UNIVERSITÁ DI PISA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Indicizzazione e compressione di flussi di rete

Candidato:
Lorenzo Vannucci

Relatore:
Luca Deri

Anno Accademico 2015–2016

Sommario

Lo scopo di questa tesi è di presentare un nuovo tipo di indice bitmap sviluppato durante il tirocinio, che permetta la ricerca e l'inserimento di flussi di rete in modo efficiente e che occupi uno spazio disco limitato. A questo scopo è stata sviluppata una nuova codifica bitmap ottimizzata al meglio solo ed esclusivamente per la creazione di indici bitmap. Infine viene descritta una possibile soluzione per la compressione dei dati a blocchi di 1024 Byte, che permette quindi di estrarre i record selezionati senza dovere leggere e decomprimere tutti i flussi di rete.

Indice

Capitolo 1

Introduzione

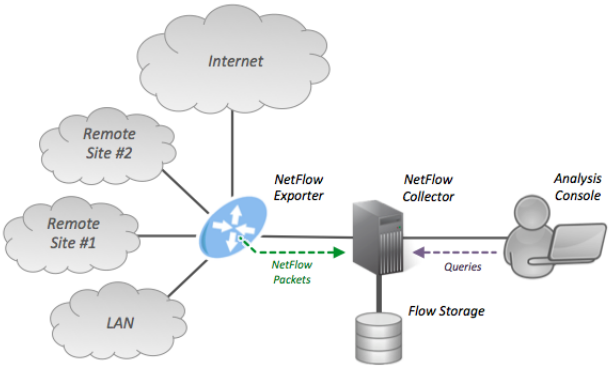
Le reti di calcolatori stanno crescendo velocemente in dimensione e complessità. Secondo le analisi e previsioni Cisco [1] nel 2020 il traffico internet triplicherà rispetto al 2015 e raggiungerà 1.9 Zettabyte annui. In questo scenario, al fine di soddisfare le aspettative degli utenti sono necessari meccanismi e infrastrutture di monitoraggio per rilevare problemi di configurazione e guasti, per misurare le prestazioni e per rendere possibili reazioni tempestive a minacce di sicurezza.

Come specificato in [2], NetFlow [3] e sFlow [4] sono gli attuali standard per costruire applicazioni per il monitoraggio delle reti. Entrambi si basano sul concetto di sonda e di collezionatore: la sonda analizza il traffico di rete ed emette flussi (informazioni su una sequenza di pacchetti con alcune proprietà in comune), che vengono inviati a un collezionatore, il quale ha il compito di memorizzare i flussi producendo archivi che forniscono funzionalità di indicizzazione e ricerca.

Lo scopo di questo tirocinio è stato quello di sviluppare un nuovo tipo di indice bitmap molto efficiente basato su una nuova codifica bitmap e un metodo per la compressione dei dati a blocchi di 1024 Byte, in modo tale da fornire gli strumenti per la realizzazione di un collezionatore di flussi di rete che indicizza e comprime i flussi, cha sia efficiente nella ricerca e nell'inserimento e che permetta quindi di estrarre i record richiesti senza dover leggere e scansionare grandi quantità di dati.

1.1 Flussi di rete e architettura NetFlow

Un flusso di rete è un'aggregazione di pacchetti IP che hanno un insieme di proprietà in comune (per esempio IP e porta) [5]. L' architettura NetFlow [3] è basata sul concetto di flusso ed è composta da tre componenti principali:



la sonda (*NetFlow exporter*):
ha il compito di aggregare pacchetti in flussi di rete ed esportarli verso uno o più collezionatori. Il protocollo di esportazione è NetFlow e il formato del pacchetto del flusso dipende dalla versione del protocollo [5].

il collezionatore (*NetFlow collector*):
i flussi vengono inviati dalla sonda in formato NetFlow al collezionatore. Quest'ultimo ha il compito di archiviare i flussi ricevuti e fornire funzionalità di ricerca e analisi.

l'applicazione di analisi (*Analysis console*):
ha il compito di effettuare query al collezionatore per verificare il corretto funzionamento della rete.

1.1.1 Inizio e fine di un flusso

Un flusso viene creato ogni volta che arriva un pacchetto che non è aggregabile a nessun flusso presente nella cache della sonda e termina quando si verifica una di queste condizioni:

- La comunicazione è terminata.
- Il flusso dura troppo (30 minuti di default).
- Il flusso non è più attivo (15 secondi di default).
- La cache utilizzata dalla *sonda* è piena e il gestore della cache deve eliminare (esportandoli al collezionatore) i dati.

1.1.2 L'importanza del monitoraggio dei flussi

L'attività di monitoraggio e analisi dei flussi è molto importante poiché grazie ad essa è possibile individuare le cause che provocano malfunzionamenti/rallentamenti della rete, in particolare:

- Sapere di che tipo è il traffico (SMPTP, HTTP, file sharing...).
- Tracciare la sorgente degli attacchi DoS.
- Sapere gli host che hanno servizi attivi.
- Trovare gli host che generano maggior traffico.

1.2 Motivazione e obiettivo del lavoro

Nelle reti ad alta capacità il numero di flussi generati da una sonda può essere molto elevato e il problema di riuscire a salvare su memoria persistente i flussi e fornire funzionalità di ricerca e analisi non è un problema banale. Secondo AppNeta [6] infatti, il numero dei flussi al secondo N generato da una sonda rispetto al traffico T (espresso in bit al secondo) della rete è pari a:

$$N = (50 flussi \times T)/10 Mbits$$

Ciò implica che nelle reti 10/40Gbps vengono generati più di 50.000 flussi al secondo.

I normali DBMS relazionali essendo disegnati per scopi più generali anche se offrono affidabilità e la possibilità di scrivere query espressive, non sono adatti a collezionare un numero così elevato di record in real-time [2].

Altri strumenti più specifici per questo scopo come ad esempio Nfdump [7], flow-tools [8], Flow-Scan [9], Stager [10] e SiLK [11] archiviano i dati in file binari (chiamati flat file) che possono essere opzionalmente compressi e forniscono la possibilità di filtrare, estrarre e aggregare i record di flussi secondo specifici criteri. Anche se questi strumenti risultano più veloci (sia durante l'inserimento dei dati, sia durante il processamento di una query) [12] [13] e occupano meno spazio disco rispetto ai DBMS tradizionali, non avendo un sistema di indicizzazione per ogni query deve essere fatta un'intera lettura e scansione dei dati e quindi nelle reti che producono un numero elevato di flussi impiegano molto tempo per eseguire una query (misurato in minuti se non in ore) [14] [15].

In conclusione l'attuale stato dell'arte per il collezionamento e l'analisi dei flussi di rete nelle reti ad alta capacità non fornisce una soluzione ottimale.

Lo scopo di questo tirocinio è stato di sviluppare un sistema di indicizzazione e compressione che occupi uno spazio limitato e che sia efficiente durante l'inserimento dei dati e durante la ricerca in modo tale da fornire gli strumenti per la realizzazione di un collezionatore che risulti migliore dello stato dell'arte attuale.

1.3 Risultati originali del lavoro

In questa sezione vengono brevemente presentati i risultati raggiunti alla fine del tirocinio.

- è stato sviluppato un nuovo schema di codifica bitmap ottimizzato solo ed esclusivamente per la creazione di indici su attributi ad alta cardinalità ed è stato disegnato, implementato e validato un sistema di indicizzazione bitmap per i flussi di rete basato sulla nuova codifica bitmap sviluppata.

- è stato disegnato, sviluppato e implementato un nuovo algoritmo di compressione dati basati su alcune proprietà che ci sono nei flussi di rete che comprime a blocchi di 1024 Byte. In questo modo è possibile estrarre i record selezionati senza dover leggere e decomprimere tutti i dati.

1.4 Struttura della tesi

La tesi è strutturata nel seguente modo:

Stato dell'arte: nel capitolo 2 viene presentato l'attuale stato dell'arte delle codifiche per la creazione di indici bitmap su attributi ad alta cardinalità e delle codifiche di compressione bitmap utilizzate per ridurre lo spazio occupato e per migliorare le performance delle operazioni logiche.

Indicizzazione flussi: nel capitolo 3 viene motivato e descritto il meccanismo di indicizzazione bitmap dei flussi, viene descritta una nuova codifica bitmap e confrontata con l'attuale stato dell'arte di codifiche per indici bitmap e infine viene analizzata, descritta e testata una possibile soluzione per ridurre ulteriormente lo spazio occupato dagli indici bitmap.

Compressione flussi: nel capitolo 4 viene presentata una breve analisi sulle proprietà dei flussi di rete, vengono brevemente descritte alcune tecniche di compressione su interi e viene descritto e validato un nuovo tipo di compressione a blocchi di 1024 Byte.

Conclusioni: nel capitolo 5 presentate le conclusioni del lavoro svolto.

Capitolo 2

Stato dell'arte

Gli indici bitmap sono tradizionalmente usati su attributi a bassa cardinalità, quando il numero di valori distinti che può assumere un determinato attributo è limitato. In questi casi dato un attributo di cardinalità C per creare un indice su un numero N di valori inseriti è sufficiente avere C bitmap, ognuna messa in corrispondenza con un possibile valore, e ogni volta che viene inserito un record di riga i andare a settare il bit *i-esimo* nella bitmap corrispondente [16], ad esempio per costruire un indice sull'attributo *Sesso* basta utilizzare due bitmap:

Numero riga	Sesso	Maschio	Femmina
0	M	1	0
1	M	1	0
2	F	0	1
3	M	1	0
4	F	0	1
5	F	0	1

In questo modo per selezionare tutti i record con *Sesso* = *Maschio* è sufficiente estrarre tutte le ennuple di riga uguale alla posizione dei bit settati a *1* nella bitmap corrispondente al valore *Maschio*. Questo metodo di indicizzazione, che in letteratura si chiama *equality encoding* ed è supportato dalla maggior parte dei DBMS commerciali, risulta essere molto efficiente in spazio e velocità quando il numero di valori distinti su un attributo è limitato [17] ma ha un costo esponenziale rispetto alla cardinalità dell'attributo poiché per K possibili valori servono K bitmap, ognuna di lunghezza N (dove N è il numero di record inseriti).

Dato che in questi casi la dimensione degli indici sarebbe troppo grande, esistono in letteratura tecniche per ridurne la dimensione basate su metodi di

indicizzazione più complessi che riducono il numero di bitmap necessarie [18] [19] [20] [21] e su algoritmi di compressione per ridurre lo spazio occupato da ogni singola bitmap [22] [23] [24] [25] [26].

In questo capitolo vengono descritti i principali algoritmi utilizzati per codificare e comprimere un indice bitmap.

2.1 Metodi di Indicizzazione Bitmap

In letteratura esistono tre tipi di indicizzazopme bitmap [19] [27]: quelli ottimizzati per le *equality queries* come *"A = 3"* [16] (come ad esempio l'*equality encoding* descritto precedentemente), quelli ottimizzati per le *one-sided range queries* come *"A <5"* [19] e quelli ottimizzati per le *two-sided range queries* come *"3 <A <10"* [19].

Dato che per i flussi di rete non c'è la necessità primaria di indicizzare i dati per fare *one-sided/two-sided range queries*, questi metodi in questo tirocinio non sono stati presi in considerazione. In questa sezione vengono descritti quindi solo i principali schemi di indicizzazione su attributi interi ad alta cardinalità ottimizzati per le *equality queries*.

2.1.1 K-of-N equality encoding

K-of-N equality encoding [28] è un metodo di indicizzazione bitmap che dato N numero di bitmap e K numero di bit settati a 1 per ogni valore indicizzato, riesce ad indicizzare $L = \binom{N}{K}$ valori distinti creando una corrispondenza biunivoca tra ogni possibile sequenza di N bit con K bit a 1 e ogni possibile valore, in questo modo per risolvere un'*equality query* basta estrarre e mettere in *and* K bitmap.

