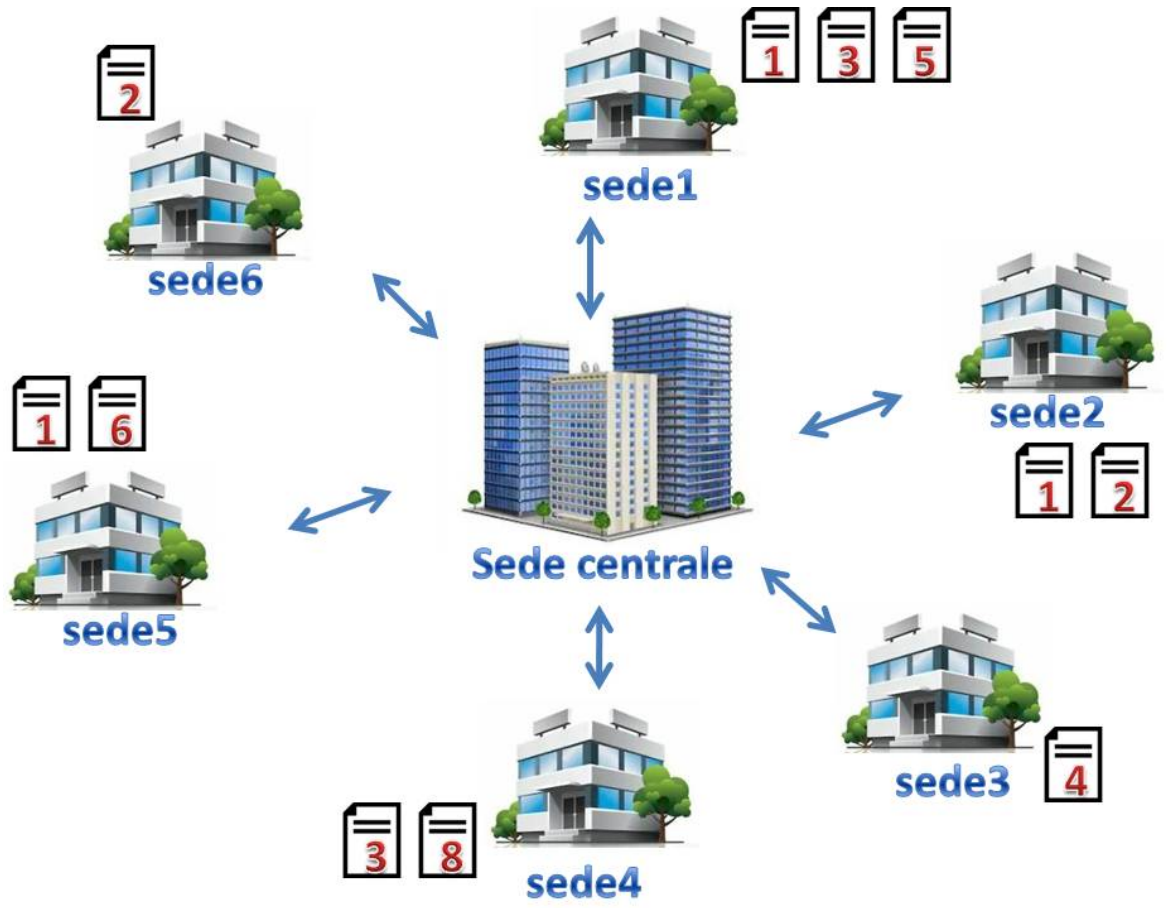

Figura 1.5: Contesto di applicazione del Bloofi [7].

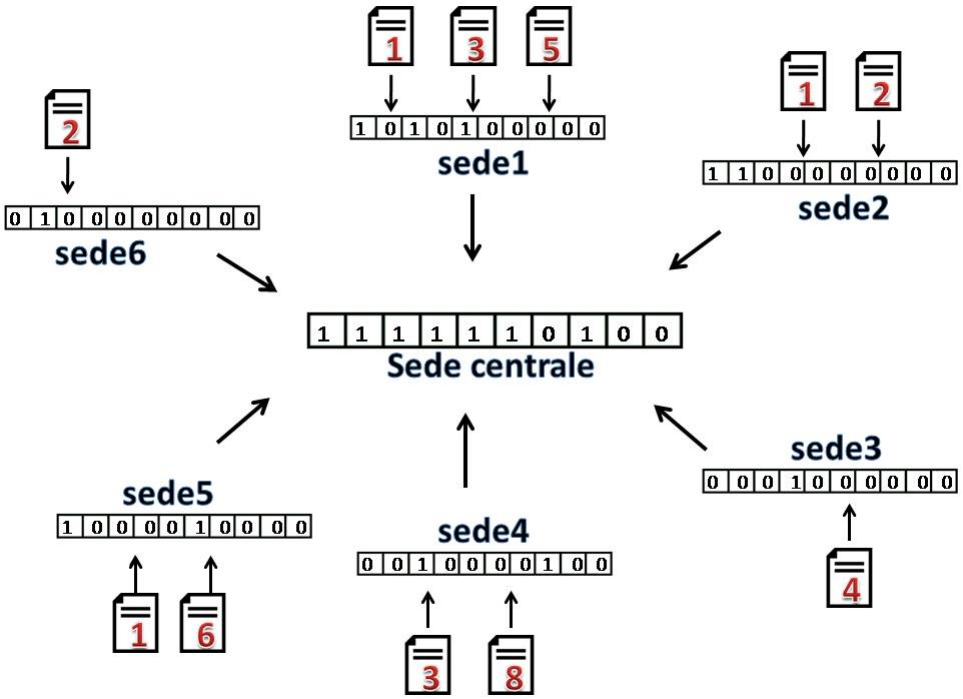

Figura 1.6: Conversione del contesto di applicazione in Bloom Filters [7]

1.5.2 Cos'è Bloofi

Bloofi è una struttura gerarchica nata con lo scopo di indicizzare molteplici Bloom Filters. La gerarchia deriva dalla natura ad albero della struttura dati: le foglie rappresentano i Bloom Filters da indicizzare. I nodi interni, padri delle foglie, sono generati invece dall'applicazione dell' OR bit a bit sui nodi di ciascun figlio. Quindi, ogni nodo che non sia foglia non è altro che l' unione degli insiemi rappresentati dai Bloom Filters appartenenti al sottoalbero che ha come radice appunto il nodo considerato. Per capire la potenzialità del Bloofi, bisogna analizzare l'operazione di ricerca di un elemento: l'elemento x deve essere contenuto in uno dei set rappresentati dai Bloom Filters, i quali sono strutturati come nodi di un albero. Un 'match' si verifica quando, applicando tutte le k funzioni hash indicate su quell'oggetto, trova k bit settati a 1 sul Bloom Filter sul primo nodo considerato. Se la risposta è negativa, allora ho la certezza che l'elemento non appartiene a nessuno degli insiemi rappresentanti dai Bloom Filters posizionati nel sottoalbero del nodo di partenza. Se la risposta è positiva, l'operazione di ricerca continua cercando nei relativi sottoalberi rappresentati dai nodi figli che hanno ottenuto un match con l' elemento x. La ricerca parte dalla radice di tutta la struttura Bloofi. Se l'albero è bilanciato allora la percor-

renza della sua altezza richiede una complessità logaritmica. Tirando le somme, nel migliore dei casi Bloofi dà risposta in tempo costante, semplicemente perchè non trova nessuna corrispondenza già dalla radice e restituisce risposta negativa. Altrimenti, una risposta positiva (o addirittura pseudo-positiva) significherebbe visitare parte dell' albero e si impiegherebbe tempo logaritmico. Nel peggiore dei casi si scorrerebbe tutta la struttura fino alla foglia. Inoltre non è da escludere la possibilità di trovare un falso positivo e quindi di dover percorrere percorsi sbagliati. Tuttavia, inserendo Bloom Filters simili nello stesso sotto-albero, il problema può essere attenuato.

1.6 Struttura di un Bloofi

L' implementazione del Bloofi prede una struttura gerarchica, la quale può variare dal tipo di albero scelto. Un albero AVL (alberi bilanciati) è tale se per ogni nodo x l'altezza del sottoalbero sinistro di x e quella del sottoalbero destro di x differiscono al più di uno e entrambi i sottoalberi sinistro e destro di qualsiasi nodo sono ancora alberi AVL, quindi un AVL è bilanciato in altezza. Un B+ [8], invece, custodisce i dati solamente nelle foglie ed usa i nodi interni solamente per ospitare delle chiavi utili nelle query di ricerca. Un albero è di ordine m se ogni suo nodo interno ha al massimo m figli, ne consegue che il numero massimo di valori chiave da memorizzare in un nodo è m. Con *half full*, invece, si intende la proprietà del nodo ad avere un numero di figli compreso tra $m/2$ ed m. Un B+ deve essere half full e i nodi foglia devono trovarsi tutti allo stesso livello. E' proprio per la flessibilità dovuta al suo bilaciamento che esso viene scelto per lo sviluppo del Bloofi. Nella Figura 1.7 viene mostrato un esempio di un Bloofi. Nelle foglie sono memorizzate i valori dei Bloom Filter da indicizzare, ed hanno rispettivamente gli identificatori 1,2,3,4,5,6. Il valore invece dei nodi interni si ottiene applicando l'OR di 1,2,3,4 per il nodo con Id = 7 e l'OR bit a bit di 5,6 per il nodo con Id = 9.

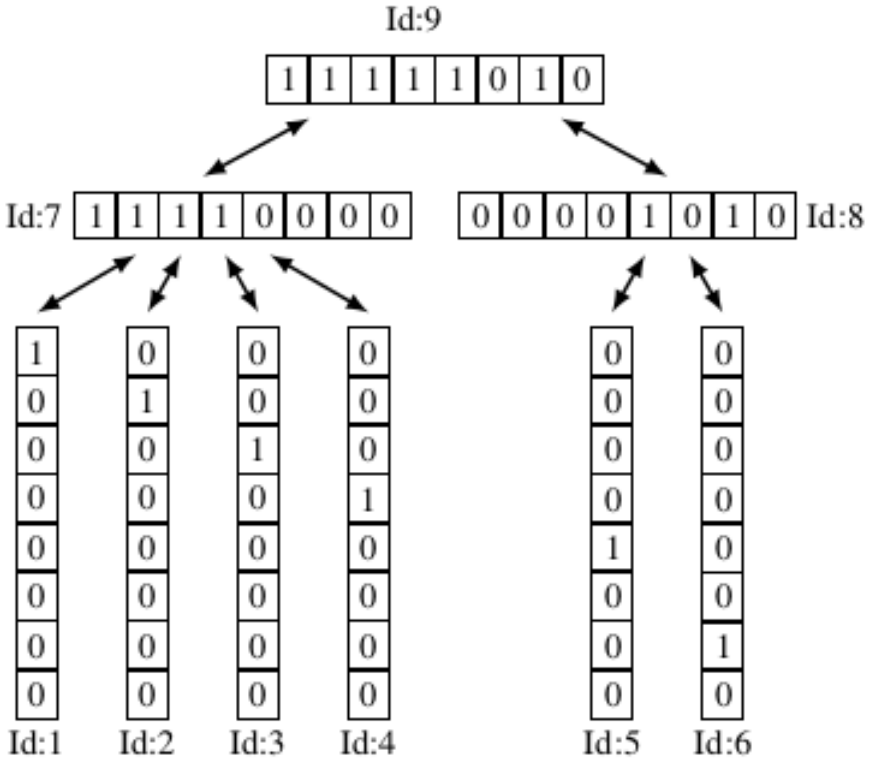

Figura 1.7: Bloofi di ordine 2.

1.7 Operazioni usuali

Le operazioni per un Bloofi sono quattro. Ricerca, inserimento, cancellazione, aggiornamento. La ricerca, si preoccupa di verificare se un elemento appartiene ad un insieme rappresentato da uno dei Bloom Filter del Bloofi. L' inserimento è utilizzato per costruire il Bloofi, aggiungendo Bloom Filter alla struttura. La cancellazione è più rara, viene rimosso un Bloom Filter indicizzato. Infine, l'aggiornamento si riferisce alla modifica dei Bloom Filter all'interno del bloofi, magari a causa di un inserimento di un nuovo oggetto.

1.7.1 Ricerca

L'operazione restituisce gli ID dei nodi foglia, contenenti i Bloom Filter che hanno un match con l' elemento (*elem*) passato come parametro alla query. Quindi la funzione prende in input $< node, elem >$, dove con *node* si intende il nodo dove iniziare la ricerca, di solito il nodo di partenza è proprio la radice.
L' algoritmo controlla inizialmente se il nodo di partenza ha un match con *elem*,

altrimenti nessun altro Bloom Filter del sottoalbero avrà un match con l' elemento. Se il match avviene, abbiamo due casi: se il nodo è una foglia, allora viene restituito l'ID del Bloom Filter, se trattasi di un nodo interno alla si riapplica ricorsivamente l' operazione ad ognuno dei suoi figli.

Algorithm 1 : `findMatches`(node,o)

```
1: //RETURN VALUE: the identifiers of leaves in the sub-
   tree rooted at node with Bloom filters matching the ob-
   ject o
2: //if node does not matches the object, return empty set,
   else check the descendants
3: if not match(node.val,o) then
4:     return ∅;
5: else
6:     //if this node is a leaf, just return the identifier
7:     if node.nbDesc = 0 then
8:         return node.id;
9:     else
10:        //if not leaf, check the descendants
11:        returnList = ∅;
12:        for i = 0; i < node.nbDesc; i++ do
13:            returnList.add(findMatches(node.children[i],o));
14: return returnList;
```

Figura 1.8: Algoritmo di ricerca per un Bloofi.

Applichiamo l' algoritmo alla Figura 1.7. Diamo in input i paramtri $< node9, 4 >$ (vogliamo cercare l'elemento 4), partiamo dal nodo con ID = 9, quindi la radice. Si verifica il match del valore 4 in $node9$: la funzione hash applicata a '4', $h(4)$, indirizza ad una cella dell'array del bloom fiter, che risulta con valore 1. Quindi l'algoritmo prosegue ricorsivamente sui nodi figli, ossia $< node7 >$, e nodo $< node8 >$. Su $< node7 >$ non vi è alcun match e viene ignorato. Si trova invece un match su $< node8 >$, quindi si prosegue sul sottoalbero sottostante. Il primo nodo figlio è $< node5 >$, nonché una foglia. Il match avviene e viene quindi restituito dall' algoritmo ID = 5. Si prosegue sul secondo figlio, $< node6 >$, che non dando un match viene ignorato.
Abbiamo concluso che $search(< node9, 4 >) = 5$.
Parlando di complessità, se venisse trovato un match in un Bloom Filter foglia,

il numero di chiamate a *search* è $O(d \log_d N)$, dove d è il minimo numero di figli che ha un nodo interno. Nel caso in cui tutti i nodi venissero acceduti, quindi nel caso peggiore, si avrà una complessità $O(N)$.

1.7.2 Inserimento

La strategia dell' operazione di *insert* è quella di inserire il Bloom Filter nuovo nelle vicinanze del nodo che contiene il Bloom Filter a cui 'assomiglia' maggiormente. La somiglianza è dedotta da una metrica usata, come ad esempio la distanza di Hamming, la quale conta il numero di bit dei quali differiscono. Il Bloom Filter dato input verrà inserito accanto alla foglia più somigliante. Come già accennato, questa accortenza velocizza le eventuali operazioni di ricerca. Il Bloom da inserire inizia l'operazione facendo un OR bit a bit del suo valore con quello del nodo di partenza; la funzione ricorsivamente prosegue richiamandola nel nodo figlio (del nodo corrente) "più simile". La ricorsione finisce quando si arriva alla foglia più somigliante, il nuovo Bloom verrà incapsulato in un nodo che sarà fratello alla foglia. Può capitare una sitazione non tanto rara: il padre della foglia può aver raggiunto il massimo numero di figli. In tal caso è necessario effettuare una operazione di *Split*. Ossia dividere il nodo appena creato e farlo restituire dalla funzione. Notasi che questa operazione può ricorsivamente capitare per tutta la lunghezza dell' albero fino ad arrivare alla radice. Se questo accade, bisogna aumentare l'altezza dell'albero.
Si prenda in considerazione la Figura 1.7. Si vuole inserire il Bloom Filter con valore "00100100". Eseguendo $Insert(< node9, NewBloom >)$, il nodo più simile che incontra è $< node7 >$, purtoppo il numero di figli del nodo 7 è 3, ossia il limite massimo; per questo motivo si verifica uno split. Il nuovo Bloom Filter verrà posizionato come mostrato nella Figura a seguire. L' operazione di inserimento ha complessità $O(d \log_d N)$, dove d è l'ordine e N è il numero di Bloom Filter da indicizzare.

