Exploiting the Computational Power of Ternary Content Addressable Memory

by

Kamran Tirdad

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2011

Abstract

Ternary Content Addressable Memory or in short TCAM is a special type of memory that can execute a certain set of operations in parallel on all of its words. Because of power consumption and relatively small storage capacity, it has only been used in special environments. Over the past few years its cost has been reduced and its storage capacity has increased significantly and these exponential trends are continuing. Hence it can be used in more general environments for larger problems. In this research we study how to exploit its computational power in order to speed up fundamental problems and needless to say that we barely scratched the surface. The main problems that has been addressed in our research are namely boolean matrix multiplication, approximate subset queries using bloom filters, Fixed universe priority queues and network flow classification. For boolean matrix multiplication our simple algorithm has a run time of $O\left(dN^2/w\right)$ where N is the size of the square matrices, w is the number of bits in each word of TCAM and d is the maximum number of ones in a row of one of the matrices. For the fixed universe priority queue problems we propose two data structures one with constant time complexity and space of $O\left(\frac{1}{\epsilon}nU^\epsilon\right)$ and the other one in linear space and amortized time complexity of $O(\frac{\lg lgU}{\lg\lg\lg U})$ which beats the best possible data structure in the RAM model namely Y-fast trees. Considering each word of TCAM as a bloom filter, we modify the hash functions of the bloom filter and propose a data structure which can use the information capacity of each word of TCAM more efficiently by using the co-occurrence probability of possible members. And finally in the last chapter we propose a novel technique for network flow classification using TCAM.

Acknowledgements

It is my pleasure to thank all the people who made this thesis possible.

It was an honor for me to work on my Master's thesis under the supervision of Professor Ian Munro. His great academic experience and sense of humor made my master's program a fun and fruitful period in my life . Throughout my Master's, he was a great company and our many discussions taught me a lot.

I have also been indebted in the preparation of this thesis to Arash Farzan and Shahin Kamali whose insightful comments was very important to derive the algorithms in chapter 2.

Chapter 3 was not possible without the help of my friend Pedram Ghodsnia and Professor Alejandro López-Ortiz. The last section of chapter 3 was the result of an insightful discussion that I had with Nima Mousavi which I am thankful of him.

Sridar Kandaswamy and Praveen Reguraman were my great accompanies during my internship in Cisco systems and provided a great exposure to many important problems in the computer networking field. One of my discussions with Sridar was the starting point of chapter 4. I should also thank Suran De Silva and Saci Nambakkam who made my internship at Cisco systems possible. It was a great experience.

I should thank my parents and siblings whose support and love is the most valuable thing in my life.

I would also like to thank Tim Hortons for their great coffees.

Dedication

This is dedicated to the humanity including all the species.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ternary Content Addressable Memory(TCAM) is a special type of content addressable memory(CAM). In order to explain TCAM, first we need to explain CAM. The key difference of CAM with a conventional memory is that in a regular memory an address is given to the device and a word corresponding to the address is returned while in a CAM the opposite occurs (i.e. the content of a word is given and the address of the first line that the data occurs is returned). It is easy to build a hash table using CAM along with a regular memory. Each key value is inserted to one line of the CAM and the paired value is stored in the corresponding position in the RAM.

The main difference of TCAM with CAM is that each bit of TCAM can be in 3 conditions of 0,1 or $*$ (don't care state). Essentially in TCAM there is a set of ternary words and each query returns the address of the first word occuring in the TCAM that conforms to the query string. Since each query is done in parallel for each word of the TCAM, each query takes approximately the same amount of time that a read operation takes in a static random-access memory (SRAM) which implies that each query to a TCAM is equivalent of one probe to the RAM in terms of time complexity. Currently the size of each word, which will be denoted by w, in a typical TCAM is between 144 to 1000 bits and has around $256k$ to $1M$ of words. So the size of each word can be assumed to be $\Theta(\lg U)$ where U is

1	0	1	1	1
1	1	0	0	0
1	0	1	0	0
*	*	*	*	*
1	0	*	0	1

Table 1.1: A simple representation of TCAM where each row represents a word in TCAM

Memory	Single-Chip Capacity	\$/chip	\$/MByte	Access speed(ns)	Watts/chip	Watts/MByte
DRAM	128MB	10-20	0.08-0.16	40-80	1-2	0.008-0.016
SRAM	9MB	50-70	5.5-7.8	3-5	1.5-3	0.17-0.33
TCAM	4.5MB	200-300	44.5-66.7	4-5	15-20	3.33-4.44

Table 1.2: Comparison between TCAM and 2 conventional type of memories in terms of cost, power consumption and speed. Table is from [32].

the size of the Universe. For larger values of w which is closer to 1000 it is more realistic to assume that w is in $\Theta\left(U^{\epsilon}\right)$ where ϵ is a small number less than 1.