Dato che ogni bit può essere solo 0 o 1 ne consegue che il numero di tutti i possibili sottoinsiemi di K elementi presi da un insieme di N elementi (che è proprio per definizione il coefficiente binomiale $\binom{N}{K}$ [29]) è uguale al numero di sequenze di N bit con K bit a 1. Di seguito è riportato un esempio di indicizzazione utilizzando lo schema *K-of-N equality encoding* con $N = 6$ (numero di bitmap) e $K = 2$ (numero di bit settati a 1) per ogni valore inserito) su un attributo A con $|A| = 15$, per semplicità chiameremo i valori di A con *v1, v2, ... , v15*.

Numero riga	Valori di A	b1	b2	b3	b4	b5	b6
0	v1	1	1	0	0	0	0
1	v4	1	0	0	0	1	0
2	v6	0	1	1	0	0	0
3	v15	0	0	0	0	1	1
4	v8	0	1	0	0	1	0
5	v1	1	1	0	0	0	0
6	v3	1	0	0	1	0	0

In questo modo per trovare ad esempio le posizioni dei record con $A = v1$ basta fare $b1 \wedge b2 = \{0, 5\}$.

Sapendo che $\frac{N^K}{K^K} \leq \binom{N}{K} \Rightarrow (\frac{N_1^K}{K^K} = \binom{N_2}{K} \iff N_1 \geq N_2)$

è possibile quindi conoscere il numero di bitmap N necessario per rappresentare L valori con K fissato utilizzando la formula $N = \sqrt[K]{L} \times K$. Questa formula rappresenta però solo un' approssimazione per eccesso di N, ad esempio per indicizzare $L = 65,536$ con $K = 2$, $\sqrt[2]{65,536} \times 2 = 512$ bitmap mentre in realtà è sufficiente utilizzare $N = 363$ bitmap poiché $\binom{363}{2} = 65,703$.

In conclusione, rispetto a *equality encoding* con *K-of-N equality encoding* è possibile diminuire notevolmente il numero di bitmap necessarie per la creazione di un indice su un attributo ad alta cardinalità, in particolare poiché $\binom{N}{K} = \binom{N}{N-K}$ [29] date N bitmap il numero massimo di valori distinti che è possibile indicizzare è dato da $K = \frac{N}{2}$ e quindi ad esempio con $N = 32$ il numero massimo di valori distinti che si possono indicizzare è $L = \binom{32}{16} = 601,080,390$.

2.1.2 Multi-Component equality encoding

Multi-Component equality encoding [30] [18] è un metodo di indicizzazione bitmap che permette di indicizzare L valori distinti con $K \times N$ bitmap, decomponendo L in K componenti ognuna delle quali grandi $\lceil \sqrt[K]{L} \rceil = N$ bitmap. Per esempio per indicizzare $A = \{0, 1, ..., 99\}$ è possibile definire due componenti C_1, C_2, che rappresentano la prima e la seconda cifra decimale. Ogni componente è dunque composta da $\lceil \sqrt[2]{100} \rceil = 10$ bitmap messe in corrispondenza biunivoca con una cifra decimale ed è quindi possibile risolvere un'*equality query* "$A = i_2 i_1$" (con i_1, i_2 = alla prima e alla seconda cifra decimale del valore che vogliamo trovare) mettendo in *and* la $i_1 - esima$ bitmap della componente C_1 e la $i_2 - esima$ bitmap dalla componente C_2.

Sapendo che per rappresentare un attributo A con cardinalità L servono $\lceil \log_2 L \rceil = B$ bit, una tecnica molto semplice per indicizzare A utiliz-

zando *Multi-component equality encoding* consiste nel decomporre B bit in K componenti $K_1, K_2, .., K_K$ grandi rispettivamente $B_1, B_2, .., B_K$ bit (con $B_1+B_2+...+B_K = B$) e quindi $2^{B_1}, 2^{B_2}, .., 2^{B_K}$ bitmap. Quindi ad esempio su un attributo A con 256 possibili valori distinti, poiché servono $\lceil \log_2 256 \rceil = 8$ bit per rappresentare ogni valore, è possibile suddivire 8 bit in 4 gruppi da 2 bit e creare quindi 4 componenti grandi ognuna $2^2 = 4$ (numero di valori distinti rappresentabili con 2 bit) bitmap. In questo modo per costruire l'indice bitmap ogni volta che viene inserito un valore v , basta decomporlo in 4 blocchi (b_1, b_2, b_3, b_4) da 2 bit e settare a 1 per ogni componente i, la $j - esima$ bitmap con $j = valore\ espresso\ da\ b_i$ mentre per eseguire *equality-query* $A = v$ basta mettere in *and* la $j - esima$ bitmap di ogni componente i (con j ottenuto nel solito modo).

In conclusione, rispetto a *equality encoding* con *Multi-Component equality encoding* è possibile diminuire notevolmente il numero di bitmap necessarie per la creazione di un indice su un attributo ad alta cardinalità, in particolare il numero massimo di valori distinti che è possibile indicizzare date N bitmap è 2^N, in questo caso l'attributo viene scomposto in N componenti ognuna composta da 1 sola bitmap e ogni bitmap b_i viene settata a 1 ogni volta che l'$i - esimo$ bit di un valore v inserito è uguale a 1, per eseguire un'*equality query* devono essere quindi lette e messe in *and* tutte le bitmap che rappresentano l'indice.

2.2 Metodi di Compressione Bitmap

Per diminuire la dimensione di un indice bitmap è possibile utilizzare la compressione. Dato che per rispondere a una query è necessario estrarre solo una parte delle bitmap che rappresentano l'indice, per non dover leggere e decomprimere tutto l'indice la compressione è solitamente applicata a ogni singola bitmap.

Poiché utilizzando i normali metodi di compressione per comprimere ogni bitmap aumenterebbe in modo significativo il tempo di risoluzione di una query (poiché ogni bitmap estratta dovrebbe essere decompressa), esistono in letteratura algoritmi di compressione bitmap che permettono di eseguire operazioni logiche sulle bitmap senza decomprimerle e che risultano quindi più efficienti [31] [32]. BBC [22], WAH [33], EWAH [35], COMPAX [36], PLWAH [38], CONCISE [39], SECOMPAX [40], PLWAH+ [41], MASC [42], SPLWAH [43], COMBAT [44], CAMP [45], BAH [46] e SBH [47], rappresentano l'attuale stato dell'arte delle tecniche di compressione bitmap.

In questa sezione vengono analizzati e descritti solo i metodi di compressione WAH, EWAH e COMPAX, analisi e confronti più dettagliati sui metodi

di compressione bitmap sono state fatte in [48] [49].

2.2.1 Word-Aligned Hybrid code

Word-Aligned Hybrid (WAH) [33] è una tecnica di compressione brevettata [34] che comprime sequenze di 31 bit uguali consecutivi. Con WAH una bitmap viene codificata come una sequenza di parole da 32 bit, dove ogni parola può essere una *literal-word* o una *fill-word*, in particolare:

- una *fill-word* rappresenta una sequenza di $31 \times C$ bit uguali consecutivi ed è codificata su una parola di 32 bit in cui il bit più significativo è sempre a 1 e:
  - se il secondo bit più significativo è uguale a 0, i rimanenti 30 bit rappresentano il numero di sequenze consecutive di 31 bit a 0.
  - se il secondo bit più significativo è uguale a 1, i rimanenti 30 bit rappresentano il numero di sequenze consecutive di 31 bit a 1.

  Di seguito viene riportato un esempio di *fill-word* che rappresenta una sequenza di 93 bit consecutivi a 0.

1	0	00000000000000000000000000011

- una *literal words* rappresenta 31 bit misti ed è codificata su una parola di 32 bit in cui il bit più significativo è sempre uguale a 0. Di seguito viene riportato un esempio di *literal-word* che rappresenza i seguenti 31 bit: 0101011100000011101001010101011

0	0101011100000011101001010101011

In conclusione utilizzando WAH: il numero massimo di bit uguali consecutivi che è possibile rappresentare con una *fill-word* è $31 \times (2^{31} - 1)$, al caso pessimo il numero totale di bit per codificare una bitmap (caso in cui viene codificata con solo *literal-word*) con N bit è $\frac{N}{32} + N$ e date due bitmap b_1, b_2 è possibile eseguire $b_1 \wedge b_2$ o $b_1 \vee b_2$ in un tempo proporzionale alla somma delle parole che compongono b_1 e b_2.

2.2.2 Enhanced Word-Aligned Compression code

Enhanced Word-Aligned Compression (EWAH) [35] è una tecnica di compressione che comprime sequenze di D bit uguali consecutivi, dove D può essere 16, 32 o 64. Con EWAH una bitmap viene codificata come una sequenza di parole da D bit, dove ogni parola può essere una *literal-word* o una *maker-word*, in particolare:

- una *maker-word* è codificata su una parola di D bit e:
  - se il bit più significativo è uguale a 0, i successivi $\frac{D}{2}$ bit rappresentano il numero di sequenze consecutive di D bit a 0.
  - se il bit più significativo è uguale a 1, i successivi $\frac{D}{2}$ bit rappresentano il numero di sequenze consecutive di D bit a 1.

  In entrambi i casi i rimanenti $\frac{D}{2} - 1$ bit rappresentano il numero di successive *literal-word*.
- una *literal-word* rappresenta D bit misti ed è codificata su una parola di D bit.

Ogni bitmap codificata con EWAH inizia sempre con una *maker-word*, di seguito viene riportato un esempio di codifica EWAH con $D = 32$ di una bitmap composta da 256 bit in cui i primi 192 bit sono uguali a 0 e rimanenti 64 sono 01010111000000111010010101010111 11000101010000101000100101100010.

0	0000000000000110	0000000000000010
01010111000000111010010101010111		
11000101010000101000100101100010		

In conclusione utiizzando EWAH con parole di D bit: il numero massimo di bit uguali consecutivi che è possibile rappresentare con una *maker-word* è $D \times (2^{D/2} - 1)$, al caso pessimo (caso in cui non ci sono sequenze consecutive di D bit uguali a 0 o 1) il numero totale di bit per codificare una bitmap con N bit è $D + N + \frac{N}{(2^{D/2})-1}$ e date due bitmap b_1, b_2 è possibile eseguire $b_1 \vee b_2$ in un tempo proporzionale alla somma delle parole che compongono b_1 e b_2 mentre è possibile eseguire $b_1 \wedge b_2$ in un tempo proporzionale alla somma delle *literal-word* allineate che compongono b_1 e b_2 poiché grazie alle *maker-word* le *literal-word* non allineate possono essere saltate.

2.2.3 COMressed Adaptive indeX code

COMpressed Adaptive indeX (COMPAX) [36] è una tecnica di compressione bitmap brevettata [37] disegnata appositamente per ridurre al minimo la dimensione di un indice bitmap su un attributo ad alta cardinalità (compe per esempio IP sorgente). Con COMPAX una bitmap viene codificata come una sequenza di parole da 32 bit, dove ogni parola può essere una *literal-word*, una *0-fill-word*, una *LFL-word* o una *FLF-word*, in particolare:

- una *literal-word* rappresenta 31 bit misti ed è codificata su una parola di 32 bit in cui il bit più significativo è sempre uguale a 1.
- una *0-fill-word* rappresenta una sequenza di $31 \times C$ bit uguali a 0 consecutivi ed è codificata su una parola di 32 bit in cui i primi 3 bit più significativi sono uguali a 000.
- una *LFL-word* rappresenta una sequenza di *[literal-word]-[0-fill-word]-[literal-word]* ed è codificata su una parola di 32 bit in cui i primi 3 bit più significativi sono uguali a 001. Per poter codificare una sequenza di bit con una *LFL-word* è necessario che all'inizio e alla fine della sequenza di bit ci sia solo 1 byte con bit diversi da 0 e la sequenza di bit a 0 non sia maggiore di 255×31 bit.
- una *FLF-word* rappresenza una sequenza di *[0-fill-word]-[literal-word]-[0-fill-word]* ed è codificata su una parola di 32 bit in cui i primi 3 bit più significativi sono uguali a 010. Per poter codificare una sequenza di bit con una *FLF-word* è necessario che all'inizio e alla fine della squenza di bit non ci sia più di 255×31 bit consecutivi uguali a zero e che l'unica *literal-word* contenga al più 1 byte con bit diversi da zero.