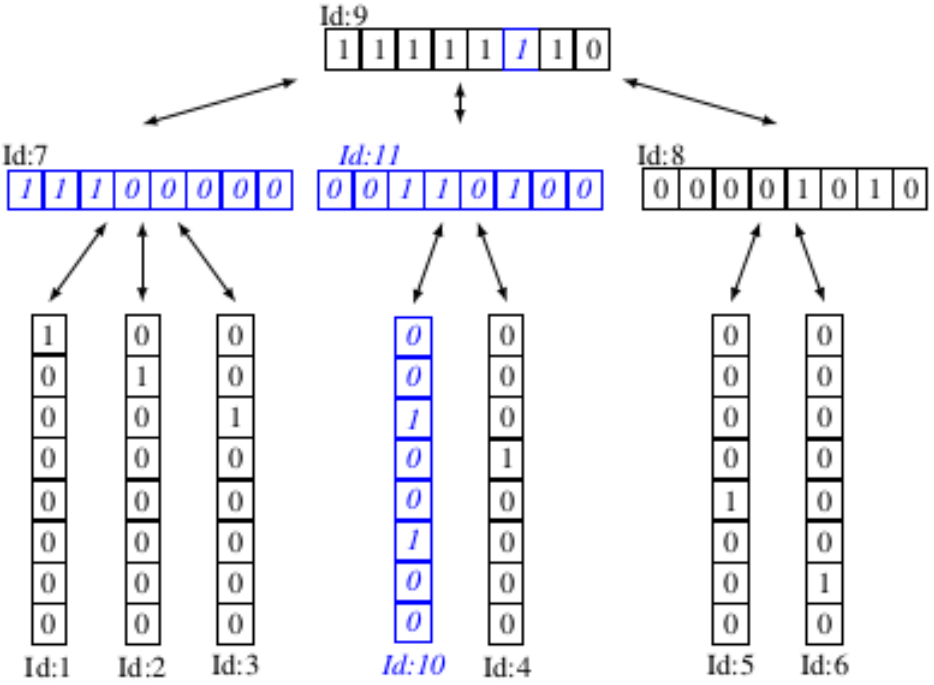

Figura 1.9: Operazione di inserimento.

1.7.3 Cancellazione

L' algoritmo di eliminazione consiste nel rimuovere il nodo che ha l'indice specificato come parametro dall' operazione. La chiamata di *remove* su una foglia, prevede l' eliminazione del puntatore del padre verso la foglia in questione. C'è da precisare che: se il nodo padre non va sotto il minimo numero di figli che un nodo deve avere, si deve eseguire un OR per tutti i Bloom Filter che seguono il cammino dal padre alla radice. Se capitasse il contrario (si definisce underflow) il padre redistribuirà i suoi figli con suo fratello, affinché il tutto rimanga ancora bilanciato. Questo comporta un aggiornamento dei valori nei Bloom Filter fratelli e ricorsivamente anche su tutto l'albero. Laddove la redistribuzione non sia possibile, si procede con uno split: viene fuso il nodo padre con il suo relativo fratello, cedendogli le entrate. Se questa operazione si propaga su tutti i nodi fino alla radice, lasciandola con un solo figlio, allora l'altezza dell'albero deve esere diminuita.
Nella Figura 1.7, si applica una rimozione del nodo 5, con valore "00001000". L'albero viene redistribuito tra il nodo 7 ed il nodo 8, il risultato ottenuto viene mostrato nella Figura 1.7. La rimozione di un Bloofi ha complessità $O(d \log_d N)$.

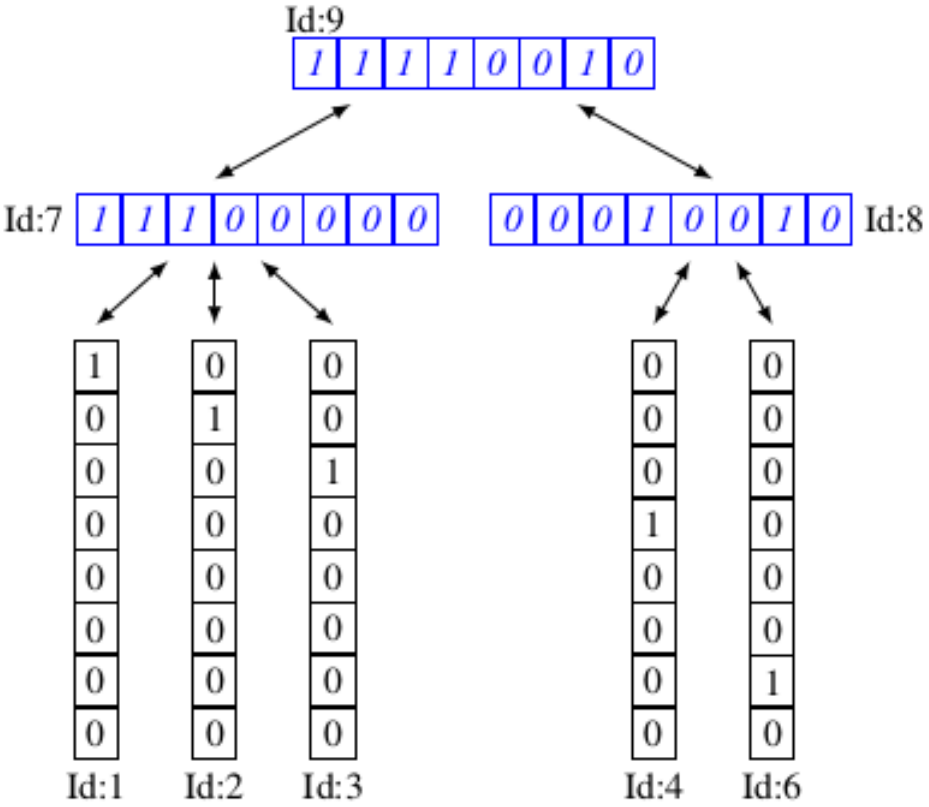

Figura 1.10: Bloofi dopo cancellazione e redistribuzione

1.7.4 Aggiornamento

Si è detto che l' operazione di aggiornamento è molto frequente. Prende in input $< nodoX, NuovoBloomFilter >$, ossia il nodo da modificare ed il nuovo Bloom Filter che deve incapsulare. Ovviamente, il valore dei nodi che seguono il cammino dalla radice al $nodoX$ devono essere aggiornati per mezzo di un OR con il nuovo valore. La Figura 1.11 mostra l'aggiornamento di $nodo6$ con il nuovo Bloom Filter con valore "00000011". La complessità di aggiornamento è $O(\log dN)$.

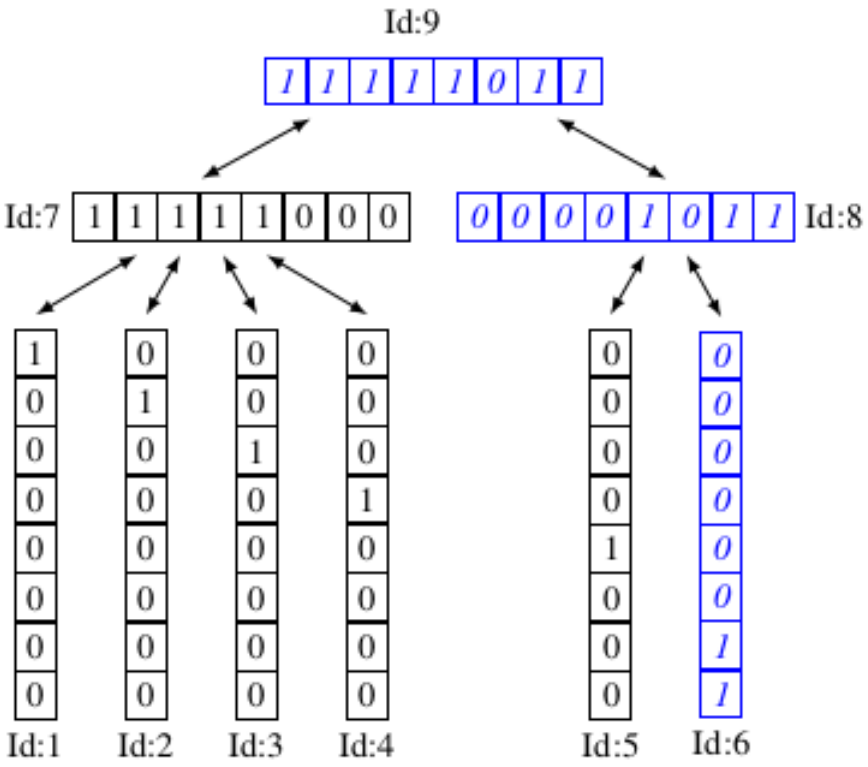

Figura 1.11: Bloofi dopo un aggiornamento.

Capitolo 2

La soluzione "flat"

2.1 Motivazioni

Una prima proposta alla risoluzione del problema dei Bloom Filters multidimensionali è il Bloofi. La sua struttura ad albero riesce a sostenere un carico di oltre 10000 Bloom Filters indicizzati senza troppi sforzi. La soluzione fornisce sicuramente un rendimento notevole se si considera il problema che si è posto all' inizio dell'elaborato, ossia gestire un alto numero di indicizzazioni. Se in questo contesto Bloofi è la scelta ottimale, non possiamo non considerare casistiche in cui il focus si sposti anche su aspetti come l'aumento degli elementi per ciascun Bloom Filter, la sua dimensione, il costo di memorizzazione, l'aumento della falsa positività o la preferenza di operazioni rispetto ad altre. Si analizzi il seguente scenario: vengono caricati sul Bloofi un basso numero di Bloom Filter (nell'ordine delle centinaia) e impostata una probabilità di incappare in falsi positivi maggiore di 1. Nel peggiore dei casi, Bloofi sarebbe costretto a cercare in sottoalberi sbagliati, poiché la percentuale di errore lo porta a seguire cammini sbagliati. Con queste premesse, il tempo di ricerca di un Bloofi può risultare addirittura peggiore rispetto ad una soluzione naive, dove i Bloom Filters sono indicizzati da una semplice lista. Questo è uno dei motivi per i quali a volte si predilige un Flat Bloofi.

2.2 Il parallelismo tra bit

Un ulteriore motivo per preferire una soluzione "flat", sta nel non sfruttare una potenzialità che fin ora non è entrata gioco, ossia il possibile *parallelismo tra bit*; un aspetto che vuole marcare di più un carattere fisico anziché logico. Il grado di parallelismo di una CPU indica il numero di bit elaborati contemporaneamente

dal processore. Un' architettura a 64 bit avrà generalmente i registri del processore larghi 64 bit e gestisce dati di questa dimensione. Questo significa che in una sola istruzione, ad esempio un banale OR bit a bit, vengono elaborati contemporaneamente tutti e 64 i bit. Flat Bloofi viene costruito ad hoc per usufruire questa peculiarità delle moderne CPU. L'uso dell' architettura x64 è stata scelta perché è quella più in uso nel mercato, si tenga conto che un processore a 128 bit potrebbe anche migliorare la struttura dati, ma non converrebbe a livello di costi considerando che con i processori attuali si riesce con 64 bit ad indicizzare già miliardi di miliardi di interi, ed indicizzare un numero infinitamente superiore usando CPU x128 sarebbe uno spreco. Il parallelismo dunque segue il principio di *località* della memoria: gli accessi sono raggruppati nel tempo e nello spazio, portando un netto miglioramento delle performance.

2.3 Struttura Flat Bloofi

In [3] propone una nuova struttura dati, chiamata Flat Bloofi, in grado di memorizzare l'equivalente di 64 Bloom Filters. Ogni Bloom Filter è composto da un Bitset, un array di m bit (che possono assumere valore 0 o 1). Per costruire un Flat Bloofi, viene istanziato un array di interi a 64 bit (si utilizza un processore x64): il primo intero corrisponde a tutti i primi bit di ciascun Bitset dei Bloom Filter da indicizzare, il secondo intero ai secondi bit e così via. Generalizzando *il valore dell'i-esimo bit del j-esimo Bitset corrisponde al valore dei j-esimo bit dell'i-esimo intero dell'array del Flat Bloofi.*. Ragionando in quest'ottica, un Flat Bloofi riesce a memorizzare un massimo di 64 Bloom Filters, perciò, se si volessero aggiungere altri Bloom Filters, bisognerebbe istanziare un nuovo Flat Bloofi, quindi un nuovo array di interi. Quindi, dati N Bloom Filters, teoricamente servirebbero $N/64$ flats. Non è permesso istanziare flat minori di 64 celle, perciò, se N non fosse multiplo di 64, inevitabilmente alcuni bit dell'ultimo flat non verrebbero usati.
La composizione di un Flat, avviene grazie all'ausilio delle seguenti strutture:

- Z Flat Bloofi, con i quali siamo in grado di indicizzare $L = Z \times 64$ Bloom Filters.
- Un array β di dimensione L, avente N celle settate a 1. Un set a 1 sta indicare che la locazione è usata da un Bloom Filters. Se un Bloom Filters viene rimosso, la cella viene risettata a 0, oppure 1 se viene rioccupata.

- Una tabella hash che mappa gli ID dei Bloom Filters ad un indice di una cella di β, quindi nel range $[0, L)$. Viene memorizzato anche anche un array di ID di lunghezza L affinché si possa gettare l'ID di un Bloom Filter dato un indice come input. Combinando la tabella hash assieme a quest'ultimo, si ottiene una doppia indicizzazione per la localizzazione delle posizioni dei Bloom Filters.

La Figura 2.1 mostra la struttura logica di un Flat Bloofi. Per convenzione, chiameremo Flat Bloofi l'intera struttura e con "flat" tutti i singoli array di interi che ne faranno parte. Quindi il modello di sotto mostra un solo flat. Esso è composto da 64 celle di interi (15303, 191, ..., 252.): si può notare come ogni cella, essendo un intero, abbia racchiusi in sé 64 bit, i quali sono suddivisi logicamente e il loro valore posizionale è il decimale rappresentato dall'intero. Al di sotto del flat, vi sono 64 Bloom Filters (per questione di spazio ne sono raffigurati solamente 4). Il Bloom più a sinistra è il primo e quello più a destra è il sessantaquattresimo. La rappresentazione raffigura come due hasher selezionano la cella 0 e la cella 62 del Flat, quindi selezionano automaticamente tutte le celle 0 e le celle 62 di ciascun Bloom Filter.

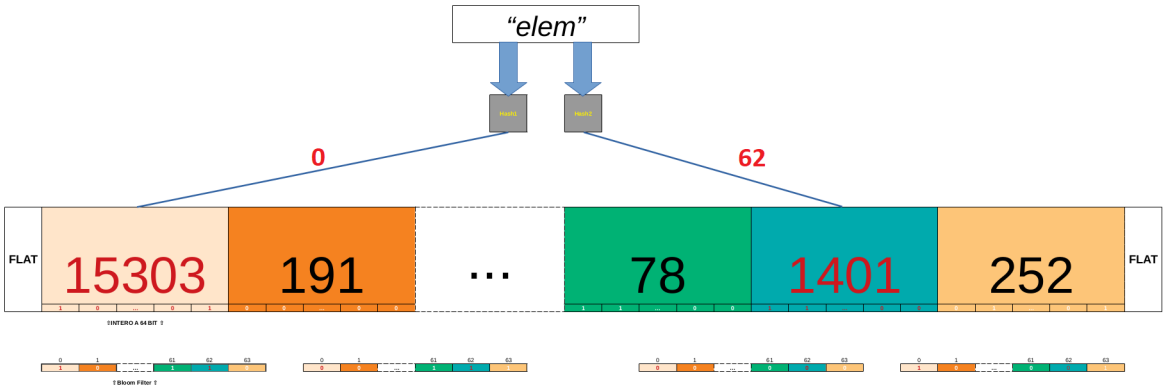

Figura 2.1: Esempio di un Flat Bloofi.

2.4 Operazioni

2.4.1 Inizializzazione

Lo stato iniziale consiste nella creazione di un Flat, composto da 64 celle di interi, tutte inizializzate a 0. Vale lo stesso per l'array β e per l'hash map, istanziata e lasciata vuota. Viene scelto un Hasher, il quale si preoccuperà di scegliere k funzioni hash.

2.4.2 Ricerca

Si vuole verificare la presenza di un elemento memorizzato in uno o più Bloom Filter indicizzati dai Flat Bloofi, perciò la funzione prende come parametro $< elem >$ e restituisce la lista degli ID dei Bloom Filters che contengono $elem$. L'algoritmo viene eseguito nel seguente modo: $elem$ viene dato in input per ognuna delle funzioni hash dell'hasher, quindi si esegue $H_i(< elem >)$ per ogni $i, ..., k$. I k valori risultanti, indicati con $h_1, ..., h_k$, hanno il compito di indicizzare k celle per ogni Flat. Perciò, per ogni array di interi (per ogni flat) F_i, vengono marcate $F_i(h_1), ..., F_i(h_k)$ e ne viene fatto l'AND bit a bit. La computazione restituisce un nuovo valore W_{fj} per ogni j-esimo flat. Se tutti i W_{fj} danno un valore pari a 0, allora vuol dire che non esistono Bloom Filter che hanno come ID l'elemento $< elem >$. Altrimenti, per ogni j, vengono scelte le posizioni di tutti i bit settati ad 1 di W_{fj}, le quali indicheranno i probabili numeri ID dei Bloom Filters cercati. Le posizioni hanno uno spiazzamento di 64 moltiplicato il numero del flat che ha generato W_{fj}.
Si descrive l'algoritmo come mostrato in Figura 2.2: viene cercata la parola "elem", che, computato con le 2 funzioni hash, si ottengono 0 e 62, valori che usiamo per indicizzare le celle di ciascun flat (anche se in tal caso il flat è solamente uno). Gli interi corrispondenti sono W_{11}, W_{12} per il primo e unico flat, che hanno valore 1503 e 1401. In genere, per ogni $k - tupla$ di ciascun flat, ne viene fatto l'AND: questa operazione mette in evidenza il punto di forza della struttura dati, ossia il parallelismo tra i bit; computare l'AND bit a bit dei vari W_{fj}, dà la possibiltà di verificare in modo "istantaneo" e simultaneo la presenza di $< elem >$. Nell'esempio, l'unico valore non nullo è $W_{\text{f1}} = 100..00$, ed ha valore 1 nella prima posizione, chiaro riferimento al Bloom Filter indicizzato con 0. Quindi "Elem" si troverà probabilmente nel primo Bloom Filter, quindi in quello che avrà l'ID = 1.

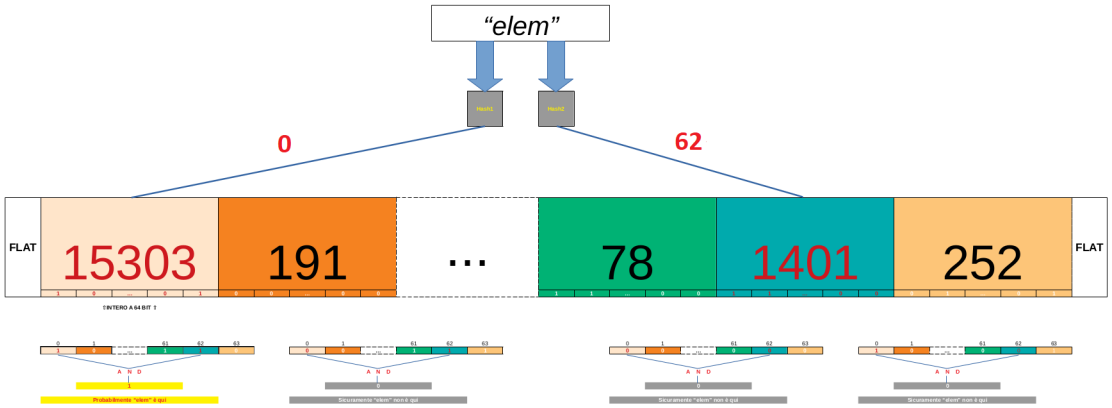

Figura 2.2: Operazione Search in un flat.

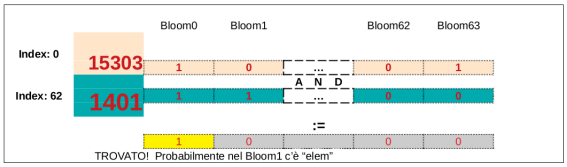

Figura 2.3: Operazione Search in un flat nel dettaglio.

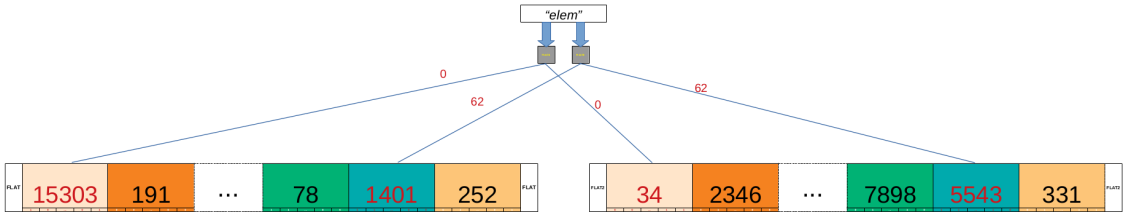

Figura 2.4: Esempio di molteplici flat.

2.4.3 Inserimento

Lo scopo è quello di inserire un nuovo Bloom Filter nella struttura. Viene fatto scorrere l'array β, finchè non legge 0, ossia finché non si trova uno spazio disponibile. Se tutti i bit di β sono 1 (quindi nessun spazio disponibile), si istanzia un nuovo flat, di conseguenza viene esteso β di altre 64 locazioni così come l'hash map e l'array di interi. Viene marcato quindi il primo indice che mi dà 0, dunque i. Successivamente, analizzo il BitSet del nuovo Bloom Filter: per ogni j-esimo bit settato a 1 del Bitset, faccio l'OR di quest'ultimo con l'i-esimo bit dell'intero j dell'ultimo flat istanziato. In questo modo si riesce a memorizzare la corrispondenza dei nuovi elementi del nuovo Bloom Filter nella struttura.
Nella Figura 2.2, viene mostrato il Bloom Filter da inserire con BitSet 10...011. Dall'array β, si ricava $i = 62$ come primo indice disponibile su cui memorizzare. I bit del Bloom Filter settati ad 1, sono nella posizione 0, 62, 63. Semplicemente eseguo l'OR di quest'ultimi con i 62-esimi bit degli interi con indice 0, 62, 63 del flat1. L'operazione di OR non è altro che un mezzo matematico per cambiare lo stato di un bit da 0 ad 1.

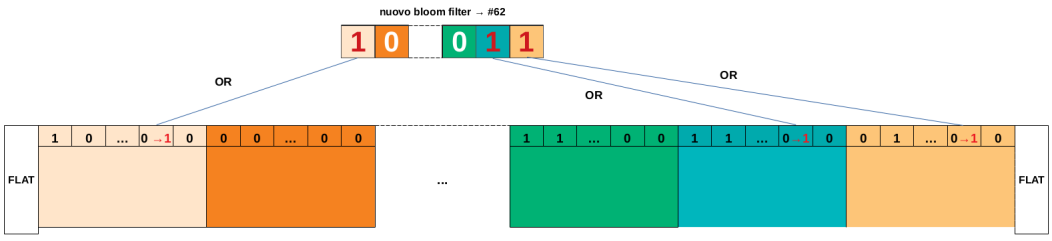

Figura 2.5: Inserimento di un nuovo Bloom Filter in un flat.

2.4.4 Cancellazione

Dato un ID di un Bloom Filter, si vuole che esso venga rimosso dal sistema Flat. L' ID viene passato all'hash map, il quale mi restituisce il corrispondente indice i nella struttura. Banalmente, l'hash map adesso potrà anche eliminare la corrispondenza $< index, ID >$ per questo Bloom Filter. Si scorre l'array β fino alla cella i e scrive 0 anziché 1, per dichiarare che quella postazione adesso è

disponibile. L'operazione di rimozione è speculare a quella di inserimento, ossia, dato il BitSet del Bloom Filter da eliminare, per ogni bit settato ad 1, viene posto 0 il bit corrispondente nella relativa struttura Flat, come nella Figura 2.2. Esiste un flusso alternativo a questa esecuzione: se il Bloom Filter da eliminare fosse il primo di un flat, allora il medesimo non avrebbe più senso di rimanere memorizzato, in tal caso è necessario eliminare l'intero flat, aggiornare l'hash map, e tutte le strutture dati che hanno dipendenze con l'indicizzazione del Bloom Filter in questione. Un esempio tangibile della cancellazione come nella prima casistica lo si trova in Figura 2.6, in cui il Bloom Filter da eliminare ha l'ID = 62. Viene scritto banalmente $\beta[62] = 0$; i bit settati ad 1 del BitSet hanno indice 0, 62, 63, dunque sono posti a 0 i sessantaduesimi bit del flat1 delle celle 0, 62, 63. sono Nella Figura seguente invece, l'ID da cancellare è il 64. E' evidente come la cella riservata al Bloom Filter da cancellare si trovi nel secondo flat. Viene anche qui aggiornato β ed il secondo flat viene eliminato.

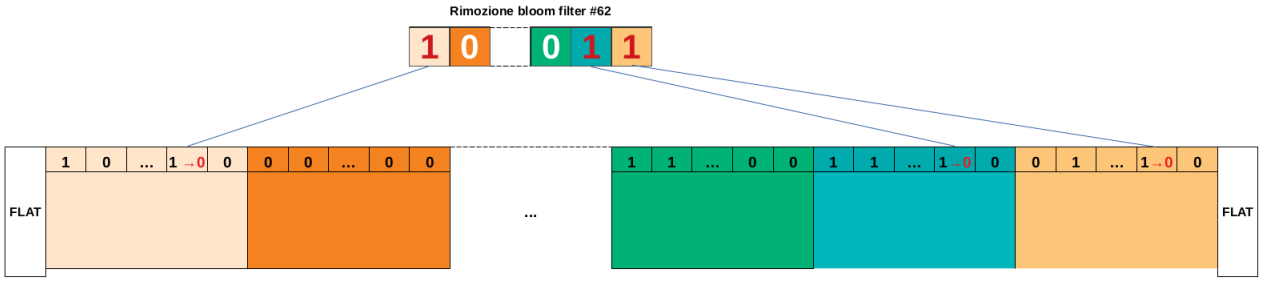

Figura 2.6: Rimozione di un Bloom Filter dalla struttura, Caso 1.

2.4.5 Aggiornamento

Aggiornare un Flat Bloofi è un operazione abbastanza semplice: dato in input l'ID del Bloom Filter da modificare, mi ricavo il corrispondente indice tramite l' hash map, così da capire in che flat è memorizzato. L' algoritmo procede applicando la stessa strategia dell' Insert, aggiornando tramite i bit del nuovo BitSet del Bloom Filter, i valori dei bit nel flat per mezzo di OR bit a bit.

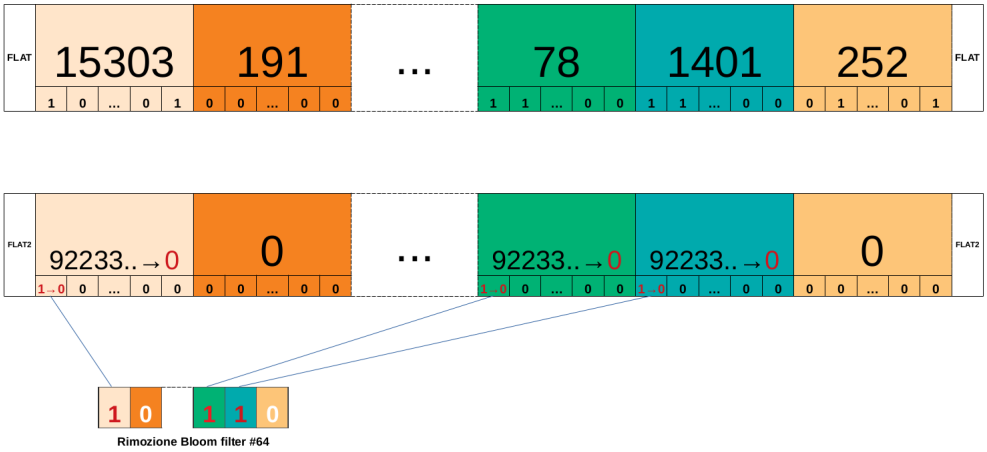

Figura 2.7: Rimozione di un Bloom Filter dalla struttura, Caso 2.

Capitolo 3

Implementazioni di un Flat Bloofi

3.1 Panoramica

In questo capitolo vengono analizzate due implementazioni del Flat Bloofi. La versione in Java, proposta dall'ideatore Daniel Lemire, è stata tratta da [2]. Nel terzo paragrafo viene fornita una nostra implementazione in C, con il fine di studiare possibili margini di miglioramento e/o trarne delle conclusioni interessanti sul loro confronto.

3.2 Implementazione in Java

L' implementazione in Java è composta da varie classi che lavorano in sinergia. Alcune di esse sono frutto del lavoro di D. Lemire, altre invece sono state a loro volta ereditate da implementazioni di strutture già disponibili, come la classe BitSet oppure la classe BloomFilter, scritta da Magnus Skjegstad.
La logica verte principalmente su cinque classi:

- **BloomIndex**: In realtà trattasi di un' interfaccia. BloomIndex viene implementata da tutte le strutture di Bloom Filters multidimensionali, poiché richiede l'override delle operazioni principali, ossia: inserimento, cancellazione, rimozione di un Bloom Filter e ricerca di un elemento.

- **FlatBloomFilterIndex**: E' la classe che ha più interesse, implementa le funzioni dell'interfaccia BloomIndex e rispecchia le qualità di FlatBloofi che abbiamo descritto nei capitoli precedenti.

- **BloomFilter**: Implementa la logica di un classico Bloom Filter, ha natura parametrica, per tal motivo, è possibile inserirci all'interno qualsiasi tipo di elemento, da stringhe, interi, etc...

- **BitSet**: BitSet è un array di tipo Long a 64 bit. E' il cuore di un Bloom Filter, anche se a prima vista non si direbbe: nell'elaborato si è detto che un Bloom Filter è formato da un array di booleani con cui tracciare gli elementi inseriti settando ad 1 le celle indicate, ma il BitSet non contiene questo tipo di array. Questo perché, per usare la classe BitSet, il Bloom Filter dovrà riconoscerlo come se ogni cella di Long fossero 64 bit a cui assegnare valori 0 o 1, quindi il numero di bit disponibili sono $N \times 64$. BitSet inoltre è usata anche in modo diretto da FlatBloomFilterIndex.

- **Hasher**: La classe incaricata della costruzione e dell'esecuzione delle funzioni hash, usate sia da BloomFilter e sia da FlatBloomFilterIndex per far riferimento agli elementi del BloomFilter o/e BloomFilter stessi. Il numero di funzioni hash non è settabile normalmente, ma viene autogenerato in relazione alla percentuale di falsa positività e alla dimensione del Bloom Filter.

  Una visione complessiva delle dipendenze è mostrata nella Figura 3.1.