Secondo [36] [48], nonostante che con COMPAX non sia possibile comprimere le sequenze di bit consecutivi uguali a 1, grazie alla possibilità di codificare le sequenze di bit *[literal]-[0-fill]-[literal]* e *[0-fill]-[literal]-[0-fill]* con una sola parola, questa tecnica di compressione comprime gli indici bitmap ad alta cardinalità in modo molto più efficiente rispetto a WAH, EWAH.

In conclusione utilizzando COMPAX: il numero massimo di bit uguali a 0 consecutivi che è possibile rappresentare con una *0-fill-word* è $31 \times (2^{29} - 1)$, al caso pessimo il numero totale di bit per codificare una bitmap (caso in cui viene codificata con solo *literal-word*) con N bit è $\frac{N}{32} + N$ e date due bitmap b_1, b_2 è possibile eseguire $b_1 \wedge b_2$ o $b_1 \vee b_2$ in un tempo proporzionale alla somma delle parole che compongono b_1 e b_2.

Capitolo 3

Indicizzazione flussi

Lo scopo principale di questo tirocinio è stato quello di sviluppare un nuovo tipo di indice bitmap per indicizzare i flussi di rete, in particolare per creare indici bitmap su indirizzi IP e porte al fine di ridurre al minimo: lo spazio dell'indice, il tempo di creazione dell'indice e il tempo di esecuzione di una *equality query*. In questo lavoro sono state quindi confrontate le prestazioni di indici bitmap per IP e porta costruiti con i metodi di indicizzazione descritti nel capitolo precedente *Multi-Component equality encoding* e *K-of-N equality encoding* utilizzando:

1. Bitmap non compresse.
2. Bitmap compresse con *Word-Aligned Hybrid* WAH.
3. Bitmap compresse con *Enhanced Word-Aligned Compression* EWAH.
4. Bitmap compresse con una nuova codifica bitmap sviluppata in questo tirocinio e chiamata *Only-Zero Byte Compression v2* (OZBCv2).

Altre tecniche di compressione più recenti come ad esempio COMPAX non sono state prese in considerazione principalmente per due motivi: il primo è perché non ci sono implementazioni già pronte in rete e non era possibile per questioni di tempo a disposizione implementarle durante il tirocinio, il secondo è perchè la maggior parte di queste tecniche anche se secondo [48] [49] rispetto a WAH e EWAH risultano esssere più efficienti in termini di spazio occupato e tempo di resuluzione di query, sono brevettate e quindi non sono utilizzabili per la realizzazione di un collezionatore.

In questo capitolo viene descritto il lavoro svolto durante il tirocinio per costruire indici bitmap efficienti su indirizzi IP e porte (interi a 32 e 16 bit).

Poiché gli indici creati devono essere efficienti sia durante la fase di inserimento sia durante l'esecuzione di query, per ogni possibile combinazione di metodo di indicizzazione e metodo di compressione bitmap è stato valutato:

- il tempo di creazione dell'indice.
- lo spazio necessario per salvare l'indice su memoria persistente.
- il tempo di esecuzione di un'*equality-query*.

3.1 Indici con bitmap non compresse

In questa sezione vengono confrontati i metodi di indicizzazione *Multi-Component equality encoding* e *K-of-N equality encoding* utilizzando bitmap non compresse per costruire indici su interi a 32 e 16 bit. Tutti i test eseguiti in questa sezione sono stati fatti utilizzando una implementazione efficiente di bitmap non compressa creata durante questo tirocinio [50].

3.1.1 Multi-Component equality encoding

Come descritto nel capitolo precedente dato un attributo con un numero di possibili valori distinti rappresentabile da B bit (e quindi con cardinalità uguale a $L = 2^B$) utilizzando *Multi-Component equality encoding* possiamo decomporre l'attributo in K componenti $K_1, K_2, ..., K_k$ grandi rispettivamente $B_1, B_2, ..., B_K$ bit (con $B_1 + B_2 + ... + B_K = B$) e quindi $2^{B_1}, 2^{B_2}, .., 2^{B_K}$ bitmap. Di seguito vengono riportare le considerazioni effettuate per stabilire il numero di componenti ideale per creare un indice su attributi rappresentabili con 8 bit (e quindi cardinalità uguale a 256). Una volta stabilito il K ideale per indicizzare valori da 1 byte, basta utilizzare $2 \times K$ e $4 \times K$ componenti per indicizzare rispettivamente attributi rappresentabili con 2 o 4 byte. Per semplicità sono state prese in considerazione solo le soluzioni con lo stesso numero di bitmap per ogni componente.

K = 1: in questo caso l'indice è formato da una sola componente composta da 256 bitmap e quindi:

- per risolvere una query è necessario estrarre una sola bitmap.
- per indicizzare N valori occorrono quindi $256 \times N$ bit e il rapporto tra la dimensione dell'indice e la dimensione del dato è uguale a $\frac{256 \times N}{8 \times N} = \frac{256}{8} = 32$.

Poiché non è accettabile avere un indice che occupa 32 volte la dimensione del dato questa soluzione non è stata presa in considerazione nelle valutazioni sul tempo di creazione dell'indice e sul tempo di esecuzione di query.

K = 2: in questo caso l'indice è formato da due componenti composte da $2^4 = 16$ bitmap ciascuna e quindi:

- per risolvere una query è necessario estrarre e mettere in and 2 bitmap.
- per indicizzare N valori occorrono quindi $16 \times 2 \times N = 32 \times N$ bit e il rapporto tra la dimensione dell'indice e la dimensione del dato è uguale a $\frac{32 \times N}{8 \times N} = \frac{32}{8} = 4$.

Lo svantaggio di questa soluzione è che l'indice occupa 4 volte la dimensione del dato.

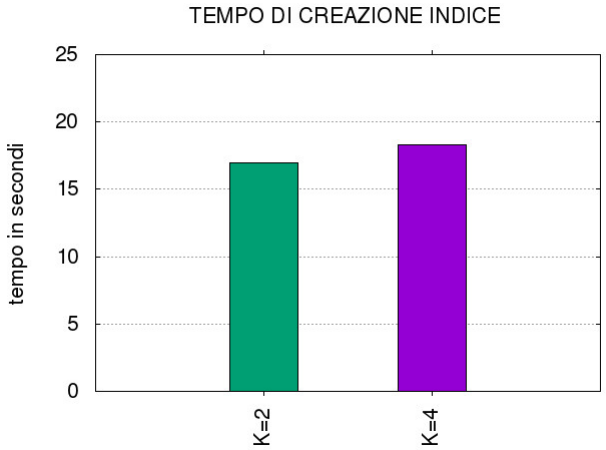
K = 4: in questo caso l'indice è formato da quattro componenti composte da $2^2 = 4$ bitmap ciascuna e quindi:

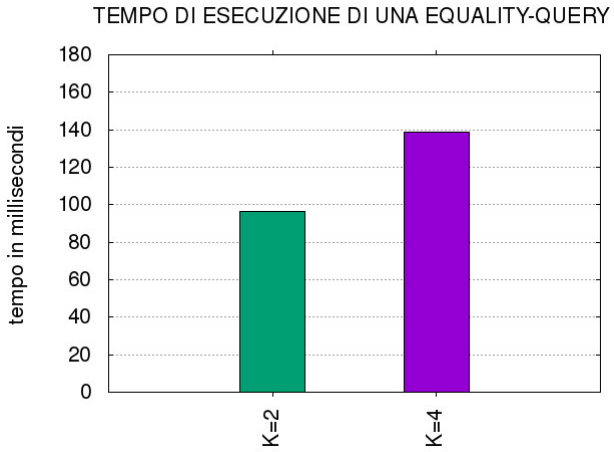
- per risolvere una query è necessario estrarre e mettere in and 4 bitmap.
- per indicizzare N valori occorrono quindi $4 \times 4 \times N = 16 \times N$ bit e il rapporto tra la dimensione dell'indice e la dimensione del dato è uguale a $\frac{16 \times N}{8 \times N} = \frac{16}{8} = 2$.

Anche se l'indice occupa 2 volte la dimensione del dato, lo svantaggio di questa soluzione è che per eseguire una query devono essere estratte 4 bitmap (che occupano la metà del dato) e messe in and tra loro.

Risultati dei test su Multi-Component con bitmap non compresse

Per confrontare il tempo di creazione dell'indice e il tempo di esecuzione di query sono stati creati indici bitmap utilizzando *Multi-Component equality encoding* con $K = 2$ e $K = 4$ su $N = 1$ Giga valori random compresi tra 0 e 255. Di seguito vengono riportati i risultati dei test eseguiti su una macchina con processore Intel Core i7-6700 - 3,4 GHz - 8 MB cache.





Dai risultati dei test è possibile vedere che mentre il tempo di creazione rimane quasi invariato (18.3 secondi per $K = 4$ e 16.9 secondi per $K = 2$), il tempo di esecuzione di un'*equality-query* è nettamente inferiore per $K = 2$. In conclusione, per gli scopi di questo tirocinio, anche se utilizzando un indice creato con $K = 4$ si ottiene un vantaggio in termini di spazio (l'indice occupa

la metà rispetto a $K = 2$) poiché utilizzando un indice creato con $K = 2$ si ottiene un vantaggio significativo sia in fase di esecuzione di una query (con un tempo di circa $\frac{2}{3}$ del tempo di esecuzione di una query con $K = 4$) e sia in fase estrazione delle bitmap da mettere in and, la scelta di utilizzare $K = 2$ risulta la più appropriata.

3.1.2 K-of-N equality encoding

Come descritto nel capitolo precedente con il metodo di indicizzazione *K-of-N equality encoding* scelti due numeri N e K è possibile costruire un indice utilizzando N bitmap che indicizza $L = \binom{N}{K}$ possibili valori distinti creando una corrispondenza biunivoca tra ogni possibile sequenza di N bit con K bit settati a 1 e ogni possibile valore. Per prima cosa quindi è stato calcolato il numero di bitmap necessarie N per costruire indici su interi a 16 e 32 bit. Dato che K equivale anche al numero di bitmap che devono essere estratte e messe in and per processare una query, sono stati presi in considerazione solo valori di $K <= 8$ per indici su interi a 16 bit e $K <= 16$ per indici su interi a 32 bit poiché altrimenti l'indice risulterebbe inefficiente. Per questi motivi e per limitare il numero N di bitmap che compongono un indice sono state prese in considerazioni solo le seguenti opzioni:

per indici su interi a 16 bit in questo caso $\binom{N}{K}$ deve essere maggiore di 2^{16}

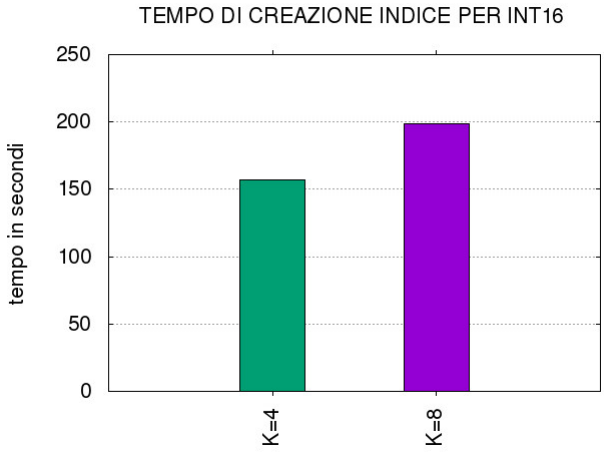
- $N = 19, K = 8$, per risolvere una query è necessario estrarre e mettere in and 8 bitmap e il rapporto tra la dimensione dell'indice e la dimensione del dato è uguale a $19 \div 16 = 1.18$.
- $N = 37, K = 4$, per risolvere una query è necessario estrarre e mettere in and 4 bitmap e il rapporto tra la dimensione dell'indice e la dimensione del dato è uguale a $37 \div 16 = 2.31$.

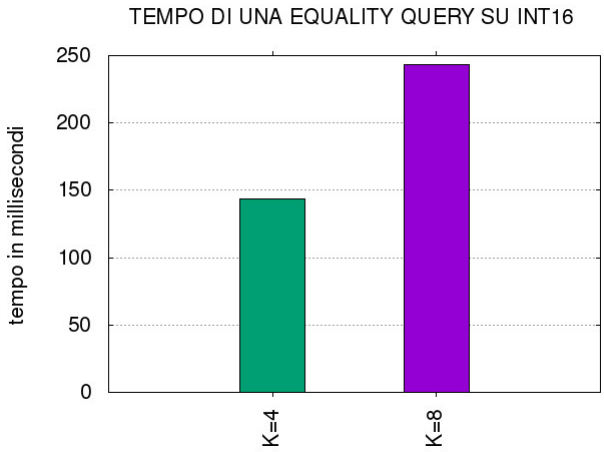
per indici su interi a 32 bit in questo caso $\binom{N}{K}$ deve essere maggiore di 2^{32}

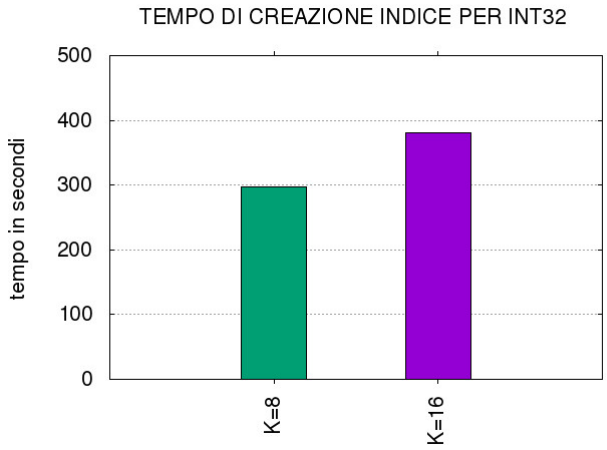
- $N = 36, K = 16$, per risolvere una query è necessario estrarre e mettere in and 16 bitmap e il rapporto tra la dimensione dell'indice e la dimensione del dato è uguale a $36 \div 32 = 1.13$.
- $N = 64, K = 4$, per risolvere una query è necessario estrarre e mettere in and 4 bitmap e il rapporto tra la dimensione dell'indice e la dimensione del dato è uguale a $64 \div 32 = 2$.

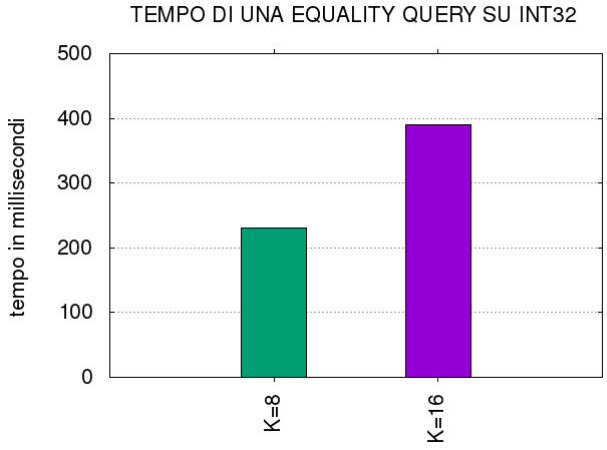
Risultati dei test su K-of-N con bitmap non compresse

Per confrontare il tempo di creazione dell'indice e il tempo di esecuzione di query sono stati quindi creati indici bitmap su 1 Giga valori a 16 bit utilizzando $(N = 19, K = 8), (N = 37, K = 4)$ e su su 1 Giga valori a 32 bit utilizzando $(N = 36, K = 16), (N = 64, K = 8)$. Per stabilire una corrispondenza biunivoca tra ogni possibile valore inserito e ogni possibile sottoinsieme è stato implementato in codice C++ l'algoritmo di tipo greedy descritto in [51]. Di seguito vengono riportati i risultati dei test eseguiti su una macchina con processore Intel Core i7-6700 - 3,4 GHz - 8 MB cache.









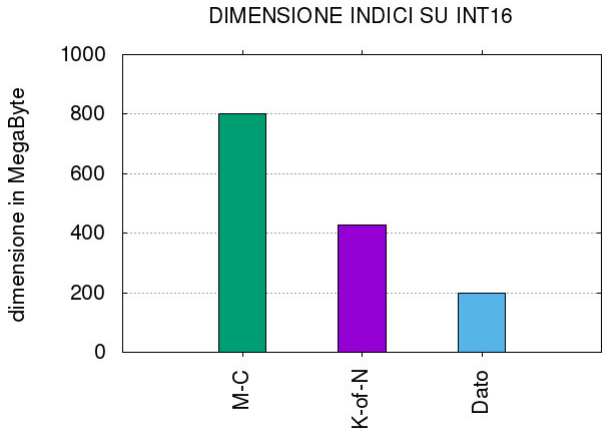
Dai risultati dei test è possibile vedere che utilizzando un valore maggiore di K risulta inferiore in modo significativo sia il tempo di creazione dell'indice e sia il tempo di esecuzione di una query e che lo spazio occupato da un indice non è mai superiore a circa il doppio della dimensione del dato (dimensione considerata accettabile). In conclusione quindi, per lo scopo di questo tirocinio, utilizzando il metodo di indicizzazione *K-of-N equality encoding* e bitmap non compresse la migliore scelta risulta essere con $K = 8$ per indici su interi a 32 bit e con $K = 4$ per indici su interi a 16 bit.

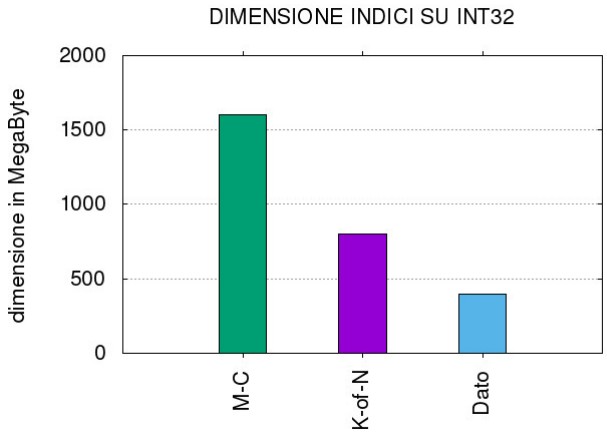
3.1.3 Test e considerazioni finali

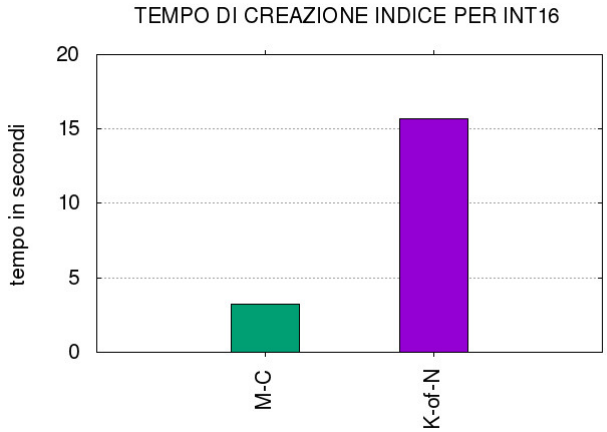
Dalle considerazioni che sono state fatte, per lo scopo di questo tirocinio:

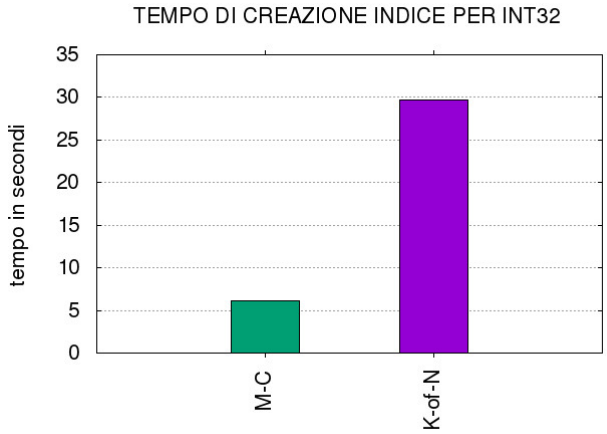
- per il metodo di indicizzazione *Multi-Component equality encoding* la scelta migliore risulta essere $K = 2$ per indicizzare 2^8 possibili valori e quindi $K = 4, K = 8$ per indicizzarne $2^{16}, 2^{32}$.
- per il metodo di indicizzazione *K-of-N equality encoding* la scelta migliore risulta essere $K = 4$ per indicizzare 2^{16} possibili valori e $K = 8$ per indicizzarne 2^{32}.

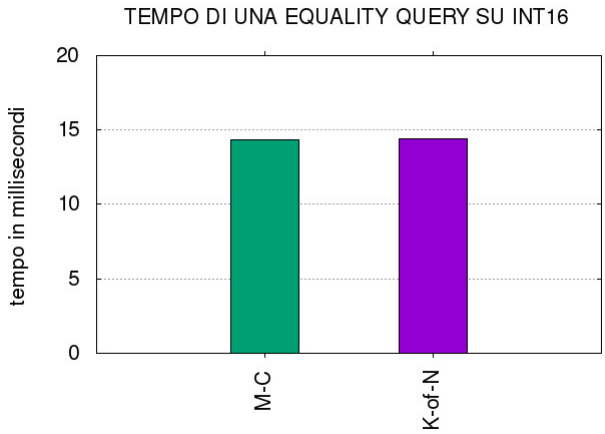
Per confrontare la dimensione, il tempo di creazione e il tempo di esecuzione di una *equality-query* di indici creati nel modo descritto ($K = 4$ per interi a 16 bit e $K = 8$ per interi a 32 bit) sono stati creati indici bitmap su 100 Mega valori. Di seguito vengono riportati i risultati dei test eseguiti su una macchina con processore Intel Core i7-6700 - 3,4 GHz - 8 MB cache.

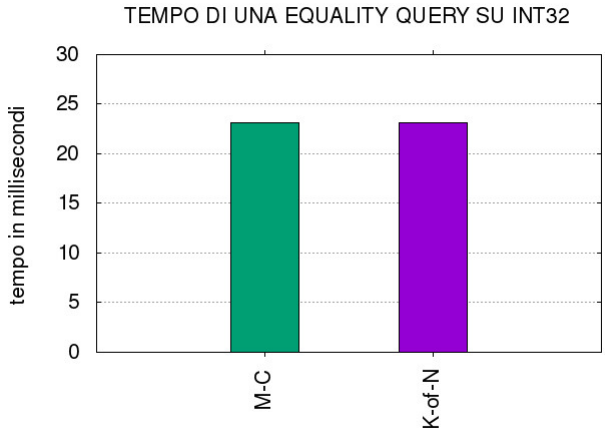



DIMENSIONE INDICI SU INT32
dimensione in MegaByte
2000
1500
1000
500
0
M-C
K-of-N
Dato


TEMPO DI CREAZIONE INDICE PER INT16
tempo in secondi
20
15
10
5
0
M-C
K-of-N


TEMPO DI CREAZIONE INDICE PER INT32
tempo in secondi
35
30
25
20
15
10
5
0
M-C
K-of-N





Dal confronto tra i due metodi è possibile è possibile determinare svantaggi e vantaggi dei due metodi, in particolare:

- gli indici creati con *K-of-N equality encoding* occupano circa la metà di quelli creati con *Multi-Component equality encoding*.
- il tempo di creazione degli indici con *K-of-N equality encoding* è uguale a circa 5 volte il tempo di creazione degli indici con *Multi-Component equality encoding*.
- a parità di K il tempo di query è lo stesso.

Con queste considerazione e osservando che la velocità di inserimento dei valori nell'indice sulla macchina su cui sono stati eseguiti i test con *K-of-N*

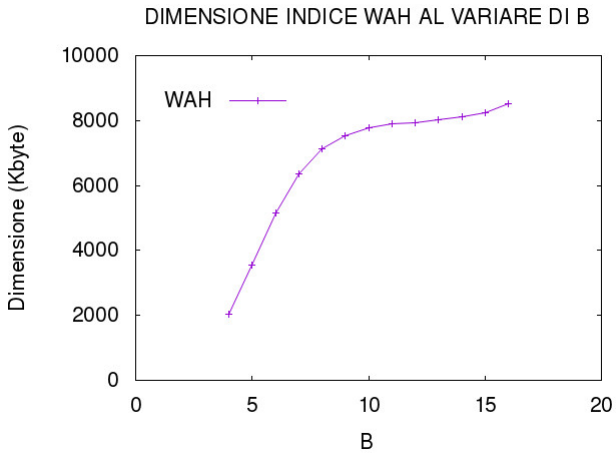
equality encoding è di $\frac{100Mvalori}{15.7s} = 6.3Mvalori/s$ per indici su interi a 16 bit e di $\frac{100Mvalori}{29.7s} = 3.4Mvalori/s$ per indici su interi a 32 bit, per lo scopo di questo tirocinio il metodo *K-of-N equality encoding* risulta essere migliore poiché permette di costruire indici con una dimensione limitate e con un tempo di inserimento accettabile.