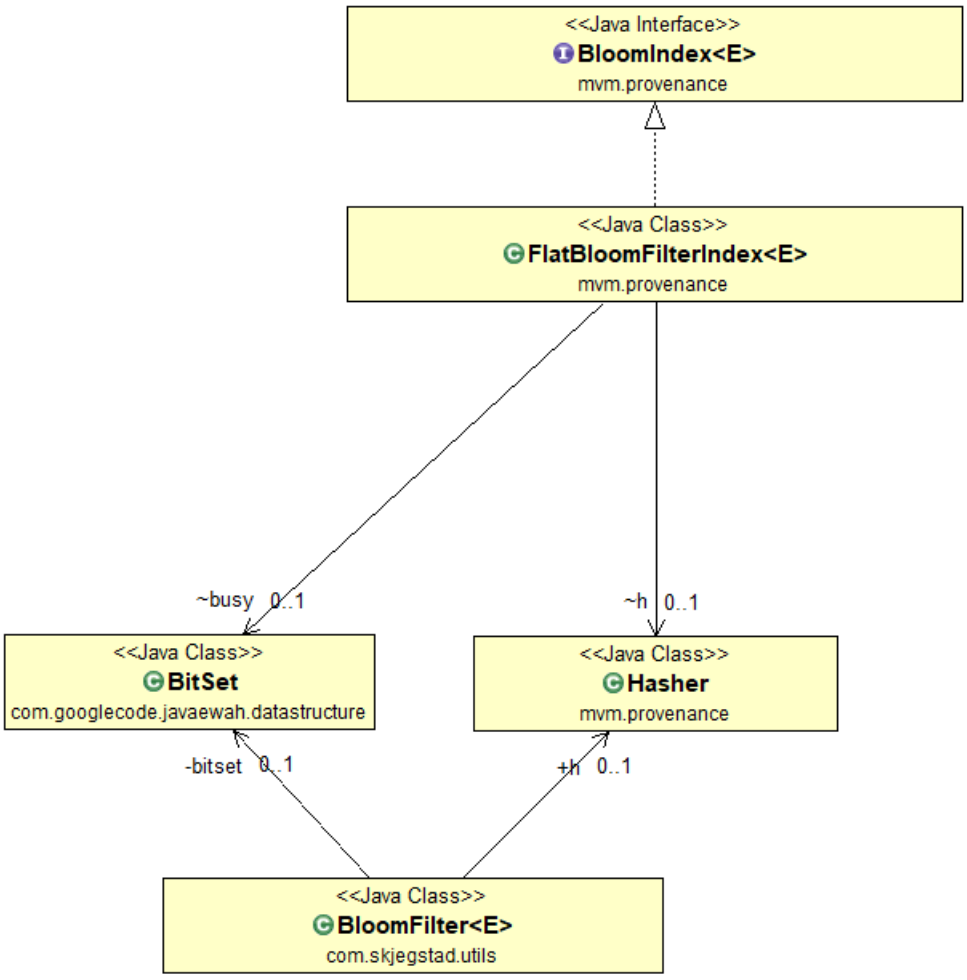

Figura 3.1: Class Diagram FlatBloofi Java.

3.2.1 BloomIndex

L' interfaccia definisce diversi prototipi, non tutti, però, sono usati per il funzionamento di FlatBloofi. Metodi come $getIsRootAllOne$, $getNbChildrenRoot$, $getHeight()$, hanno senso solo nell'implementazione Bloofi. Gli altri metodi sono:

- **DeleteFromIndex**: Incaricato di eliminare un Bloom Filter dalla struttura, dove $< i >$ è l'indice del Bloom Filter a cui riferirsi.
- **GetBloomFilterSize**: Restituisce il numero di Bloom Filters memorizzati all'interno del Flat.
- **insertBloomFilter**: Permette di inserire un Bloom Filter nella struttura.
- **search**: Restituisce un ArrayList che come elementi gli indici dei Bloom Filter che probabilmente contengono l'oggetto dato in input $< o >$.
- **updateIndex**: E' possibile modificare il BitSet di un Bloom Filter, magari modificando degli elementi all'interno, e ricaricalo nel flat tramite questa funzione.

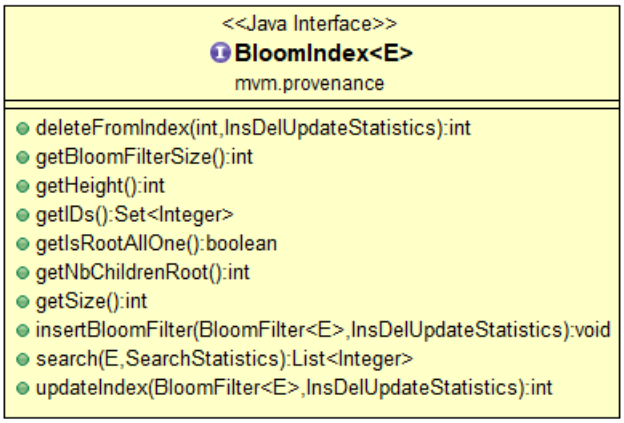

Figura 3.2: Interfaccia BloomIndex.

3.2.2 BloomFilter

Vengono mostrati solo i parametri ed i metodi importanti nel contesto Flat.

- **bitSet**: Il BitSet del Bloom Filter.
- **bitSetSize**: la taglia del Bloom Filter.
- **bitPerElement**: il numero di bit riservati per ciascun elemento.
- **h**: rappresenta l'hasher usato.
- **k**: il numero di funzioni hash.

Metodi

Notasi che metodi come *computeHammingDistance*, *computeCosineDistance*, *computeJaccardDistance*, vengono trattati solamente nel Bloofi per posizionare vicini man mano Bloom Filters che si differenziano di meno, al fine di incrementare l'efficienza nell'operazione di ricerca.

- **void add(E)**:
  Aggiunge un elemento di tipo E al Bloom Filter.

- **void clear()**:
  Setta tutti i del BitSet a 0.

- **boolean contains(E)**:
  Restituisce $True$ se l'elemento E è, con una certa probabilità, memorizzato nel Bloom Filter.

- **boolean isFull()**:
  Restituisce $True$ se il Bloom Filter è pieno, ossia tutti i bit suo BitSet sono a 1.

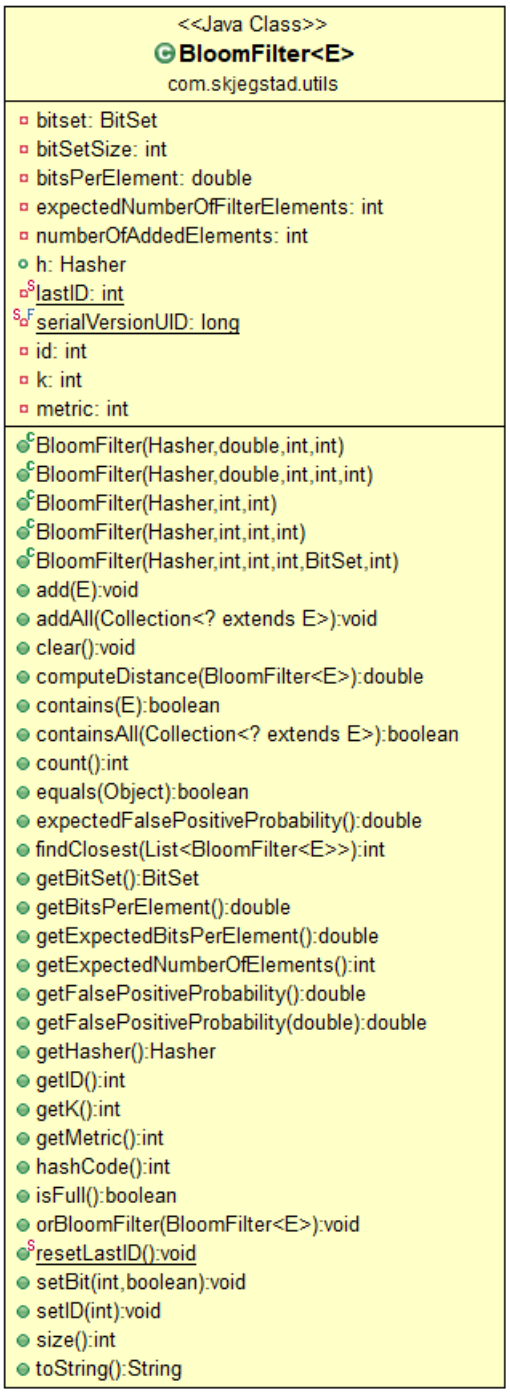

Figura 3.3: La classe Bloom Filter.

3.2.3 BitSet

Parametri

- **data**: l'array di interi long a 64 bit.

Metodi

- **int cardinality()**: Fornisce il numero di bit a 1 ricavato da ogni Long.
- **int nextUnsetBit(int)**: Restituisce l'indice del primo bit a 0, controllando a partire dalla posizione indicata dal parametro dato in input. Dà -1 se non ci sono bit a 0.
- **int clear()**: Resetta tutti i bit a 0 (Svuota il BitSet).

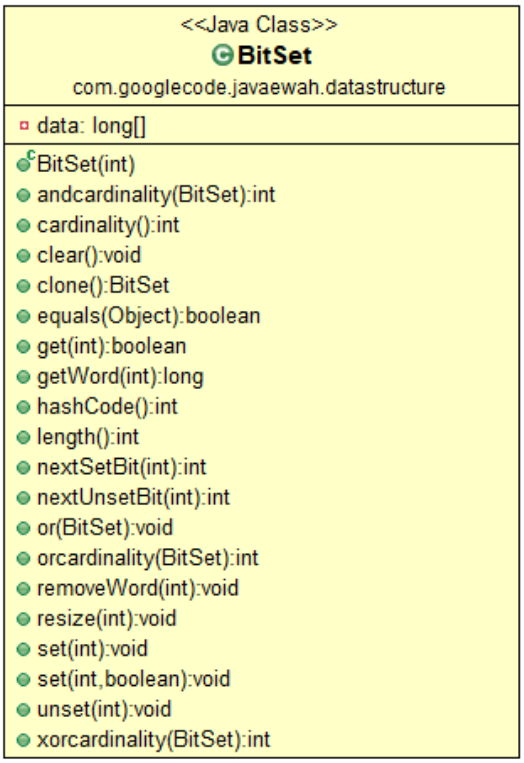

Figura 3.4: La classe BitSet.

3.2.4 Hasher

Parametri

- **maxval**: la massima taglia che può raggiungere il BitSet da computare.
- **randomkeys**: Un array di interi che predispone valori casuali per generare un hash, si ha un seme per ogni funzione hash istanziata.
- **r**: Generatore di numeri casuali.

Metodi

- **int hash(Object, int)**: Computa il valore hash dell'elemento usando il numero di chiave specificata dal secondo parametro
- **void setNumberOfRandomKeys(int)**: Setta il numero di funzioni hash da utilizzare.
- **void setMaxValue(int)**: Aggiorna la dimensione del BitSet con il nuovo valore.

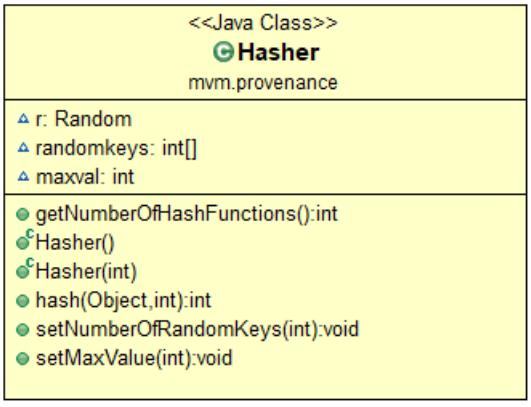

Figura 3.5: La classe Hasher.

3.2.5 FlatBloomFilterIndex

Dulcis in fundo la classe Java che rappresenta la struttura Flat. Si fa presenta che quest'ultima implementa i metodi dell'interfaccia $BloomIndex$.
Parametri

- **fromindextoId**: Un array di interi che contiene tutti gli ID dei Bloom Filter indicizzati.
- **idMap**: Mantiene il mapping tra un ID di un Bloom Filter ed il suo corrispettivo indice nella strutttura.
- **buffer**: E' un ArrayList di array di long, rappresenta l'insieme di tutti i Flat che sono presenti nell' apparato.
- **busy**: Un BitSet che traccia le posizioni occupate (con 1) e quelle libere(con 1) dai Bloom Filter nel Flat Bloofi.

InsertBloomFilter(BloomFilter)
La funzione inizia controllando la prima posizione libera nella struttura in cui inserire il nuovo Bloom Filter. Questo è possibile grazie alla procedura $nextUnsetBit$, metodo di *busy*. Se restituisce -1, quindi non ha trovato posizioni libere, allora viene effettuata una $resize$ di *busy*, aggiungendo altre 64 postazioni e aggiungendo un nuovo flat. Deleghiamo quest'ultima operazione alla funzione *add* di $buffer$ che aggiungerà un nuovo array di long. Quando $nexUNsetBit$ non restituisce -1, allora possiamo inserire il nuovo Bloom Filter nella posizione indicata, eseguendo $setBloomAt$. L' inserimento si conclude aggiornando $idMap$ con il nuovo indice e settando ad 1 il *bit* di busy che prima era settato a 0.

search(E)
Viene preimpostato un ArrayList *answer* che avrà il compito di mantenere traccia degli ID dei Bloom Filter che conterranno probabilmente E. Dopodiché viene fatta scorrere $buffer$, quindi, per ogni flat inserito:
Per ciascuna funzione chiave hash, calcola il valore hash dell'elemento e, con l'hash che si ottiene, fai riferimento alla cella hash-esima dell'i-esimo flat. Faccio l'AND dei valori ottenuti. Finché tutti i bit dei valori finali, risultati di ogni flat, non sono tutti a 0: prendi gli indici dei bit che sono a 1, i quali corrisponderanno agli indici dei Bloom Filter che contengono probabilmente l'elemento. Vengono salvati questi ID in *answer* e restituiti come risposta.

updateIndex(BloomFilter)
Chiamo semplicemente la funzione *setBloomAt* dando in input il BitSet aggiornato e l'indice del Bloom Filter da modificare ricavato da *idMap*.

deleteFromIndex(int id)
Dato in input l'ID del Bloom Filter, si ricava tramite *idMap* l'indice del Bloom Filter da rimuovere. Si setta a 0 il bit di *busy* con l'indice appena trovato. Si diramano due flussi di esecuzione. Se il Bloom Filter da eliminare è il primo di un Flat, allora rimuovo anche il flat corrispondente usando il metodo *remove* di buffer. Altrimenti elimino solo il Bloom Filter 'pulendo' il relativo Flat con la funzione "clearBloomAt".

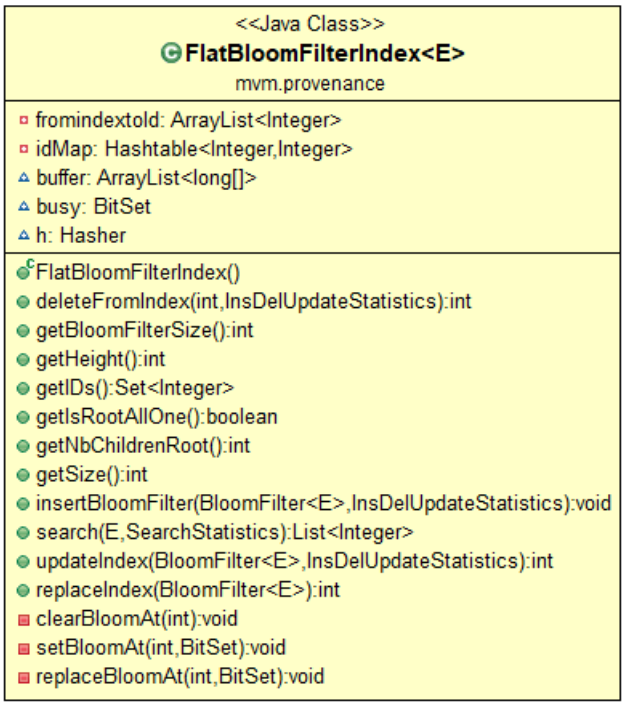

Figura 3.6: La classe FlatBloomFilterIndex.