3.2 Indici con bitmap WAH

In questa sezione vengono confrontati i metodi di indicizzazione *Multi-Component equality encoding* e *K-of-N equality encoding* utilizzando bitmap compresse con la codifica *Word-Aligned Hybrid* (WAH) per costruire indici su interi a 32 e 16 bit. Tutti i test eseguiti in questa sezione sono stati fatti utilizzando una implementazione in C++ efficiente di bitmap WAH disponibile su github [52].

3.2.1 Risultati sulla dimensione dell'indice

Come descritto nel capitolo precedente con la tecnica di codifica WAH la dimensione di una bitmap dipende esclusivamente dal numero di sequenze di 31 bit consecutivi a 0 o 1. Per questo motivo prima di iniziare a valutare le possibili opzioni su come scegliere K il numero di bit settati a 1 per ogni valore indicizzato e N il numero di bitmap che compongono l'indice, è stata analizzata la dimensione di un indice creato con $L = 2^B$ bitmap WAH su un attributo con L possibili valori distinti al variare di B. Di seguito viene riportato il risultato del test effettuato, nel test vengono indicizzati 1 Mega valori random compresi tra 0 e $L-1$ per $B = \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$ utilizzando $L = 2^B$ bitmap.



Dal grafico è possibile vedere che per $B >= 8$ la dimensione dell'indice rimane quasi invariata. Questo implica che un indice con $2^{16} = 65536$ bitmap occupa la stessa dimensione (poco di più) di un indice costruito con $2^8 = 256$ bitmap e quindi che per indicizzare 65536 possibili valori distinti serve lo stesso spazio utilizzato per indicizzarne 256. Per indicizzare interi a 16 e 32 bit valgono le seguenti considerazioni:

- con il metodo *Multi Component equality encoding* sia decomponendo 16 bit in 2 componenti da 2^8 bitmap ($K = 2$) e sia decomponendo 16 bit in 4 componenti da 2^4 bitmap ($K = 4$) la dimensione dell'indice sarebbe maggiore o uguale a un indice creato con 2^{16} bitmap. Per questo motivo quindi per un indice su un attributo a 16 bit conviene utilizzare una sola componente ($K = 1$) e per un indice su un attributo a 32 bit conviene utilizzare 2 componenti ($K = 2$) da 2^{16} bitmap ciascuna.

- con il metodo *K-of-N equality encoding* le soluzioni valutate sono $N = 363, K = 2$ per indicizzare interi a 16 bit e $N = 569, K = 4$ per indicizzare interi a 32 bit.

3.2.2 Considerazioni finali

In conclusione con WAH la dimensione dell'indice dipende sia dal numero di bitmap e sia dalla densità di bit settati a 1 in ogni bitmap. In questo scenario utilizzando il metodo *K-of-N equality encoding* la dimensione dell'indice è praticamente uguale a quella del corrispondente indice costruito con il metodo *Multi-Component equality encoding* utilizzando gli stessi valori di K poiché il numero di bitmap viene dimezzato ma la densità di bit settati a 1 in ogni bitmap raddoppia, ad esempio: per costruire un indice su un intero

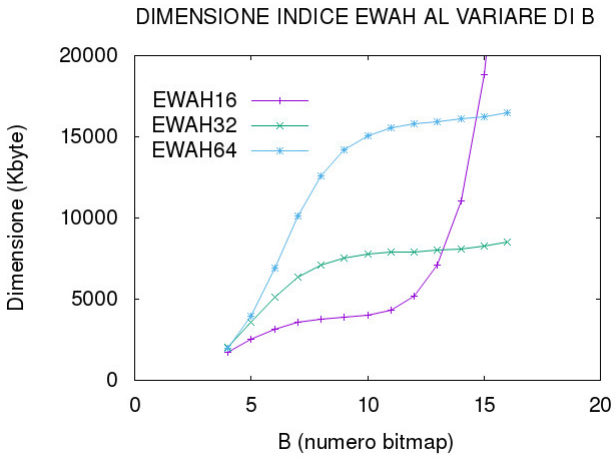
a 16 bit, se $K = 2$ con *K-of-N equality encoding* servono 363 bitmap con densità $\frac{2}{363} = 0.0055$ ciascuna mentre con *Multi-Component equality encoding* ne servono 512 bitmap con densità $\frac{1}{256} = 0.0039$. Per questo motivo il metodo più efficiente per costruire un indice utilizzando WAH è *Multi-Component equality encoding* con una sola componente da 65536 bitmap per indicizzare 2^{16} valori distinti e 2 componenti da 65536 bitamp ciascuna per indicizzare 2^{32} valori distinti e lo spazio occupato dall'indice su valori random è quindi circa 4 volte la dimensione del dato.

3.3 Indici con bitmap EWAH

In questa sezione vengono confrontati i metodi di indicizzazione *Multi-Component equality encoding* e K-of-N equality encoding utilizzando bitmap compresse con la codifica *Enhanced Word-Aligned Compressione Hybrid* (EWAH) per costruire indici su interi a 32 e 16 bit. Tutti i test eseguiti in questa sezione sono stati fatti utilizzando una implementazione in C++ efficiente di EWAH disponibile su github [53].

3.3.1 Analisi sulla dimensione dell'indice

Come descritto nel capitolo precedente con la tecnica di codifica (EWAH) la dimensione di una bitmap dipende esclusivamente dal numero di sequenze di D bit consecutivi a 0 o 1, dove D può essere 16, 32, 64. Per questo motivo, come per WAH prima di iniziare a valutare le possibili opzioni su come scegliere K il numero di bit settati a 1 per ogni valore indicizzato e N il numero di bitmap che compongono l'indice, è stata analizzata la dimensione di un indice creato con $L = 2^B$ bitmap EWAH (per $D = \{16, 32, 64\}$) su un attributo con L possibili valori distinti al variare di B. Di seguito viene riportato il risultato del test effettuato, nel test vengono indicizzati 1 Mega valori random compresi tra 0 e $L - 1$ per $B = \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$ utilizzando $L = 2^B$ bitmap.



Tenendo conto delle considerazioni fatte nella sezione precedente su WAH, dal grafico è possibile vedere che:

- utilizzare EWAH64 non conviene in nessun caso.
- per EWAH32 valgolo le stesse identiche considerazioni fatte per WAH.
- per $6 < B < 12$ l'indice costruito con EWAH16 occupa la metà di spazio rispetto all'indice costruito con EWAH32.
- per $B > 12$ la dimensione dell'indice costruito con EWAH16 cresce in modo esponenziale.

Questo implica che diversamente da EWA32 e WAH, con EWAH16 per creare indici su interi a 16 e 32 bit conviene utilizzare componenti composte da $2^8 = 256$ bitmap poiché quando il numero delle bitmap supera $2^{12} = 4096$ la dimensione dell'indice cresce in modo esponenziale.

3.3.2 Considerazioni finali

In conclusione dato che come per WAH anche per EWAH la dimensione dell'indice dipende sia dal numero di bitmap sia dalla densità di bit settati a 1 in ogni bitmap anche in questo caso utilizzare il metodo *K-of-N equality encoding* non diminuisce lo spazio dell'indice. Per questo motivo il metodo più efficiente per costruire un indice utilizzando EWAH è *Multi-Component equality encoding* con una componenti da 65536 bitmap per EWAH32 (come per WAH) e con componenti da 256 bitmap per EWAH16. Come per WAH, con EWAH32 e EWAH16 lo spazio occupato dall'indice su valori random è circa 4 volte la dimensione del dato.

3.4 Indici con bitmap OZBCv2

In questa sezione viene presentata e analizzata *Only-Zero Byte Compression v2* [54], una nuova tecnica di compressione bitmap sviluppata e implementata durante il tirocinio. OZBCv2 è stata implementata in C++ e rilasciata con licensa Gnu Lesser General Public Licensev3.0 [55] su github [54].

3.4.1 Only-Zero Byte Compression v2 code

Only-Zero Byte Compression v2 è una nuova tecnica di compressione di tipo Word-Aligned (come WAH, EWAH e COMPAX) disegnata appositamente per ridurre al minimo la dimensione di un indice bitmap su un attributo ad alta cardinalità costruito utilizzando il metodo *Multi-Component equality encoding*. La scelta di creare una nuova codifica ottizzata per il metodo *Multi-Component equality encoding* è dovuta al fatto che questo metodo risulta più veloce in fase di inserimento e che secondo i risultati delle analisi fatte sulla dimensione dell'indici creati con WAH e EWAH, poiché con le tecniche di compressione di tipo Word-Aligned la dimensione dell'indice dipende sia dal numero di bitmap N e sia dalla densità dei bit settati a 1 in ogni bitmap D, utilizzare *K-of-N equality encoding* non diminuisce la dimensione dell'indice (poiche diminuisce N ma aumenta D). Inoltre sono state fatte le seguenti osservazioni:

- in un indice bitmap costruito con il metodo di indicizzazione *Multi-Component equality encoding* la densità D di bit settati a 1 in ogni bitmap è nel caso di distribuzione uniforme (numeri random) uguale a 1 diviso il numero di bitmap di una componente.
- indipendentemente dalla distribuzione dei valori da indicizzare con *Multi-Component equality encoding* per ogni valore indicizzato c'è un solo bit settato a 1 per ogni componente che compone l'indice.
- con *Multi-Component equality encoding* nel caso di distribuzione uniforme dei valori da indicizzare, ogni bitmap di ogni componente è composta da sequenze di bit a 0 seguite da un solo bit settato a 1.

In questo scenario quindi, riuscire a comprimere in modo efficiente sequenze di bit a 0 è molto più importante di riuscire a comprimere sequenze di bit a 1. Per questo motivo *Only-Zero Byte Compression v2* è una tecnica di compressione bitmap che comprime solo le sequenze di bit a 0, di seguito viene descritta la codifica in dettaglio.

Codifica OZBCv2

Con OZBCv2 una bitmap viene codificata come una sequenza di parole da 16 bit, dove ogni parola può essere una *bytes_zero_word*, una *2bytes_zero_word*, una *2bytes_dirty_word* o una *dirty_word*, in particolare:

- una *bytes_zero_word* rappresenta una sequenza di $8 \times C$ bit uguali a 0 consecutivi seguiti da 1 byte misto (in cui ci possono essere 0 e 1) ed è codificata su una parola di 16 bit in cui il bit più significativo è uguale a 0, i 7 bit successivi rappresentano il numero di byte consecutivi a 0 e i rimanenti 8 bit sono dirty. Il numero massimo di 0 consecutivi rappresentabili con questa parola è uguale a $2^3 \times 2^7 = 2^{10} = 1024$ bit.

- una *2bytes_zero_word* rappresenta una sequenza di $16 \times C$ bit uguali a 0 consecutivi ed è codificata su una parola di 16 bit in cui i primi due bit più significativi sono uguali a 01 e i rimanenti 14 bit rappresentano il numero di 2byte consecutivi a 0. Il numero massimo di 0 consecutivi rappresentabili con questa parola è uguale a $2^4 \times 2^{14} = 2^{18} = 262144$ bit.