3.3 Implementazione in C

3.3.1 Da Java in C

Punto nevralgico dell'implementazione in C di questo lavoro di tesi è stato sicuramente il controllo dei puntatori a tutte le strutture di supporto al Flat Bloofi, badare alle allocazioni di memoria ed anche al passaggio dei riferimenti. Riportare in C un' idea concepita ad oggetti (come da buona prassi Java) significa dover ristrutturare la logica delle chiamate a funzioni, le quali sono scisse dall'oggetto non essendoci più metodi di classe. E' stato necessario implementare, oltre alla logica principale, anche delle strutture di sostegno, come HashTable, liste, Hasher etc.. le quali magari sono di facile accesso in Java. Si presenta il lavoro svolto nei successivi sotto paragrafi.

3.3.2 Uno sguardo alle strutture di supporto

```
struct flatbloofi{

    list fromindextoId;
    struct DataItem* idMap[SIZE];
    list buffer;
    bitset_t* busy;

    struct hasher* h;

};
```

Figura 3.7: La struttura FlatBloofi in C.

flatbloofi

La struttura rievoca quelle che erano le entità già presenti nella struttura in Java. Abbiamo sta volta $fromindexToId$ e $buffer$ che sono implementate con una comune lista scritta in C. Per una lista abbiamo fatto uso delle operazioni indicate dalla figura 3.1. Da notare come la soluzione "a lista" fornisce una soluzione generica per concatenare oggetti uno in fila all'altro senza preoccuparsi del tipo, è stato fatto dunque un lavoro di casting laddove era necessario.
Con *DataItem* abbiamo fatto riferimento ad una struttura HashTable con il

quale si è correlato, anche in questo caso, l'ID del Bloom Filter all'indice della posizione nel Flat Bloofi (vedi figura 3.9). L' Hasher invece, è stato curato dal collega F. Priore che ha integrato una funzione per il calcolo di valori hash su una stringa di input.

```
list newList();
list insertElement(list l,void *x);
boolean isFullList(list l);
boolean isEmptyList(list l);
int getSize(list l);
void* getElement(list l,int index);
int compareElement(void *a,void *b);
list addAll(list l,list new);
list insertElementByIndex(list l,int index,void *newChild);
list deleteElement(list l,void *x);
list deleteElementByIndex(list l,int index);
int indexOfElement(list l,void *bfn);
list removeLastHalf(list l,int from);
```

Figura 3.8: Operazioni su una lista.

```
struct DataItem {
   int data;
   int key;
};
```

Figura 3.9: Composizione HashTable.

BitSet

Passiamo all'analisi della struttura *BitSet*. L'implementazione di esso nasce da una rivisitazione di una versione già esistente [4]. il BitSet è costituito questa volta con un puntatore al tipo $uint64t$ per rappresentare l'array di dati, sostituto del Long in Java. Abbiamo inoltre riprogettato diverse funzioni dei BitSet tra cui $bitsetResize$, $bitsetUnset$, $getLength$, e $nextUnsetBit$ creata invece ex novo. La funzione ritorna l'indice del primo bit a 0 che incontra scorrendo il BitSet dato in input fig.3.11.

```
struct bitset_s
{
   uint64_t * restrict array;
   size_t arraysize;
   size_t capacity;
};

typedef struct bitset_s bitset_t;
```

Figura 3.10: la struttura di BitSet.

```
int nextUnSetBit(const bitset_t *bitset, int i) {

    int x = i >> 6;

    if (x >= bitset->arraysize) {
      return -1;
    }


    uint64_t w = ~ bitset->array[x];
    w >>= (i & 63);
    if (w != 0) {
        int mh = i += __builtin_ctzll(w);
        return mh;
    }

    ++x;

    for(; x < bitset->arraysize; ++x) {
      w = bitset->array[x];
      if (w != ~0) {
          return i = x * 64 + __builtin_ctzll(~w);
      }
    }


    return -1;
}
```

Figura 3.11: La funzone nextUnsetBit.

Bloom Filter

Come ultima struttura ausiliaria, abbiamo la struttura Bloom, il quale rappresenta il nostro Bloom Filter. Abbiamo usato anche qui un'implementazione già esistente [11], anche se non forniva tutti gli strumenti necessari per i nostri scopi. Si è stilata un' unica struttura "bloom", la quale è adatta sia per bloofi che per flat bloofi; viene fornita da bloom.h.

```
struct bloom
{
    int bf;
  int entries;
  double error;
  int bits;
  int bytes;
  int hashes;
  int id;
  double bpe;
  bitset_t *b;
  int ready;
  int numberOffElement;
  int metric;
  struct hasher*h;
};
```

Figura 3.12: bloom.h.

3.3.3 Implementazione delle operazioni fondamentali

Nella figura 3.13, si presenta l'implementazione in C dell'inserimento di un Bloom Filter. La logica richiama l'algoritmo già descritto nei paragrafi precedenti. Si presti attenzione ad una sottoprocedura fondamentale, *setBloomAt*. Quest'ultima è incaricata dell'effettivo inserimento nel flat e viene usata in modo analogo anche dall' operazione di aggiornamento. In figura 3.14, si viene mostrato come, all'interno del for, il BitSet del Bloom Filter viene memorizzato nelle destinate posizioni nel Flat a cui è stato assegnato. Questo è possibile grazie all'ausilio di una maschera, composta da un intero long con valore 1000..0 di i zeri (dove i è la posizione futura del bloom flter) che, applicando un OR con il bit associato al flat, simula l'aggiornamento del bit da 0 a 1.

```
void insertBloomFilter(struct flatbloofi *bl,struct bloom *b){


    if(bl->h != NULL){
        if(b->h != NULL);
            printf("Stai usando più di un hasher\n");
    }
    else bl->h = b->h;

    int i = nextUnSetBit(bl->busy, 0);

    if (i < 0) {
        i = bitset_size_in_bits(bl->busy);
        bitset_resize2(bl->busy, i+64);
        int capienza = get_length(b->b);

        uint64_t *arraydilong = calloc(capienza, sizeof(uint64_t));

        insertElementDim(bl->buffer, arraydilong, capienza);
    }


    int idBloomFilterNew  =  b->id;

    if(i < getSize(bl->fromindextoId)){
        insertElementByIndex(bl->fromindextoId, i, idBloomFilterNew);
    }
    else{
        insertElement(bl->fromindextoId, idBloomFilterNew);
    }


    setBloomAt(bl, i, b->b);
    insert(bl->idMap, idBloomFilterNew, i);
    bitset_set(bl->busy, i);


}
```

Figura 3.13: Operazione di inserimento di un Bloom Filter in C.

```
void setBloomAt(struct flatbloofi*bl, int i, bitset_t *bitset) {
    ......................
    uint64_t* mybuffer;
    ......................
   uint64_t mask = (1l << i);
    for (int k = nextSetBit2(bitset, 0); k >= 0; k = nextSetBit2(bitset, k+1)) {
        mybuffer[k] |= mask;
    }

    ......................
}
```

Figura 3.14: Funzione setBloomAt.

Ricerca

Viene riportato anche un estratto dell'operazione di ricerca (fig. 3.15). w è il valore finale W_f, risultato degli AND tutti degli interi selezionati dalla funzione hash. Per simulare questa operazione, si assegna a w il valore 1111...11 a 64 cifre(nel codice viene fatto l'opposto del valore 0), e viene fatto l'AND concatenato con tutti i valori W_i indicati. Poiché Wf è composto solo da 1, allora l'effetto di AND su di esso è nulla nella prima iterazione. Per farla breve, la maschera è usata solo come struttura di supporto iniziale per poter effettuare più AND durante le varie iterazioni. Se W_f, dopo questo ciclo, risulta diverso da 0, allora è probabile che *search* abbia trovato Bloom Filters. Nel while in Figura 3.15, il valore di W_f viene ancora una volta ciclato e, tramite l'operazione w AND -w (trattasi di un AND bit a bit), in t viene salvato un valore la cui rappresentazione binaria è intuitivamente un 1 in una posizione i seguita da degli zeri: l'i-esimo bit non è altro che il primo bit di W_f più a destra settato a 1. Se volessimo dare una descrizione più ad alto livello, potremmo dire che stiamo scorrendo per tutti i bit a 1 di W_f. Questo succede perché, alla fine di ogni ciclo, viene effettuata un' operazione di XOR tra il W_f corrente e t che azzererà il bit a 1 più a destra, per poter lasciare all'iterazione successiva la possibilità di ripete la stessa l'operazione sul bit $i + 1$ (dato che sarà il primo bit a 1 pù a destra). Ma come viene calcolato l'ID dei Bloom Filters che rispondono alla query? Ogni volta questi vengono salvati nella variabile *indexbloom* che sarà inserita poi in una lista *answer* con un valore sempre diverso. L'ID in *indexbloom* è pescato dall'indice ottenuto da builtinPopcountll(t-1), il quale mi restituisce la posizione dell'1 in t decrementata di 1. Alla fine, il numero ottenuto viene spiazzato di 64 per il numero del flat corrente.

```
int* search(struct flatbloofi*bl, const void*o){

   ....................................

    for (int i = 0; i < getSize(bl->buffer); ++i) {

        uint64_t w = -0l;

        for(int l = 0; l <  bl->h->numdichiavi; ++l){
            int hashvalue = (int) hash( bl->h, o, l);


            struct nodeList_withSize *nodoDiLong= getNodeElementDim(bl->buffer, i );

            w &=  (uint64_t) nodoDiLong->array[hashvalue];

        }

        int counter = 0;
        while (w != 0) {
            uint64_t t = w & -w;
            int indexbloom = getElement(bl->fromindextoId, i * 64 +  __builtin_popcountll(t-1));
            ..............
            w ^= t;
```

Figura 3.15: Operazione di ricerca in C.

Rimozione

Dato in input l'ID, Dopo aver ricavato il corrispondente indice del Bloom Filter di interesse tramite IdMap, viene chiamata la funzione *bitsetUnset* per settare a 0 la cella in *Busy* che è stata indicata. Tramite *getWord*(i/64), viene dedotto se il Bloom è il primo di un flat, se è così, viene cancellato tutto l'array di interi aggiornando tutte le strutture dati che hanno un riferimento con quest'ultimo. Altrimenti viene eseguita la funzione *clearBloomAt* che azzererà i bit a 1 del flat che fanno riferimento al BitSet del BloomFilter da cancellare.

Capitolo 4

Testing

4.1 Introduzione

il Capitolo 4 è dedicato interamente al testing delle strutture dati nelle varie implementazioni. Viene messo alla prova il nostro elaborato tramite casi di test per mostrare che rispetti quelli che sono i requisiti funzionali delle 4 operazioni, questo avviene nel Test di correttezza. Nel paragrafo successivo, viene fatta invece un'analisi della memoria durante l'esecuzione del nostro elaborato in C e paragonata con i risultati sperimentali in Java ottenuti con le medesime condizioni di esecuzione. A tal proposito, si fa presente che tutti i casi di test sono stati eseguiti sulla stessa macchina (CPU i7-8550U a 1.8GHz, 8GB di RAM) e, per la implementazione in Java, è stata scelta la jdk v.1.8. Il test di confronto C e Java, ha una prosecuzione sui tempi di esecuzione, in funzione di un numero di Bloom Filters in crescita. Ultimo aspetto testato è stato il confronto tra le nostre implementazioni, ossia Bloofi e Flat Bloofi in C. Infine, nel paragrafo finale, viene fatta una panoramica degli esiti dei test e cenni a possibili miglioramenti futuri.

4.2 Test di correttezza

4.2.1 Preparazione casi di test

- **Inizializzazione, inserimento e ricerca**
  Testare la correttezza delle 3 operazioni diviene un compito arduo se testate separatemente. Si pensi ad un' inizializzazione. Senza effettuare un inserimento e una ricerca, è difficile garantirne la robustezza senza poter notare riscontri su altre operazioni. Discorso analogo per l'inserimento se

non viene eseguita la ricerca di un elemento sul Bloom Filter appena memorizzato. Per questi motivi, abbiamo deciso di creare un caso di test che possa verificare tutte e 3 le operazioni in una sola esecuzione: creiamo un numero x di un Bloom Filters, che vengono aggiunti in una lista e carichiamo ciascun Bloom Filter con m elementi. Poi, facciamo scorrere la lista di Bloom e man mano vengono inseriti nel Flat Bloofi con $insertBloomFilter$. dopodiché viene fatta una ricerca di un elemento casuale nella struttura. L' algoritmo viene ripetuto su valori diversi e, se in tutti i casi il test ha esito positivo, allora abbiamo una buona probabilità che tutte e 3 le operazioni funzionino a dovere.

- **Aggiornamento**
  L'update affronta le stesse perplessità: anche qui, creiamo una lista di Bloom Filters carichi di elementi, aggiorniamo il BitSet di uno di essi, magari aggiungendo un nuovo elemento, ed effettuiamo la ricerca su quest'ultimo per verificarne il successo dell' aggiornamento.

- **Rimozione**
  Viene caricato un Flat Bloofi con diversi Bloom, in uno di questi, aggiungiamo una parola s che non è contenuta in nessun altro Bloom Filter. Viene eseguita $deleteBloomFromIndex$ su quest'ultimo e si verifica, con una search, la presenza della parola s. Se s non viene trovata, allora siamo sicuri della correttezza dell'operazione di cancellazione.

4.2.2 Risultati

Inizializzazione, inserimento e ricerca sono verificate come in Figura 4.1. In questo test, si può notare l'inserimento di 4 Bloom Filters in $bfList$ e l'inserimento di parole al loro interno. Si è fatto in modo che parole di uno stesso Bloom Filters siano facilmente riconoscibili; ad esempio nel primo Bloom filter, sono contenute parole del tipo "gatto0, gatto1, gatto2, ..., cane0, cane2,.., cane500". Successivamente si applica $insertBloomFilter$ aggiungendo i Bloom dalla lista $BfList$ e poi cercata la parola "forchetta177". L'esito è risultato positivo, infatti la parola è contenuta nel Bloom filter con ID = 3, come si vede un Figura. Anche in altri casi vi è stato successo. Un altro test che abbiamo deciso di preparare è il verificare che copie di elementi uguali, salvati in Bloom Filters diversi, possano essere intercettati dalla search. Detto in altri termini, se ho copie di elementi sparsi in vari Bloom, si vogliono gli ID di tutti i Bloom filter che contengono le copie. Anche questo test ha avuto esito positivo, come si può vedere in Figura

4.2. Infatti, la parola "forchetta177" questa volta è stata trovata sia nel Bloom Filter con ID = 2 e sia in quello con ID =3.

```
bfList=insertElement(bfList,bf); //Inserimento del bf nella lista
bfList=insertElement(bfList,bf2); //Inserimento del bf nella lista
bfList=insertElement(bfList,bf3); //Inserimento del bf nella lista
bfList=insertElement(bfList,bf4); //Inserimento del bf nella lista


 for(int i = 0; i <500; i++) {

    bloom_add(bf,concatena2("gatto", i));
    bloom_add(bf,concatena2("cane", i));

    bloom_add(bf2, concatena2("giacomo", i));
    bloom_add(bf2,concatena2("aldo", i));
    bloom_add(bf2,concatena2("giovanni", i));

    bloom_add(bf3,concatena2("pentola", i));
    bloom_add(bf3,concatena2("forchetta", i));

    bloom_add(bf4,concatena2("pc", i));
 }


struct bloom* current = NULL;

for(int i = 0; i < bfList->size; i++){
    current = getElement(bfList,i);
    insertBloomFilter(bl,current);
 }

search(bl, "forchetta177");
```

```
Console
<terminated> FlatBloofi [C/C++ Application] /home/fernet/workspaceC/FlatBloofi/Debug/FlatBlo
******* FLAT BLOOFI  C *******
la parola si trova nel bloom filter con id: 3
```