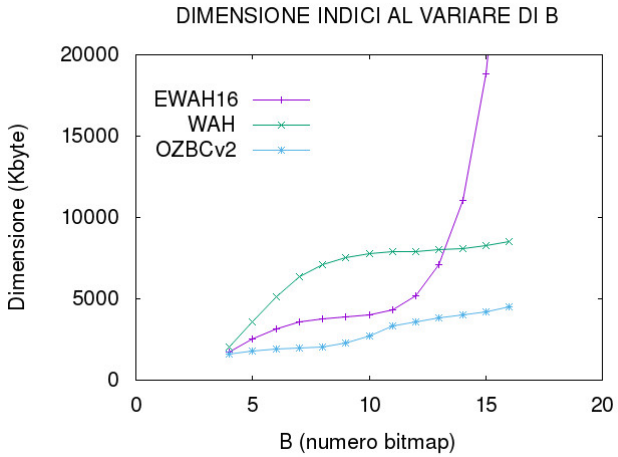
- una *2bytes_dirty_word* rappresenta una sequenza di $16 \times C$ bit misti consecutivi ed è codificata su una parola di 16 bit in cui i primi due bit più significativi sono uguali a 11 e i rimanenti 14 bit rappresentano il numero di 2byte consecutivi misti (*dirty_word*). Il numero massimo di bit misti consecutivi rappresentabili con questa parola è uguale a $2^4 \times 2^{14} = 2^{18} = 262144$.

- una *dirty_word* rappresenta 16 bit misti ed è codificata su una parola di 16 bit. In una bitmap OZBCv2 una sequenza di *dirty_word* è sempre preceduta da una *2bytes_dirty_word*.

In conclusione utilizzando OZBCv2: è possibile comprimere in maniere efficiente sequenze di bit consecutivi a 0 seguite da un byte misto utilizzando una parola (2 byte), il numero massimo di 0 consecutivi che è possibile rappresentare con una parola è 2^{18} e date due bitmap b_1, b_2 è possibile eseguire $b_1 \vee b_2$ in un tempo proporzionale alla somma delle parole che compongono b_1, b_2 mentre è possibile eseguire $b_1 \wedge b_2$ in un tempo proporzionale alla somma delle che contengono bit misti allineate poichè grazie alle *2bytes_dirty_word* le *dirty_word* non allineate possono essere saltate.

3.4.2 Analisi sulla dimensione dell'indice

Come per EWAH e WAH anche per OZBCv2 è stata analizzata la dimensione di un indice creato con $L = 2^B$ bitmap OZBCv2 su un attributo con L possibili valori distinti al variare di B. Di seguito viene riportato il risultato del test effettuato insieme ai risultati ottenuti su WAH e EWAH16, nel test vengono indicizzati 1 Mega valori random compresi tra 0 e $L - 1$ per $B = \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$ utilizzando $L = 2^B$ bitmap.



Dal grafico è possibile vedere che con OZBCv2 la dimensione dell'indice è sempre minore di EWAH16 e WAH. Inoltre per creare indici con il metodo *Multi-Component equality encoding* su interi a 16 e 32 bit conviene utilizzare componenti composte da $2^8 = 256$ bitmap poiché la dimensione di una sola componente da 2^{16} bitmap è maggiore della dimensione di due componenti composte da 2^8 bitmap ciascuna.

3.4.3 Considerazioni finali

In conclusione dopo aver osservato alcune proprietà degli indici bitmap creati con il metodo *Multi-Component equality encoding*, è stato possibile realizzare una codifica bitmap (OZBCv2) che comprimendo nel migliore dei modi le sequenze di bit a 0 seguite da un byte misto, permette di creare indici con una dimensione inferiore rispetto alle codifiche EWAH e WAH con uno spazio occupato dall'indice su valori random è di circa 2 volte la dimensione del dato.

3.5 Validazione

In questa sezione vengono riportati i test effettuati su indici bitmap a 16 e 32 bit costruiti sui campi IP_SORGENTE, IP_DESTINAZIONE, PORTA_SORGENTE e PORTA_DESTINAZIONE da un insieme di 5312352 (5 Mega) flussi di rete generati da un archivio reale di pacchetti di dimensione 9 GigaByte fornito da CAIDA [56]. Nei test viene confrontata la dimensione, il tempo di creazione e il tempo per risoluzione di un'*equality-query* per indici costruiti con bitmap non compresse, bitmap WAH, bitmap EWAH e bitmap OZBCv2. Per ogni indice, il metodo di indicizzazione è stato scelto sulla base delle considerazioni riportate nelle sezioni precedenti, in particolare:

indici con bitmap non compresse: in questo caso l'indice viene creato utilizzando il metodo di indicizzazione *K-of-N equality encoding* con $(N = 64, K = 8)$ per indici su IP_SORGENTE e IP_DESTINAZIONE e con $(N = 37, K = 4)$ per indici su PORTA_SORGENTE e PORTA_DESTINAZIONE. Il numero delle bitmap che compongono un indice è N.

indici con bitmap WAH: in questo caso l'indice viene creato utilizzando il metodi di indicizzazione *Multi-Component equality encoding* con $K = 2$ (numero di componenti) per indici su IP_SORGENTE e IP_DESTINAZIONE e con $K = 1$ per indici su PORTA_SORGENTE e PORTA_DESTINAZIONE. Poiché ogni componente è composta da $2^{16} = 65536$ bitmap, il numero delle bitmap che compongono un indice è $K \times 65536$.

indici con bitmap EWAH16: in questo caso l'indice viene creato utilizzando il metodo di indicizzazione *Multi-Component equality encoding* con $K = 4$ (numero di componenti) per indici su su IP_SORGENTE e IP_DESTINAZIONE e con $K = 2$ per indici su PORTA_SORGENTE e PORTA_DESTINAZIONE. Poiché ogni componente è composta da 256 bitmap, il numero delle bitmap che compongono un indice è $K \times 256$.

indici con bitmap OZCBv2: in questo caso l'indice viene creato utilizzando il metodo di indicizzazione *Multi-Component equality encoding* con $K = 4$ (numero di componenti) per indici su su IP_SORGENTE e IP_DESTINAZIONE e con $K = 2$ per indici su PORTA_SORGENTE e PORTA_DESTINAZIONE. Poiché ogni componente è composta da 256 bitmap, il numero delle bitmap che compongono un indice è $K \times 256$.

Per confrontare il tempo di risoluzione di query sono state definite 3 query ($x1, x2, y1, y2$ sono valori di un record qualsiasi dal file dei flussi):

query_1: SELECT * FROM flussi WHERE IP_DESTINAZIONE=$x2$.

query_2: SELECT * FROM flussi WHERE IP_SORGENTE=$x1$ AND PORTA_SORGENTE=$y1$.

query_2: SELECT * FROM flussi WHERE IP_SORGENTE=$x1$ AND IP_DESTINAZIONE=$x2$ AND PORTA_SORGENTE=y1 AND PORTA_DESTINAZIONE=y2.

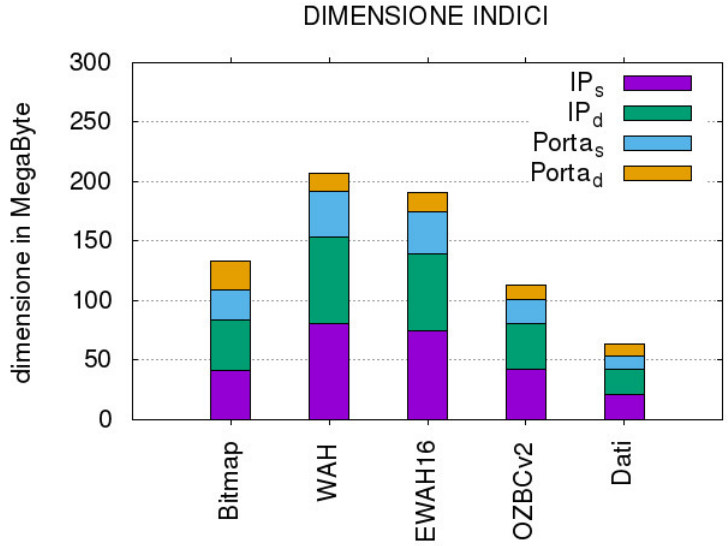
Tutti i test sono stati eseguiti su una macchina con processore Intel Core i7-6700 - 3,4 GHz - 8 MB cache con i flussi già caricati in memoria principale, in questo modo:

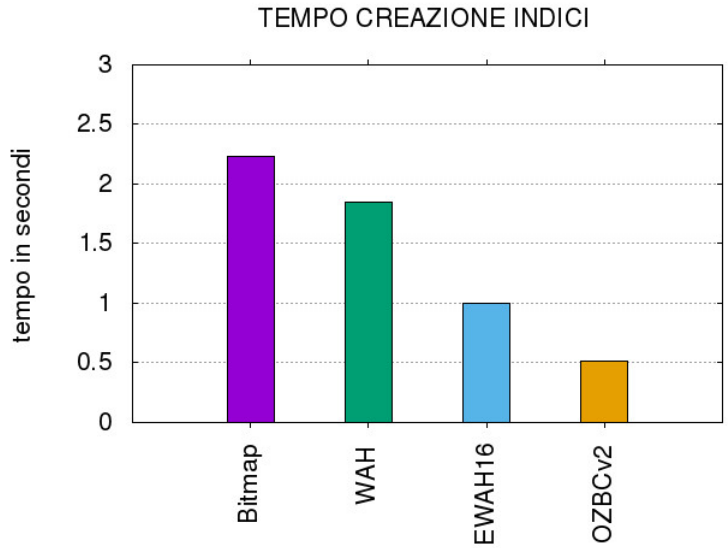
- Nel tempo di creazione degli indici non è compreso il tempo necessario a leggere i flussi da memoria persistente.
- il tempo di esecuzione di una query è dato dal tempo impiegato per trovare le posizioni dei record che soddisfano la clausola "WHERE" della query avendo gli indici già in memoria principale.

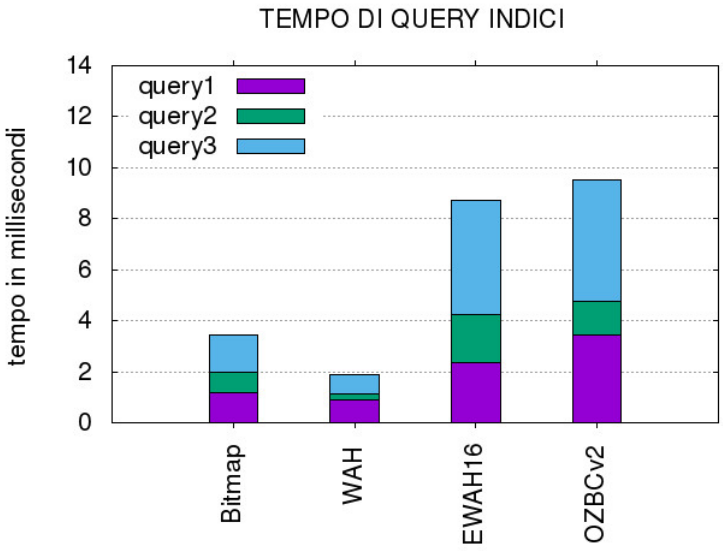
Infine poiché la dimensioni degli indici creati con le bitmap compresse dipende anche dall'ordine e dal valore dei record indicizzati, tutti i test sono stati rieseguiti dopo aver ordinato i flussi. Al fine di valutare una soluzione realmente utilizzabile per un collezionatore, i flussi sono stati ordinati con il *qsort* su IP_SORGENTE a blocchi di 100 K records in modo tale da rendere possibile ordinare e indicizzare i flussi in pipeline e da non compromettere in modo significativo il tempo di creazione degli indici. In questo caso quindi il tempo di creazione dell'indice è il massimo tra il tempo di ordinamento a blocchi di 100 K e il tempo impiegato per costruire l'indice.

3.5.1 Risultati su flussi non ordinati

Di seguito sono riportati i grafici che rappresentano i risultati dei test effettuati sui flussi non ordinati.