Figura 4.1: Test inzializzazione, inserimento e ricerca

Si precisa che il test ha successo anche su Bloom filter di taglie diverse. Passiamo all'analisi del test dell'Update. In questo caso, abbiamo aggiornato il BitSet del Bloom filter con ID = 1, aggiungengo la nuova parola "lupo". La ricerca è stata efficace dando anche qui una risposta positiva.

```
bfList=insertElement(bfList,bf); //Inserimento del bf nella lista
bfList=insertElement(bfList,bf2); //Inserimento del bf nella lista
bfList=insertElement(bfList,bf3); //Inserimento del bf nella lista
bfList=insertElement(bfList,bf4); //Inserimento del bf nella lista


 for(int i = 0; i <500; i++) {

    bloom_add(bf,concatena2("gatto", i));
    bloom_add(bf,concatena2("cane", i));

    bloom_add(bf2, concatena2("forchetta", i));
    bloom_add(bf2,concatena2("aldo", i));
    bloom_add(bf2,concatena2("giovanni", i));

    bloom_add(bf3,concatena2("pentola", i));
    bloom_add(bf3,concatena2("forchetta", i));

    bloom_add(bf4,concatena2("pc", i));
 }


struct bloom* current = NULL;

for(int i = 0; i < bfList->size; i++){
    current = getElement(bfList,i);
    insertBloomFilter(bl,current);
 }

search(bl, "forchetta177");
```

Console

```
<terminated> FlatBloofi [C/C++ Application] /home/fernet/workspaceC/FlatBloofi/Debug/FlatB
******* FLAT BLOOFI  C *******
la parola si trova nel bloom filter con id: 2
la parola si trova nel bloom filter con id: 3
```

Figura 4.2: Ricerca su elemento contenuto in più Bloom Filters.

```
        for(int i = 0; i <500; i++) {

           bloom_add(bf,concatena2("gatto", i));
           bloom_add(bf,concatena2("cane", i));

           bloom_add(bf2, concatena2("forchetta", i));
           bloom_add(bf2,concatena2("aldo", i));
           bloom_add(bf2,concatena2("giovanni", i));

           bloom_add(bf3,concatena2("pentola", i));
           bloom_add(bf3,concatena2("forchetta", i));

           bloom_add(bf4,concatena2("pc", i));
        }


        struct bloom* current = NULL;

        for(int i = 0; i < bfList->size; i++){
            current = getElement(bfList,i);
            insertBloomFilter(bl,current);
         }


       current=getElement(bfList,0); // indice 0 ma ID = 1
       bloom_add(current, "lupo");  // aggiungo 'lupo' al primo bloom filter
       updateIndex(bl,current);

       search(bl, "lupo");
```

Console

```
<terminated> FlatBloofi [C/C++ Application] /home/fernet/workspaceC/FlatBloofi/Debug/FlatBloofi (15
******* FLAT BLOOFI  C *******
la parola si trova nel bloom filter con id: 1
```

Figura 4.3: Test Update su Bloom filter con ID =1.

Per la remove, abbiamo settato il Flat Bloofi similimente a come si è fatto in precedenza. Viene eliminato tramite *deleteBloomFilter* il Bloom Filter con ID = 3. La parola "forchetta177" che apparteneva al Bloom Filter 3, non viene rilevata dalla search, questo è segno che la rimozione è avvenuta con successo.

Figura 4.4: Rimozione di un Bloom Filter.

L'indagine sulla correttezza, si è estesa sull'invocazione di più remove su molteplici Bloom Filters. Abbiamo notato che il programma si arresta e dà in errore se si vuole cancellare un Bloom Filters che è indicizzato da solo in un Flat. Quindi: inserendo il 65esimo Bloom Filters, verrà a crearsi un nuovo flat. Se si chiama *deleteBloomFilter* su input 65, il programma dovrebbe eliminare l'intero flat poiché la sua intera memorizzazione non avrebbe più senso perché occuperebbe spazio inutile, cosa che invece non accade e il test fallisce, quindi il problema si

manifesta nella rimozione di ID congruo a 64.
Durante il debugging, abbiamo analizzato la stessa casistica anche nel programma in Java, ai fini di carpire il problema seguendo da lì il flusso logico delle chiamate. Sorprendentemente l'anomalia si è manifestata anche nel Flat Bloofi in Java se si pongono le stesse condizioni: viene lanciata un' eccezione quando bisogna eliminare il parametro k dall' array di ID $fromIndexToId$. Per tal motivo, si è deciso di accantonare i test di confronto successivi per la remove.

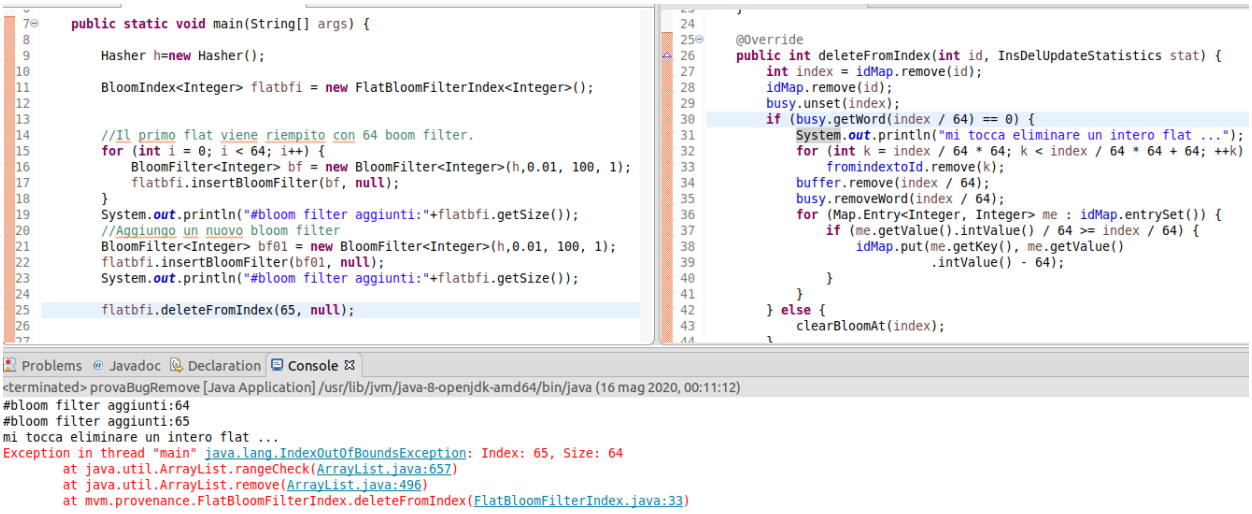

Figura 4.5: Anomalia dell'operazione di rimozione.

4.3 Memoria

Lo spreco di memoria può essere determinante nel prediligere una implementazione rispetto ad un' altra. Se in Java la gestione della memoria viene delegata al Garbage Collector, in C tali accortezze devono essere esplicitate all'interno del codice, allocando, ridimensionando e liberando spazio, aspetti trasversali che abbiamo curato per questo progetto di tesi. Usare minuziosamente funzioni come malloc() o free() può portare ad un rendimento tangibile, specialmente dal punto di vista computazionale, non dovendo far uso di un Garbage Collector come invece succede in Java. L'analisi del consumo di Heap in C è stata eseguita con lo strumento "Massif", un tool integrato nell'ambiente Eclipse. La misurazione tiene traccia dell'esecuzione delle 4 operazioni(5 se si considera l'inizializzazione) su 1000 Bloom Filters da 500 elementi ciascuno. Il risultato è riportato in Figura 4.9. Il consumo di Heap in C ha un picco di circa 25 Mb. Sull'asse y viene mostrata il numero di Kb della memoria utilizzata mentre sulle ascisse il numero di istruzioni eseguite nell'ordine del milione.

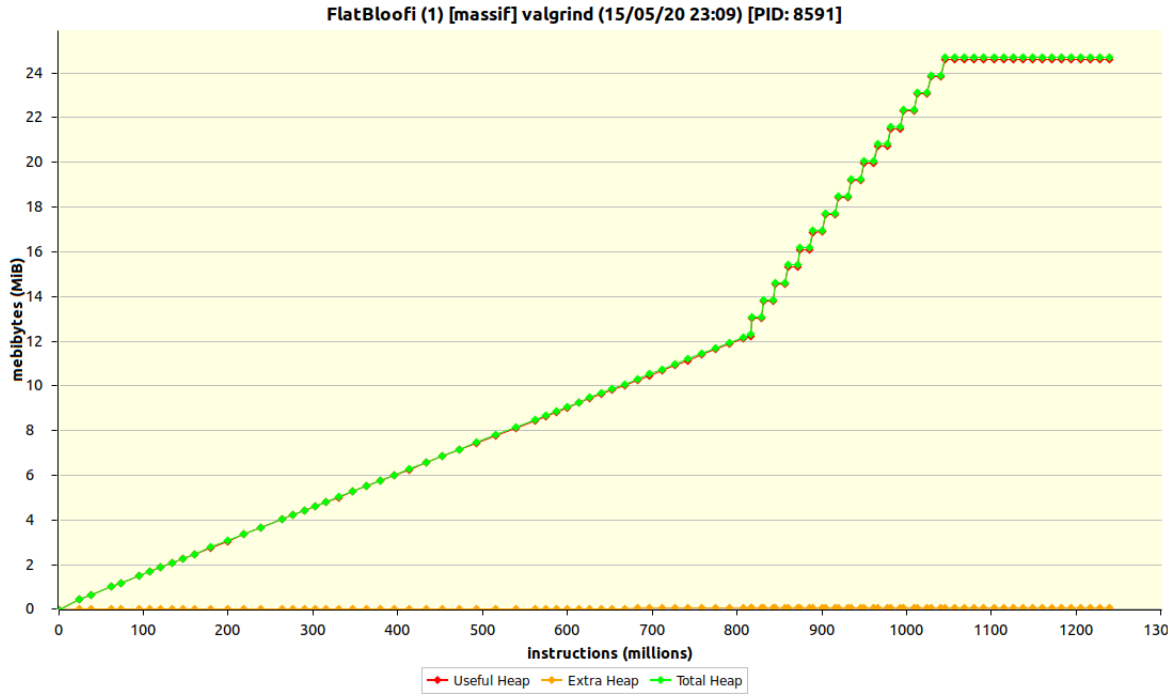

Figura 4.6: Test memoria in C

La misurazione in Java è stata effettutata con il tool "Visual VM", monitorando l'esecuzione del programma in Java predisposto con gli stessi parametri che abbiamo usato in C. Ne è risultato che, il consumo di Heap ha un picco di 38Mb, superiore rispetto ai 25Mb del programma in C. Nella Figura 4.7, l'area delimita l'Heap utilizzata dal processo, in arancio la taglia massima di Heap messa a dispozione.

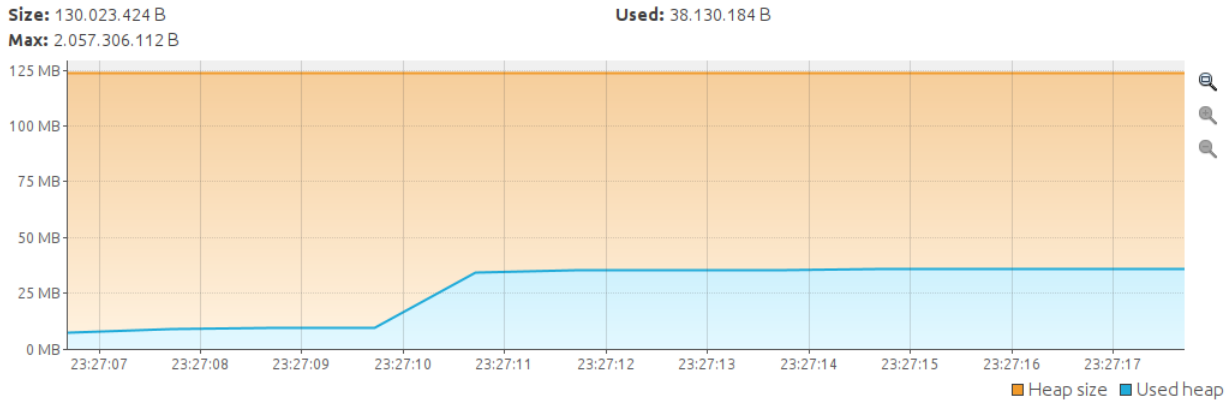

Figura 4.7: Test memoria in Java

4.4 Tempi di esecuzione

4.4.1 Confronto Java e C

Riportiamo in seguito in risultati dei test sul tempo di esecuzione delle operazioni di inserimento, aggiornamento e ricerca per le due operazioni eseguite su parametri di diverso ordine. Per ogni casistica, il test verrà ripetuto per 10 tentativi. Verrà stilato un tempo medio per ogni operazione, i quali sono abbastanza attendibili. Si consideri che, la misurazione del reale tempo effettivo è solo teorizzabile, poiché l'ambiente di esecuzione può essere turbato da fattori esterni come disturbi al livello hardware, oppure la presenza di altri processi nel Sistema Operativo. Tutto sommato, i risultati possono considerarsi attendibili (i valori sono espressi in millisecondi).

- **Implementazione:** Java
- **Numero Bloom Filters:** 1
- **Numero elementi per Bloom Filters:** 5

Numero prova	Inserimento	Aggiornamento	Ricerca
1	0.421	0.036	0.014
2	0.416	0.036	0.014
3	0.416	0.035	0.014
4	0.430	0.036	0.015
5	0.430	0.036	0.015
6	0.699	0.061	0.079
7	0.417	0.036	0.014
8	0.419	0.066	0.054
9	0.457	0.036	0.015
10	0.512	0.036	0.015

Tabella 4.1: Test Java su 1 Bloom Filters di 5 elementi

Inserimento: 0,461 ms
Aggiornamento: 0,041 ms
Ricerca: 0,025 ms

- **Implementazione:** C
- **Numero Bloom Filters:** 1
- **Numero elementi per Bloom Filters:** 5

Numero prova	Inserimento	Aggiornamento	Ricerca
1	0.167	0.009	0.009
2	0.100	0.006	0.003
3	0.164	0.008	0.007
4	0.104	0.007	0.004
5	0.125	0.008	0.005
6	0.107	0.007	0.004
7	0.110	0.007	0.003
8	0.125	0.009	0.006
9	0.093	0.007	0.004
10	0.102	0.007	0.004

Tabella 4.2: Test C su 1 Bloom Filters di 5 elementi

Inserimento: 0,120 ms
Aggiornamento: 0,008 ms
Ricerca: 0,005 ms

Le medie ottenute risaltano un leggero vantaggio, quasi impercettibile, dell' implementazione in C. I valori sono ancora bassi per trarre delle conclusioni.

- **Implementazione:** Java
- **Numero Bloom Filters:** 100
- **Numero elementi per Bloom Filters:** 50

Numero prova	Inserimento	Aggiornamento	Ricerca
1	6.635	0.910	1.944
2	3.971	0.919	1.992
3	4.752	0.717	1.399
4	3.385	0.653	1.247
5	4.442	0.679	1.331
6	3.689	0.655	1.351
7	3.931	1.210	1.804
8	5.433	1.458	2.461
9	3.596	0.658	1.271
10	3.440	0.670	1.165

Tabella 4.3: Test Java su 100 Bloom Filters di 50 elementi.

Inserimento: 4,327 ms
Aggiornamento: 0,853 ms
Ricerca: 1,597 ms

- **Implementazione:** C
- **Numero Bloom Filters:** 100
- **Numero elementi per Bloom Filters:** 50

Numero prova	Inserimento	Aggiornamento	Ricerca
1	4.043	1.062	0.207
2	3.145	2.205	0.294
3	2.906	0.792	0.187
4	3.160	0.944	0.209
5	3.223	0.832	0.214
6	3.207	0.867	0.215
7	2.599	0.686	0.165
8	3.197	0.832	0.208
9	3.280	0.979	0.226
10	3.440	0.670	1.165

Tabella 4.4: Test C su 100 Bloom Filters di 50 elementi.