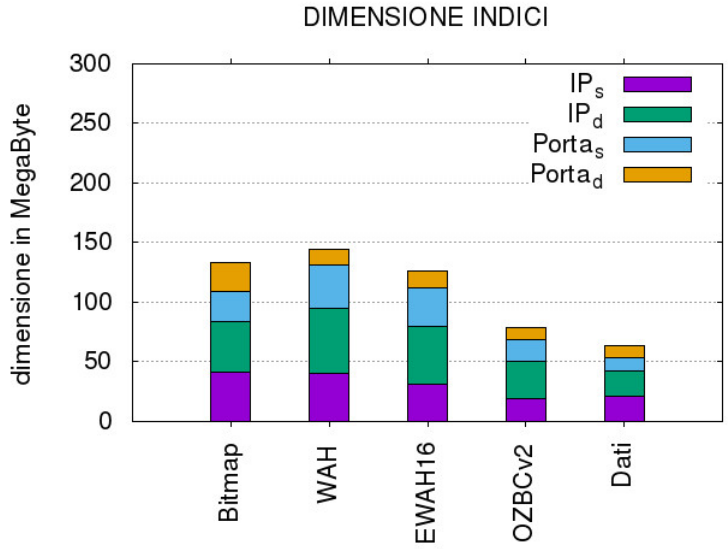


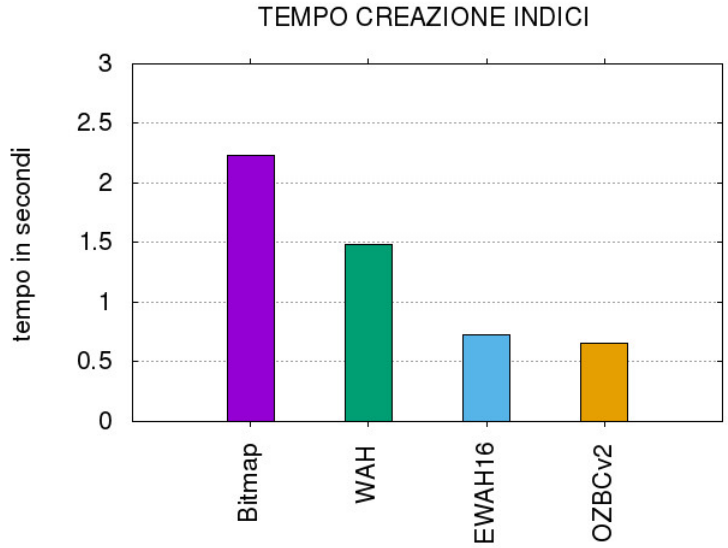
Dai risultati dei test effettuati è è possibile determinare che:

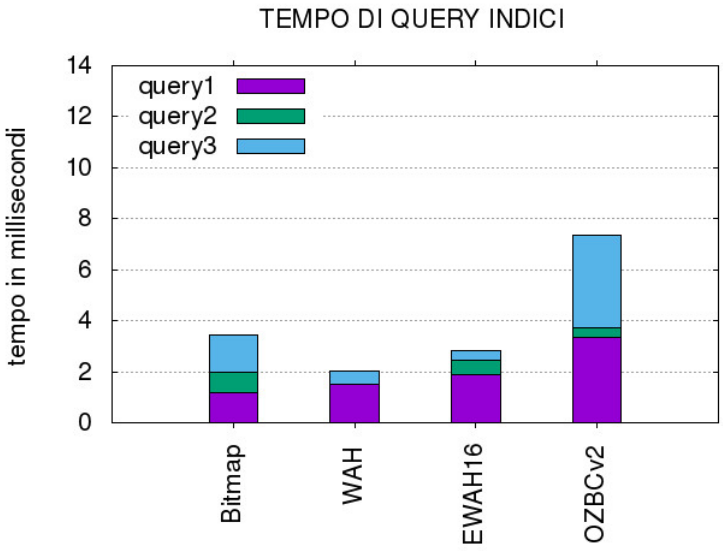
- su flussi non ordinati la dimensione degli indici è praticamente identica a quella analizzata su numeri random (4 volte la dimensione del dato per WAH e EWAH16 e 2 volte la dimensione del dato per Bitmap non compresse e OZBCv2).
- il tempo di creazione è nettamente inferiore in OZBCv2.
- il tempo di esecuzione delle 3 query è nettamente inferiore per WAH e bitmap non compresse.

3.5.2 Risultati su flussi ordinati a blocchi

Di seguito sono riportati i grafici che rappresentano i risultati dei test effettuati su flussi ordinati su IP_Sorgente a blocchi di 100 K record. Per il tempo di creazione dell'indice, viene considerato il massimo tra il tempo di ordinamento (783 millisecondi) e il tempo impiegato a settare l'indici poiché è possibile ordinare e settare gli indici in pipeline.







Dai risultati dei test effettuati è è possibile determinare che dopo aver ordinato a blocchi i flussi su IP_SORGENTE:

- la dimensione degli indici costruiti con WAH, EWAH16 e OZBCv2 si è quasi dimezzata rispetto a prima.
- il tempo di creazione è diminuito negli indici costruiti con WAH e EWAH16 mentre è leggermente aumentato in OZBCv2 poiché in questo caso il tempo di sort è leggermente superiore al tempo impiegato per settare l'indici.
- il tempo di esecuzione delle 3 query è doppio in OZBCv2 rispetto a WAH, EWAH16 e bitmap non compresse. le codifiche di compressione tranne che per EWAH16.
- il tempo di esecuzione delle 3 query è diminuito rispetto a prima ed è sempre simile per tutte le codifiche bitmap tranne che per EWAH16.

3.5.3 Considerazioni finali

Dopo aver analizzato i risultati dei test effettuati è possibile dire che ordinando i flussi su IP_SORGENTE si ottengono notevoli miglioramenti sotto tutti gli aspetti di vista (soprattutto per la dimensione degli indici) poiché l'ordinamento a blocchi di 100 K record impiega pochissimo tempo e può essere fatto in parallelo al settaggio degli indici. Inoltre per quanto riguarda

il tempo di query anche se dai test il tempo tempo minore è dato dagli indici con bitmap non compresse e WAH, c'è da considerare che in una reale implementazione si deve aggiungere a questo tempo il tempo di estrazione delle bitmap, che nel caso di indici costruiti con WAH, EWAH16 e OZBCv2 è nettamente inferiore poiché il numero di bitmap che devono essere estratte è minore e ognuna occupa molto meno spazio rispetto a una normale bitmap (proprio perchè è compressa). In conclusione quindi grazie al metodo *Multi-Component equality encoding* con componenti composte da 256 bitmap, all'ordinamento e alla nuova codifica bitmap sviluppata (OZBCv2) è possibile costruire indici con una dimensione poco superiore a quella dei dati indicizzati e con una velocità di esecuzione di *equality-query* notevole poiché in fase di query devono essere estratte solo 4 o 2 bitmap compresse per ogni indice su interi a 32 o 16 bit.

Capitolo 4

Compressione flussi

Archiviare i flussi di rete nelle reti ad alta capacità richiede una grossa quantità di spazio su memoria persistente. Soluzioni comuni come Nfdump permettono di limitare la dimensione degli archivi utilizzando librerie di compressione comunemente disponibili come GZIP, Snappy [57] o LZ4 [58]. Queste tecniche di compressione, anche se permettono di comprimere gli archivi di flussi in maniera molto efficiente, hanno il limite di non riuscire a comprimere piccole quantità di dati e non permettono quindi di estrarre record senza leggere e decomprimere l'intero archivio.

In questo scenario, per comprimere i dati e per permettere di indicizzare i flussi con il sistema di indicizzazione bitmap presentato nel capitolo precedente in modo efficiente, c'è la necessità di trovare soluzioni alternative.

In questo capitolo vengono brevemente descritte alcune tecniche di compressione su interi stantard per la compressione dei dati, viene presentata una breve analisi sulle proprietà dei flussi di rete e viene infine proprosta e validata una possibile tecnica di compressione a blocchi di 1024 Byte che permette di estrarre i flussi richiesti decomprimendo una piccola quantità di dati.

4.1 Tecniche di compressione su interi

In questa sezione vengono brevemente descritte alcune tecniche di compressione su interi.

VByte: VByte [59] è una tecnica di compressione su interi che codifica ogni intero a 32 bit in una sequenza di byte in cui per ogni byte i primi 7 bit rappresentano l'intero e il bit più significativo indica se i byte successivi dell'intero sono diversi da 0. Grazie a questa tecnica ogni intero minore di 128 viene codificato con un solo byte.

Binary packing: Binary packing [60] è una tecnica di compressione su interi che data una sequenza di interi con valore massimo uguale a m, utilizza $\log_2(m) + 1$ bit per codificare ogni elemento.

Frame of reference: Frame of references [61] è una tecnica di compressione su interi in cui una sequenza di interi viene codifica con il valore minimo della sequenza e la differenza rispetto al minimo. in questo modo ogni valore può essere codificato con $\log_2(x_{max} - x_{min}) + 1$.

Simple-9: Simple-9 [62] è ina tecnica di compressione su interi che permette di comprimere sequenze di interi sopprimendo gli zero utilizzando parole a 32 bit in cui per ogni parola i primi 4 bit rappresentano il numero di interi codificabili nei rimanenti 28.

4.2 Analisi flussi

In questa sezione vengono brevemente analizzate le proprietà dei campi più comuni in un flusso di rete, in particolare poiché con NetFlow versione 9 [63] il formato dei flussi è personalizzabile, sono analizzati solo i seguenti attributi:

- IP_SORGENTE (intero a 32 bit).
- IP_DESTINAZIONE (intero a 32 bit).
- FIRST_SEEN (intero a 32 bit) tempo di arrivo del primo pacchetto.
- LAST_SEEN (intero a 32 bit) tempo di arrivo dell'ultimo pacchetto.
- PORTA_SORGENTE (intero a 16 bit).
- PORTA_DESTINAZIONE (intero a 16 bit).
- CONTATORE_PACCHETTI (intero a 32 bit).
- TOT_BYTE (intero a 32 bit).

Le proprietà che sono state notate su questi attributi sono le seguenti:

- in quasi tutti i flussi i campi TOT_BYTE CONTATORE_PACCHETTI contengono un valore minore di 2^{16} (spesso minore anche di 2^8).
- dopo aver eseguito un ordinamento sui flussi (sulla colonna IP_SORGENTE), il numero di valori distinti nelle colonne IP_SORGENTE, IP_DESTINAZIONE, FIRST_SEEN, LAST_SEEN, PORTA_SORGENTE e PORTA_DESTINAZIONE in un blocco di dimensione qualsiasi (anche piccola) è quasi sempre minore della metà rispetto alla grandezza del blocco.

Gli algoritmi presentati nella sezione precedente, riescono a comprimere in maniere molto efficiente gli interi con molti bit settati a 0 nella parte significativa e per questo motivo riescono sono tutti possibili soluzioni per comprimere le colonne CONTATORE_PACCHETTI e TOT_BYTE a piccoli blocchi. A differenza di questi due attributi però, sia gli indirizzi di rete, sia le porte e sia FIRTS_SEEN e LAST_SEEN, enon hanno la proprietà di avere molti bit settati a 0 nella parte più significativa e non risultano quindi comprimibili con algoritmi di compressione su interi standard. In questo scenario per riuscire a comprimere anche gli attributi IP_SORGENTE, IP_DESTINAZIONE, FIRST_SEEN, LAST_SEEN, PORTA_SORGENTE e PORTA_DESTINAZIONE è stato sviluppato un metodi di compressione a blocchi di 1024 byte basato sulla proprietà di avere in media in ogni blocco un numero di indirizzi e porte distinti minore del numero di elementi nel blocco.

4.3 Metodo per comprimere flussi a blocchi

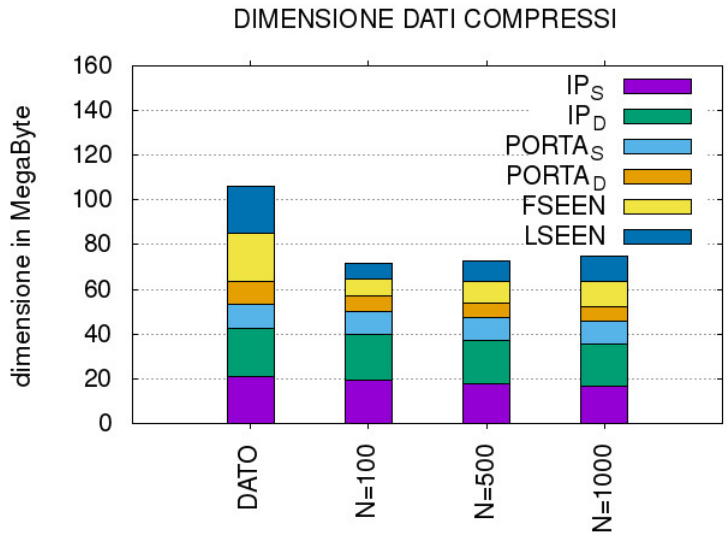
In questa sezione viene presentata una possibile soluzione per riuscire a comprimere indirizzi e porte di un flusso a piccoli blocchi da 1024 byte. La soluzione sviluppata è basata sulla considerazione che dopo aver ordinato i flussi in un blocco di dimensione piccola ci sono in media meno della metà valori distinti rispetto al numero di elementi. Di seguito viene fornita una breve descrizione dell'algoritmo. L'algoritmo considera ogni blocco da 1024 byte preso in input come un array di interi a 32 bit di 256 elementi e comprime ogni blocco nel seguente modo:

- vengono scritti tutti i valori distinti in un vettore di 32 bit
- ogni elemento x di posizione i dell'array da comprimere viene memorizzato in un array a 8 bit alla posizione i con il valore j, dove j è la posizione di x all'interno del vettore a 32 bit contenente solo gli elementi distinti.