Inserimento: 3,220 ms
Aggiornamento: 0,986 ms
Ricerca: 0,309 ms

Con 100 Bloom Filters, il divario tra i tempi di inserimento e aggiornamento in Java ed in C è ancora minimo. Per quanto riguarda l'operazione di ricerca, C sembra accumulare un vantaggio importante.

- **Implementazione:** Java
- **Numero Bloom Filters:** 1000
- **Numero elementi per Bloom Filters:** 500

Numero prova	Inserimento	Aggiornamento	Ricerca
1	59.747	14.745	25.396
2	42.324	15.803	25.383
3	49.196	16.178	36.776
4	40.489	13.571	26.925
5	40.450	13.239	27.900
6	43.788	14.205	27.022
7	42.295	14.007	26.044
8	46.588	16.006	25.274
9	44.015	15.949	25.120
10	42.562	13.643	27.366

Tabella 4.5: Test Java su 1000 Bloom Filters di 500 elementi.

Inserimento: 45,145 ms
Aggiornamento: 14,734 ms
Ricerca: 27,320 ms

- **Implementazione:** C
- **Numero Bloom Filters:** 1000
- **Numero elementi per Bloom Filters:** 500

Numero prova	Inserimento	Aggiornamento	Ricerca
1	68.118	30.444	21.095
2	65.839	29.829	20.952
3	68.694	30.454	22.913
4	68.694	31.154	26.060
5	67.555	29.695	20.953
6	67.600	29.915	21.156
7	67.630	30.272	21.097
8	66.243	29.957	21.916
9	69.488	30.155	22.426
10	68.849	30.117	22.328

Tabella 4.6: Test C su 1000 Bloom Filters di 500 elementi.

Inserimento: 67,871 ms
Aggiornamento: 30,1992 ms
Ricerca: 22,089 ms

Con il crescere della taglia dell'input, le operazioni di inserimento ed aggiornamento sono più veloci in Java. C invece, risulta più performante sull'operazione di ricerca.

4.4.2 Performance

I test hanno avuto un seguito anche con diversi numeri di elementi per ciascun Bloom Filter. Anche variando la falsa positività, le conclusioni tratte sono le stesse: l'inserimento e l'aggiornamento di un Bloom Filter performa meglio nell'implementazione Java su una taglia di indicizzazione dal 1000 in poi. C, però, ha una buona prestazione nel tempo di ricerca, battendo Java. Dovendo dare un giudizio, possiamo sostenere che un' implementazione Java è ottimale per il

mantenimento della struttura, ossia risponde meglio alla scrittura del Flat Bloofi. Se invece l'uso del Flat prevede maggiormente una "consultazione", allora C è la scelta migliore.
La diversità dei comportamenti appena mostrati è stato un punto di interesse nel corso della nostra indagine. Ci siamo soffermati sul Flat in C, misurando il tempo medio delle sottoprocedure che rallentano l'insert e l'update. La causa primaria del deficit risiede nella funzione *nextSetBit2* che la maggior parte dello sforzo computazionale sia nell'aggiornamento che nell'inserimento. Entrambe le operazioni hanno in comune l'operazione *setBloomAt*, una procedura che permette di risettare i bit del Flat rispetto al BitSet di un Bloom Filter. *setBloomAt* ha come sottoprocedura *nextSetBit2*, il quale diventa quindi un collo di bottiglia.

```
for (int k = nextSetBit2(bitset, 0); k >= 0; k = nextSetBit2(bitset, k+1)) {
    mybuffer[k] |= mask;
}
```

Figura 4.8: La funzione nextSetBit2 nella procedura SetBloomAt

4.5 CBloofi e CFlatBloofi

Avendo a disposizione le implementazioni C del Bloofi e del Flat Bloofi, non potevamo esimerci dal testing fra i due. Il lavoro svolto prevede le stesse dinamiche con il confronto in Java. Riportiamo le medie del CBloofi su diverse dimensioni dell'input:
1 Bloom Filter con 5 elementi

Inserimento: 0.011ms
Aggiornamento: 0.011ms
Ricerca: 0.002ms
Nel caso unitario, le strutture sono quasi equivalenti. Il flat però è più lento nelle operazioni, poiché deve creare un flat intero da 64 celle per il singolo bloom filter, invece il Bloofi avendo pochissimi nodi, ha un accesso rapido.

100 Bloom Filters con 50 elementi ciascuno

Inserimento: 13.385ms
Aggiornamento: 2.453ms
Ricerca: 4.087ms

Con 100 Bloom Filter si ha un lieve vantaggio per il Flat Bloofi.

1000 Bloom FIlter con 500 elementi ciascuno

Inserimento: 162.464ms
Aggiornamento: 30.663ms
Ricerca: 549.82ms

Salvo l'inserimento, il Flat performa al meglio con 1000 Bloom Filters aggiunti, arrivando a performare addirittura 27 volte meglio rispetto al Bloofi sull'operazione search. In più, si fa noto che il tempo di ricerca del Bloofi, in questo caso, è molto variabile, poichè oscilla dai 300 ms fino a picchi di 2000 ms, mentre il flat si ha un andamento più costante.
Ma cosa succede se aumentassimo ancora la taglia dei Bloom Filters? Il testing si è prolungato su input di valori delle migliaia. Come sarà confermato teoricamente nel capitolo successivo, il Bloofi recupera terreno con il crescere del numero di Bloom Filter indicizzati. Di seguito riportiamo l'andamento dei tempi medi di ricerca delle due implementazioni, con l'incremento di Bloom filters (per convenzione ciascun Bloom ha 500 elementi).

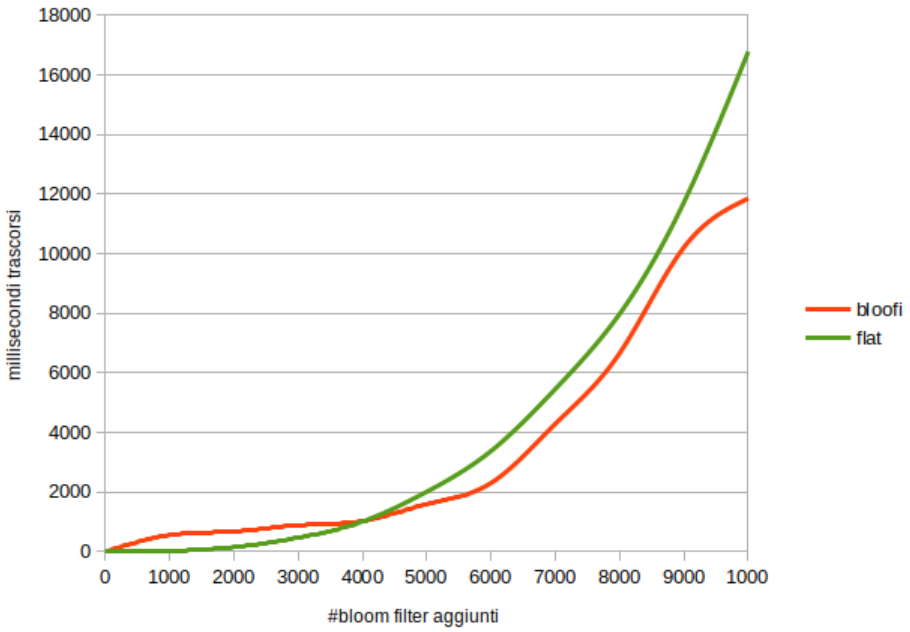

Figura 4.9: Tempi di ricerca con il crescere del numero di Bloom Filter.

4.6 Conclusioni

Tirando le somme, si è dedotto che: una implementazione in C che ha cura della gestione della memoria, riesce a sprecare di meno rispetto al lavoro automatizzato del Garbage Collector di Java. Il C performa bene nelle operazioni di ricerca, un po' meno nell'aggiornamento e nell'inserimento, anche se, abbiamo fatto riferimento a come la procedura $nextSetBit2$ rallenti entrambe le operazioni e a come una riscrittura migliore di quest'ultima possa portare ad un ulteriore vantaggio nell'implementazione in C. L'esecuzione della remove rimane una questione ancora aperta, si invita alla risoluzione del bug affinché si possa affrontare un testing delle performance anche per l'eliminazione di un Bloom Filter dalla struttura. Concludiamo il capitolo, facendo dei cenni su ulteriori possibili testing: la gestione di Bloom Filters multidimensionali ha senso se si garantisce la correttezza del funzionamento dei Bloom Filter e che non diano troppi falsi positivi, rispettando la percentuale data in input. Per questo motivo, è possibile fare un test di falsa positività dei Bloom Filter all'interno di un Bloofi o Flat Bloofi, impostando, una soglia massima di falsi numeri positivi tollerati e, questo limite, non deve essere superato dalla media dei falsi positivi scovati.

Capitolo 5

Naive, Bloofi, Flat Bloofi a confronto

5.1 Valutazione delle prestazioni

In questo capitolo viene data una visione più ampia di quali sono i criteri di performance per Bloofi, Flat Bloofi e Naive e del loro comportamento al variare di alcuni parametri dati in input. Si fa presente che i risultati teorici che stanno per essere mostrati, assieme ai grafici, sono tratti da [3] ed eseguiti su una CPU Intel Xeon da 2.50 GHz con 32 GB di RAM.

- **Costi di ricerca per un Bloom Filter:** espresso come il numero di Bloom Filter controllati prima di trovare quello che risponde alla query di ricerca.
- **Tempo di ricerca:** Tempo di risposta della query di ricerca.
- **Costi di archiviazione** La quantità dello spazio di richiesto per memorizzare i Bloom Filters e l' intera struttura. Per il Bloofi, si ottiene moltiplicando i bytes occupati da un bloom filter per il numero di nodi. Per il Flat, il rapporto bytes per Bloom Filter per il numero di Bloom deve essere approssimato al più vicino muliplo di 64. Questo perché viene istanziato un Flat intero alla volta. Per il Naive, semplicemente il rapporto bytes per Bloom moltiplicato il numero di Bloom Filters.
- **Costi di mantenimento:** E' la media dei Bloom Filter acceduti durante le operazioni.
- **Tempi di manutenzione:** è il tempo medio di esecuzione di una delle operazioni.

5.2 Variazione del numero di Bloom Filters

Si assuma N come il numero di Bloom Filter indicizzati e si consideri l'esecuzione di una query di ricerca. Bloofi mostra una crescita logaritmica se la probailità di falsa positività rimane sotto l'1% per il Bloom Filter che si trova alla radice. Un' elevata percentuale di falsa positività potrebbe far aumentare il tempo di ricerca, specialmente se la percentuale è alta all'inizio dell'albero; Bloofi potrebbe intraprendere fin da subito un percorso sbagliato e rallentare di molto la query. In ogni caso, anche con $N > 10000$, Bloofi risulta comunque rispondere meglio rispetto ad una semplice soluzione Naive. L'unica condizione in cui Naive e Bloofi hanno performance simili è quando N ha assume un valore basso. Sempre per valori di N più o meno bassi (anche se stiamo parlando dell'ordine delle centinaia/migliaia), come già accennato e testato nel Capitolo 4, il Flat Bloofi ha prestazioni migliori, questo perché sfrutta, appunto, il parallelismo tra bit.

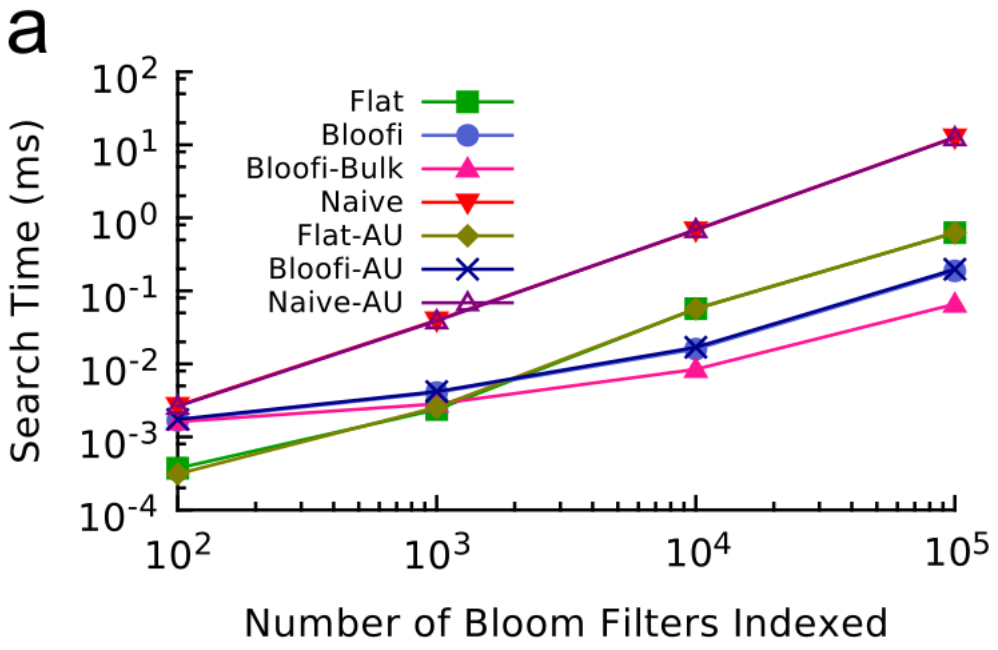

Figura 5.1: Implementazioni al variare del numero di Bloom Filters.

Per quanto riguarda il costo di ricerca, è un criterio che non viene preso in considerazione da una soluzione Flat o una Naive, poiché per queste tipologie di strutture vengono visitati tutti i Bloom Filter durante una search, cosa che invece non avviene per il Bloofi, che riesce a "potare" diversi cammini man mano che prosegue la sua discesa nei vari sottoalberi.

5.3 Costi di mantenimento

Per una soluzione Naive, il costo di mantenimento è trascurabile; la struttura Naive consiste in una semplice lista dove sono contenuti tutti i Bloom Filters. I costi invece per un Flat aumentano lievemente con l'aumentare del numero di Bloom. In realtà, per un Flat Bloofi, il peso di un operazione è dipende principalmente dalla dimensione di ciascun Bloom Filters e dal numero di bit settati ad 1. Per un Bloofi, una insert o una delete diviene sempre più costosa con l'aumento di N, fino a diventare meno efficiente anche del Naive (Fig.5.5). Ecco perchè a volte si predilige il Flat. Per quanto riguarda il costo di un aggiornamento, il Bloofi sembra assestarsi verso i 10000 Bloom Filters, probabilmente perché tenderà a fare sempre meno operazioni di split.

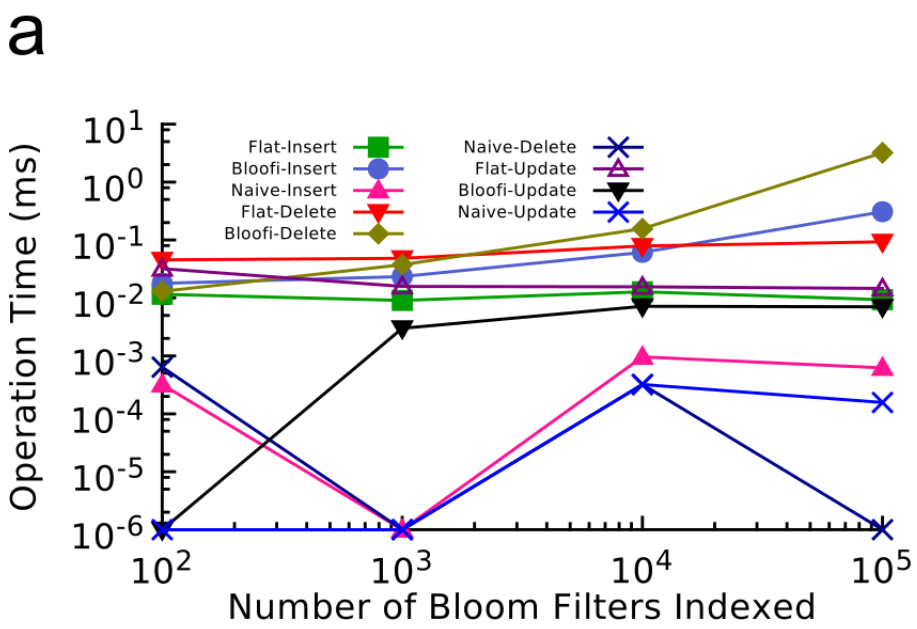

Figura 5.2: Panoramica dei costi delle operazioni.