In questo modo il blocco compresso sarà formato da 1 byte per descrivere il numero di valori distinti, 4 byte per ogni intero a 32 bit distinto e 256 byte per salvare le posizioni dei valori distinti. Dato N numero di valori a 32 bit distinti, la dimensione del blocco compresso è quindi uguale a $1+4\times N+256$ byte. Infine poiché nel caso in cui ci siano molti valori distinti la dimensione di un blocco compresso sarebbe maggiore di quella di un blocco compresso, è stato deciso che se ci sono più di 140 valori distinti, il blocco non viene compresso (questo caso è identificato da un valore del primo byte superiore a 140).

4.4 Validazione

Dopo aver implementato l'algoritmo descritto nella sezione precedente, l'algoritmo è stato testato su un campione di circa 5 Mega flussi generati da un data set di pacchetti IP di dimensione 9 GigaByte fornito da CAIDA. Il test è stato eseguito su una macchina con processore Intel Core i7-6700 - 3,4 GHz - 8 MB. Poiché in una implementazione reale i dati devono essere ordinati a blocchi al fine di soddisfare lo scopo del tirocinio, nel test viene valutato il livello di compressione dei campi IP_SORGENTE, IP_DESTINAZIONE, PORTA_SORGENTE, PORTA_DESTINAZIONE, FIRST_SEEN e LAST_SEEN al variare del numero N (espresso in Kilo) di record ordinati in blocco. Come per il capitolo precedente, i record sono sempre ordinati sulla colonna IP_SORGENTE. Di seguito viene riportato il risultato del test effettuato.



In conclusione dal grafico è possibile vedere che i dati compressi occupano circa il 70% della dimensione del dato e che all'aumentare della grandezza del numero di record N ordinati in blocco si ha una compressione peggiore, in particolare: la grandezza delle colonne IP_SORGENTE, IP_DESTINAZIONE, PORTA_SORGENTE, PORTA_DESTINAZIONE diminuisce e la grandezza delle colonne FIRST_SEEN e LAST_SEEN aumenta in modo da far aumentare anche la dimensione totale dei flussi compressi. Infine, poiché il tempo impiegato per comprimere le colonne riportate (445 millisecondi) è minore del tempo impiegato per sortare i dati (circa 600 millisecondi), è possibile fare compressione e sort in pipeline in modo tale da pagare solo il tempo di ordinamento dei dati sulla colonna IP_SORGENTE.

Capitolo 5

Conclusioni

Grazie al lavoro svolto durante il tirocinio è stato possibile sviluppare un sistema di indicizzazione bitmap per i flussi di rete, efficiente e con dimensione limitata.

I risultati raggiunti dimostrano come sia possibile creare indici su attributi ad alta cardinalità scegliendo in base al tipo di codifica bitmap il metodo di indicizzazione più opportuno, in particolare sono stati sviluppati indici ottimali per le codifiche bitmap WAH, EWAH, OZBCv2 e per le bitmap non compresse.

È stata sviluppata e pubblicata su github una nuova tecnica di compressione bitmap (OZBCv2), ottimizzata esclusivamente per la creazione di indici bitmap su attributi ad alta cardinalità.

È stato dimostrato che ordinando i flussi di rete sulla colonna IP_SORGENTE e utilizzando OZBCv2 è possibile creare indici efficienti durante la query, con una dimensione poco superiore rispetto alle colonne indicizzate e con una velocità di inserimento superiore a 8 Mega flussi al secondo. Inoltre poiché l'indice OZBCv2 è costruito utilizzando il metodo *Multi-Component equality encoding* ed è composto da 256 bitmap per ogni byte indicizzato, la dimensione di ogni bitmap estratta per risolvere una query sarà in media di $\frac{1}{256}$ della dimensione del dato.

È stata sviluppata una nuova soluzione per comprimere i flussi di rete a blocchi di 1024 byte, in modo tale da permettere di estrarre i valori selezionati senza dover leggere e decomprimere tutti i record.

Bibliografia

[1] Global -2020 Forecast Highlights (2015-2020), http://www.cisco.com/c/m/en_us/solutions/service-provider/vni-forecast-highlights.html.

[2] L. Deri, V. Lorenzetti, S. Mortimer. Collection and exploration of large data monitoring sets using bitmap databases, (2010).

[3] B. Claise. NetFlow Services Export Version, RFC 3954, (2004).

[4] P. Phaal, S. Panchen and N. McKee. inMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks, (2001).

[5] NetFlow Export Datagram Formats, http://www.cisco.com/c/en/us/td/docs/net_mgmt/netflow_collection_engine/3-6/user/guide/format.html.

[6] Amanda Leung. FlowView: How We Did It (Parti 1), (2013), https://www.appneta.com/blog/flowview-flow-records-big-data-1/.

[7] NFDUMP, http://nfdump.sourceforge.net/.

[8] flow-tools, https://code.google.com/archive/p/flow-tools/.

[9] Flow-Scan, https://www.caida.org/tools/utilities/flowscan/.

[10] Stager - A Generic Tools for Presenting Network Statistics, (2010).

[11] SiLK, https://tools.netsa.cert.org/silk/.

[12] A. Pras, R. Sadre, A. Sperrotto. Using NetFlow/IPFIX for Network Management, (2009).

[13] Rick Hofstede. Performance measurements of NfDump and MySQL and the development of a SURFmap plug-in for NfSen, (2009).

[14] R. Hofstede, A. Sperrotto, T. Fioreze, A. Pras. The Network Data Handling War: MySQL vs. NfDump, (2010).

[15] M. Siekkinen, E.W. Biersack, G. Urvoy-Keller. InTraBase: Integrated Traffic Analysis Based on a Database Management System, (205).

[16] P. O'Neil. Model 204 Architecture and Performance, (1987).

[17] V. Sharma. Bitmap Index vs B-tree Index: Which and When?, Oracle Technology Network, (2005).

[18] C-Y. Chan and Y. E. Ioannidis. Bitmap Index design and evaluation, (1998).

[19] C-Y. Chan and Y. E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries, (1999).

[20] P. O'Neil and D. Quass. Improved query performance with variant indices, (1997).

[21] Kesheng Wu, Kurt Stockinger and Arie Shoshani. Breaking the Curse of Cardinality on Bitmap Indexes, (2008).

[22] G. Antoshenkov. Byte-aligned bitmap compression, Technical report, Oracle Corp, (1994).

[23] Abraham Bookstein, Shmuel T. Klein, and Timo Raita. Simple bayesian model for bitmap compression, (2000).

[24] Kesheng Wu, Wkow J. Otoo and Arie Shoshani. An Efficient Compression Scheme For Bitmap Indices, (2004).

[25] Kesheng Wu, Ekow J. Otoo, Arie Shoshani, Henrik Nordberg. Notes on Design and Implementation of Compressed Bit Vectors, (2001).

[26] F. Corrales D. Chiu, J. Sawin. Variable Length Compression for Bitmap Indices, (2011).

[27] M. BHAN, Dr. RAJANIKANTHK, Dr. Suresh KUMAR T.V. Multi-level and Multi-component Bitmap Encoding for Efficient Search Operations, (2012).

[28] H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem and L. Wong. Bit trasposed files, (1985).

[29] Binomial Coefficient, http://mathworld.wolfram.com/BinomialCoefficient.html.

[30] Kesheng Wu, Kurt Stockinger, Arie Shoshani. Analyses of Multi-Level and Multi-Component Compressed Bitmap Indexes, (2010).

[31] Kesheng Wu, Ekow J. Otoo, Arie Shoshani. A Performance Comparison of bitmap indexes, (2001).

[32] Kesheng Wu, J. Otoo and Arie Shoshani. Compressing Bitmap Indexes for Faster Search Operations, (2002).

[33] Kesheng Wu, Ekow J. Otoo, Arie Shoshani. Compressing Bitmap Indexes for efficient query processing, (2001).

[34] Word aligned bitmap compression method, data structure, and apparatus, US 6831575 B2, https://www.google.com/patents/US6831575.

[35] Owen Kaser, Daniel Lemire, Kamel Aouiche. Histogram-Aware Sorting for Enhanced Word-Aligned Compression in Bitmap Indexes (2008).

[36] Francesco Fusco, Marc Ph. Stoecklin, Michail Vlachos. NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic, (2010).

[37] Network analysis, https://www.google.com/patents/US20120054160.

[38] Francois Deliège, Torben Bach Pedersen. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps, (2010).

[39] Alessandro Colantonio, Roberto Di Pietro. Concise: Compressed 'n' Composable Integer Set, (2010).

[40] Z. Chen, W. Zheng, J. Cao, G. Peng. SECOMPAX: A bitamp index compression algorithm, (2014).

[41] J. Chang, Z. Chen, W. Zheng, Y. Wen, J. Cao, W. Huang. PLWA+: A Bitmap Index Compressing Scheme based on PLWAH, (2014).

[42] H. Wang, Z. Chen, Y. Wen, J. Cao, G. Peng, W. Huang. MASC: a Bitmap Index Encoding Algorithm for Fast Data Retrieval, (2015).

[43] Z. Chen, J. Cao, W. Zheng, G. Peng. SPLWAH: A bitmap index compression scheme for searching in archival Internet traffic, (2015).

[44] Z. Chen, W. Zheng, Y. Wen, J. Cao. COMBAT: A new bitmap index coding algorithm for big data, (2016).

[45] Z. Chen, C. Li, J. Cao, W. Zheng. CAMP: A New Bitmap Index for Data Retrieval in Traffic Archival, (2016).

[46] C. Li, W. Zheng, Z. Chen, J. Cao. BAH: A Bitmap Index Compression Algorithm for Fast Data Retrieval, (2016).

[47] S. Kim, J. Lee, S. Rao Satti, B. Moon. SHB: Super byte-aligned hybrid bitmap compression, (2016).

[48] Z. Chen, Y. Wen, J. Cao, W. Zheng, J. Chang, Y. Wu, G. Ma, M. Hakamaoui and G. Peng. A Survey of Bitmap Index Compression Algorithms for Big Data, (2015).

[49] Gheorghi Guzun, Guadalupe Canahueate. Performance evaluation of word-aligned compression methods for bitmap indices, (2015).

[50] `https://github.com/uccidibuti/OZBCv2Bitmap`.

[51] Finding the k-combination for a given number, `https://en.wikipedia.org/wiki/Combinatorial_number_system`.

[52] `https://github.com/lemire/Concise`.

[53] `https://github.com/lemire/EWAHBoolArray`.

[54] `https://github.com/uccidibuti/OZBCv2Bitmap`.

[55] `https://www.gnu.org/licenses/lgpl-3.0.en.html`.

[56] Anonymized Internet Traces 2016, `http://www.caida.org/data/overview`.

[57] `https://github.com/google/snappy`.

[58] `https://github.com/lz4/lz4`.

[59] Jedd Plaisance, Nathan Kurz, Daniel Lemire. Vectorized VByte Decoding, (2015).

[60] `https://github.com/lemire/simdcomp`.

[61] Daniel Lemire, Christoph Rupp. Upscaledb: Efficient integer-key compression in a key-value store using SIMD instructions.

[62] https://github.com/choobin/simple9.

[63] http://www.cisco.com/en/US/technologies/tk648/tk362/technologies_white_paper09186a00800a3db9.html.