Ricapitolando, per Bloofi l'operazione meno costosa è l'aggiornamento; la remove, assieme l'inserimento, le più dispendiose con l'aumentare di N. Il Flat non ha problemi ad eseguire inserimenti, aggiornamenti e rimozioni. La remove tendezialmente può risultare più veloce in un Bloofi sotto i 10000 Bloom Filter inseriti, ma con l'aumentare di quest'ultimi il Flat torna in vantaggio.

5.4 Variazione della taglia di un Bloom Filters

Variare la taglia di un Bloom Filters, significa variare il numero di elementi che si aspetta. Settando tal valore a 10000 Bloofi in ogni caso performa meglio di una soluzione Naive in termini di costi. Si fa presente come un Bloom Filter, se aumenta il numero di Bloom che si aspetta, allora la probabilità di incappare in falsi positivi diminuisce, a discapito dello spazio richiesto. La diminuzione della percentuale di falsa positività nei nodi vicino la radice porta quindi inevitabilmente a prestazioni migliori. Tuttavia, se il Bloofi non ha euristica nel potare i sottoalberi di cammini sbagliati, il tempo di ricerca per un Flat Bloofi è minore rispetto ad un Bloofi con l'aumentare della taglia del Bloom Filter (sempre con valori non troppo alti di N). Questo accade sempre grazie alla sua capacità di sfruttare il parallelismo tra bit.

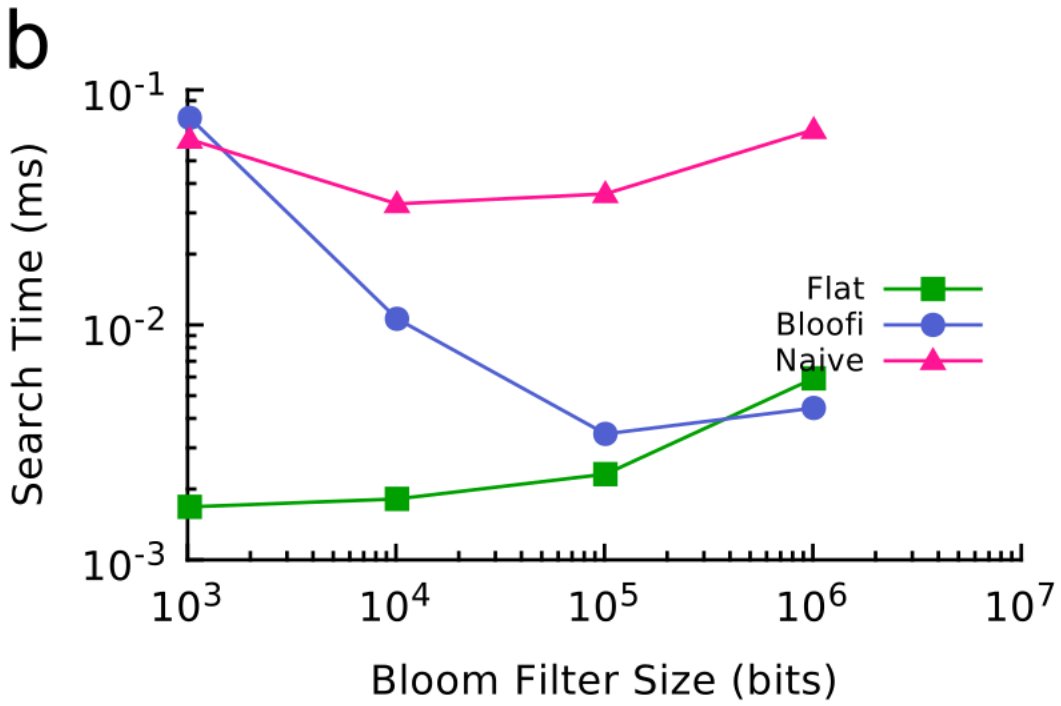

Figura 5.3: tempo operazione di ricerca al variare della taglia dei Bloom Filter.

5.5 Variazione della probabilità di falsi positivi

Discorso contrario a quello precedente,se la probabilità dei falsi positivi aumenta, allora diminusce la taglia del Bloom filter, e quindi il numero di elementi che si aspetta.

5.6 Variazione del numero di elementi

Flat-Bloofi offre le migliori prestazioni in questo caso, poiché beneficia della località di memoria e del livello di parallelismo tra i bit. Ricordiamo che con località si intende che gli accessi sono raggruppati nel tempo e nello spazio. Inoltre, la variazione di N sembra essere indipendente dal numero di elementi di ciascun Bloom nel calcolo delle prestazioni. Il Bloofi invece, è più sensibile alla variazione degli elementi quando si supera l'ordine dei 10000.

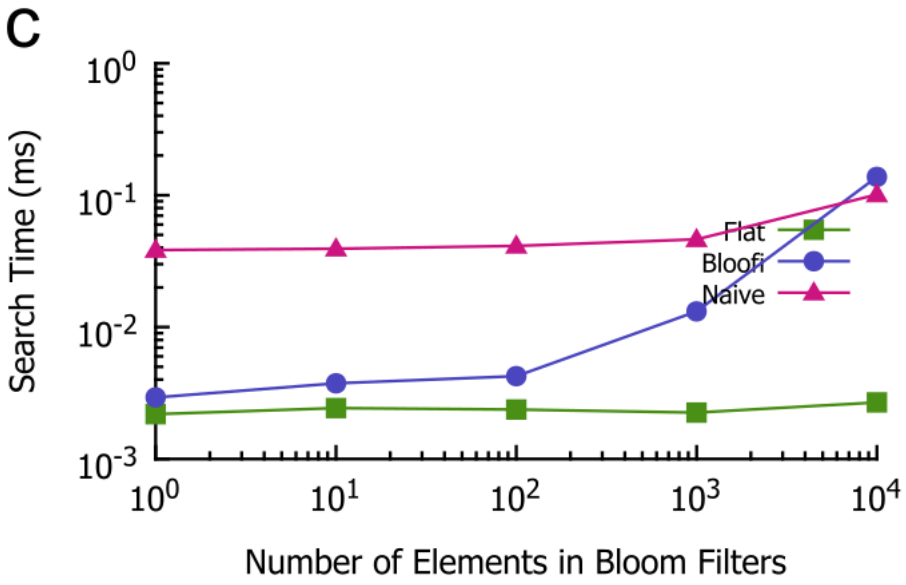

Figura 5.4: Tempo di ricerca al variare del numero di elementi per ogni Bloom Filter.

5.7 Variazione della distribuzione dei dati

Un aspetto fin ora non preso in considerazione è la distribuzione dei dati in un Bloom Filters. E' possibile fare una stima analizzando due casi: uno in cui l'inserimento dei dati all'interno avviene ad intervalli regolari nel BitSet oppure in modo randomico. Ciò che avviene è che, nel caso degli intervalli regolari, quando vengono inseriti nuovi elementi nel BitSet di un Bloom, questi andranno a settare '1' in locazioni più equidistanti possibili. Invece, nel caso randomico, con alta probabilità ci saranno sovrapposizioni di riferimenti a locazioni con bit già settati ad 1. Ovviamente il riempimento casuale è il più comune. Le prestazioni sono peggiori in questa casistica perchè un bit a 1, può far riferimento a più

elementi diversi, quindi c'è una tendenza alla falsa positività. Tuttavia, non è un parametro che influisce molto rispetto ad altri.

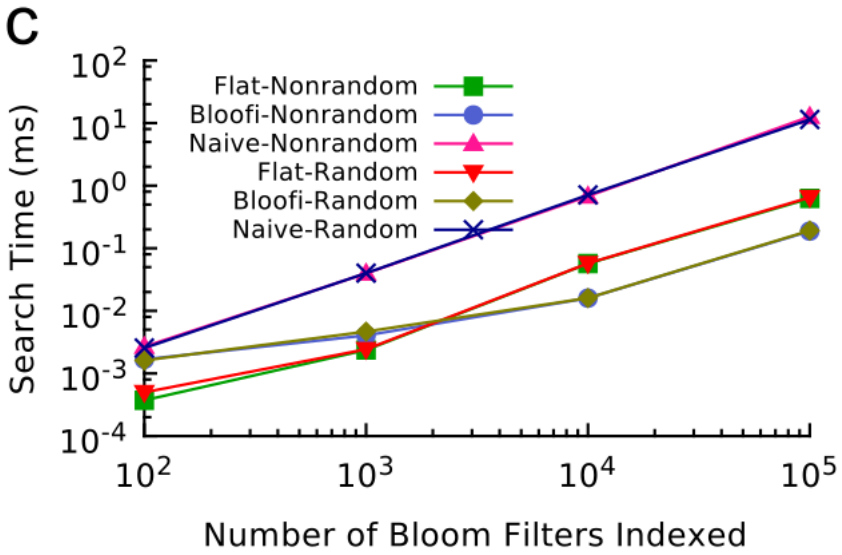

Figura 5.5: Tempo di ricerca al variare della distribuzione dei dati nei Bloom Filters.

5.8 Conclusioni

Abbiamo descritto come le varie implementazioni sono sensibili ai cambiamenti dei parametri più importanti. Inizialmente, l'obiettivo è stato indicizzare più Bloom Filter possibili, questo giustifica la nascita di Bloofi. I vari studi condotti hanno evidenziato, però, che la soluzione non è ottima in tutte le circostanze. Questo poteva risultare un limite per l'indicizzazione dei Bloom Filters. Grazie alla soluzione Flat, un' applicazione basata su Bloom Filters nell'ordine delle poche migliaia può godere di un rendimento massimo sui tempi di ricerca. Facciamo riferimento ad un altra casistica, in cui il numero di Bloom Filter è alto: è vero che, con $N > 10000$, Bloofi sarebbe la scelta da adottare ma, se l'applicazione richiedesse frequenti operazioni di inserimento o cancellazione, la scelta verterebbe di nuovo sul Flat Bloofi. Infine, mostriamo un ulteriore scenario, in cui si è stabilito che, ogni Bloom Filter, per una serie di circostanze, deve avere un alto numero di elementi da memorizzare, magari perchè vi è sistema cloud e l'intero database di un nodo fa riferimeno ad un unico Bloom Filter. Anche qui, il Flat è nettamente superiore se non sfora i 10000 Bloom Filter, quindi i 10000 nodi.

Capitolo 6

Conclusioni e sviluppi futuri

6.1 Obiettivi raggiunti

Il fine dell'elaborato è stato evidenziare le problematiche relative ai limiti dell' indicizzazione dei Bloom Filter e trovarne una possibile soluzione. La soluzione banale (Naive), ossia la memorizzazione dei Bloom in una lista, non aveva soddisfatto i requisiti prefissati. La nostra attenzione è stata successivamente catturata dalla proposta del Bloofi, mostrato nell'articolo "Multidimensional Bloom Filters" [3]. Il nostro lavoro di tesi si è basato sullo studio di esso, analizzando i comportamenti della struttura e anche della nuova implementazione Flat. Dopo aver dedotto la logica algoritmica applicata all'implementazione in Java già esistente [2], il nostro interesse si è spostato sullo studio del codice e sulla ricerca di eventuali miglioramenti. Durante la nostra analisi, abbiamo notato, oltre ad un' anomalia nell' operazione di rimozione, alcune logiche comportamentali che sono legate a Java che ostacolavano ulteriori potenziamenti che potevano essere implementati solo programmando ad un livello più basso. Per questo motivo abbiamo scelto di proporre una soluzione alternativa in C. Le potenzialità del linguaggio ci hanno permesso di lavorare direttamente sulla memoria e il risultato è stato positivo; l'Heap utilizzato è stato ridotto dai 38 MB in Java ai 25 MB in C. Il secondo step è stato il confronto tra il codice sviluppato con quello in [2]. Gli esperimenti hanno portato ad un margine di miglioramento sull'operazione di ricerca a discapito dell'inserimento ed aggiornamento, un trade-off accettabile se si considera che in parecchie applicazioni l'operazione più frequente è proprio la search. Tuttavia, rivistando il nostro codice, abbiamo intuito quale poteva essere il tallone d'Achille della nostra struttura, il quale risiedeva una sottoprocedura che si mostra troppo onerosa. Perciò, anche se C attualmente non è la soluzione ottimale in qualunque contesto di applicazione, abbiamo aperto un varco di stu-

dio dove poter insistere, poiché un' implementazione ancora più accurata in C potrebbe portare a risultati ancor più interessanti. Per queste ragioni, possiamo ritenerci soddisfatti del lavoro di tesi che si è sostenuto.

6.2 Possibili potenziamenti

Oltre al lasciar spazio a nuovi studi verso il potenziamento del codice C implementato, mettiamo in evidenza altre idee e possibili sviluppi che potrebbero essere messi in pratica in futuro. Facendo riferimento all'implementazione Flat Bloofi, si fa presente di come essa sprechi ad ogni nuova istanza di un flat dei bit inutilizzati. Un' idea sarebbe poter sfruttare l'informazione inutilizzata per poter memorizzare dei meta dati della struttura in modo che una query di ricerca, inserimento, aggiornamento o rimozione possa consultare tali bit per velocizzare l'operazione, il tutto senza dover sprecare troppi cicli di clock. Un' altra possibile miglioria potrebbe essere l'introduzione di più thread che cooperano al fine di migliorare le prestazioni.

6.3 Applicazioni

Nell' elebarato abbiamo spesso rimarcato come implementazioni diverse sono ottimali in contesti differenti, quindi si propone lo sviluppo di un' ambiente settato per l'esecuzione di programmi con molteplici Bloom Filters costrutito nel seguente modo: in funzione dei parametri ricavati da questo lavoro, un' algoritmo dovrà calcolare l'implementazione Bloofi da scegliere in base a dei parametri di archiviazione e mantenimento forniti in input. Se le variabili dovessero distaccarsi troppo dalle condizioni iniziali, il programma deve rielaborare le nuove condizioni dell'ambiente e trasformare la struttura corrente in quella più adatta.

Bibliografia

[1] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] Adina Crainiceanu and Daniel Lemire. Bloofi: A java implementation of multidimensional bloom filters. *https://github.com/lemire/bloofi*.

[3] Adina Crainiceanu and Daniel Lemire. Bloofi: Multidimensional bloom filters. *Information Systems*, 54:311–324, 2015.

[4] Daniel Lemire. A simple implementation of bitset library in c. *https://github.com/lemire/cbitset*.

[5] Saibal K Pal and Puneet Sardana. Bloom filters & their applications. *International Journal of Computer Applications and Technology*, 1(1):25–29, 2012.

[6] Ripon Patgiri, Sabuzima Nayak, and Samir Kumar Borgohain. Role of bloom filter in big data research: A survey. *arXiv preprint arXiv:1903.06565*, 2019.

[7] Fabiano Priore. Cbloofi. *Tesi di Laurea in Informatica, Univ. degli Studi di Salerno*, 2020.

[8] Alan Sexton and Hayo Thielecke. Reasoning about b+ trees with operational semantics and separation logic. *Electronic Notes in Theoretical Computer Science*, 218:355–369, 10 2008.

[9] Alex C Snoeren, Craig Partridge, Luis A Sanchez, Christine E Jones, Fabrice Tchakountio, Stephen T Kent, and W Timothy Strayer. Hash-based ip traceback. *ACM SIGCOMM Computer Communication Review*, 31(4):3–14, 2001.

[10] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.

[11] Jyri J. Virkki. A simple and small bloom filter implementation in plain c. *https://github.com/jvirkki/libbloom*.