Table 1.1 illustrates a simple representations of a TCAM where each line shows one word of the TCAM. In this simple representations if we give the query string "10*0*" to this TCAM the return result would be 3 and we can claim that there will not be any input which returns a number more than 4 since the 4th word conforms to all possible quert strings and no query can pass that line.

While TCAM requires much more power in comparison to other type of memories, its fast query resolution makes it a suitable device in special environments. Table 1.2 provides a good comparison between TCAM and two other conventional main memories namely dynamic random-access memory (i.e. DRAM) and static random-access memory (i.e. SRAM).

Adding the primitive operations of TCAM to any computational model (RAM model or cell probe model or pointer machine model) we have the corresponding TCAM model. The simple computational model that we use to represent TCAM is to consider the RAM model with a TCAM besides it. In the RAM model, memory is laid out as a finite array of words. If we know the index of a word we can access it in $O\left(1\right)$ time. The cost of an algorithm in the RAM model is the total number of memory accesses and arithmetic/logical operations. In the accompanying TCAM there are three operations of insert_at_line, disable_line, and query("query string") which each can be done constant time.

The goal of this thesis is to find different ways to exploit the computational power of TCAM in order to improve the time complexity of different algorithms in this new computational model and furthermore proposing new applications and use cases for TCAM.

One simple algorithm that can be improved is boolean matrix multiplication which can be done in $O\left(\frac{dn^2}{w}\right)$ where w is the width of TCAM and d is the maximum number of 1 in a row of one of the matrices. Another interesting problem is fixed universe dynamic priority queue problem. Two algorithms are devised for this problem, one in $O(1/\epsilon)$ time in $O\left(n\frac{1}{\epsilon}U^{\epsilon}\right)$ space and the other one in amortized time of $O\left(\frac{\lg\lg U}{\lg\lg\lg U}\right)$ in $O(n)$ space. Moreover each line of TCAM can be considered a bloom filter in order to solve approximate subset query problem. A data structure namely COCA filter is introduced which improves the average false positive of bloom filters in special environments. Another interesting problem which has been solved using TCAM is approximate nearest neighbor problem[51]. This algorithm was used in order to solve the real time network classification problem.

In the next section we have a case study of a simple algorithm for boolean matrix multiplication and will use a few techniques which will be used in the next chapter extensivly.

1.1 Case Study: TCAM and Boolean Matrix Multiplication

In this section we consider the fundamental problem of boolean matrix multiplication and provide two algorithms which can perform the boolean matrix multiplication in $O\left(\frac{dn^2}{w}\right)$ where w is the width of TCAM and d is the maximum number of ones in a row of one of the matrices. The first algorithm requires $O(n^2)$ auxilary space in the RAM and the second algorithm requires no extra space at the cost of using $\lg n/w$ bits from each word of TCAM . Note that both in boolean and non-boolean square matrix multiplication we have the lower bound of $\Omega\left(n^2\right)$ which is the size of the input Improving the time complexity to $O\left(n^2\right)$ using TCAM, remains an open problem.

The naive approach of square matrix multiplication requires $O\left(n^3\right)$. The naive algorithm for the boolean version of matrix multiplication has been improved to $O\left(\frac{n^3}{\lg n}\right)$ in the RAM model by Arlazarov et al. in [11] using a technique which is known as "four russion" algorithm using $O\left(\frac{n^3}{\lg n}\right)$ bits auxilary space. With simple techniques the extra space can be improved to $O\left(n^2\right)$ bits[12]. Strassen in 1969 [53] provided an algorithm which improved the time complexity to $O\left(n^{\log_2 7}\right) \approx O(n^{2.807})$. The best algorithm in terms of time complexity is due to Coppersmith et al. in [25] with the time complexity of $O\left(n^{2.376}\right)$. To the best of our knowledge there is not any better lower bound for the boolean case of matrix multiplication.

Considering two n by n boolean matrices of A and B whose product is C.

$$C_{n,n} = \begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} \end{pmatrix}$$

Each element of C is defined as:

$$c_{i,j} = (a_{i,1} \wedge b_{1,j}) \vee (a_{i,2} \wedge b_{2,j}) \vee \cdots (a_{i,n} \wedge b_{n,j})$$

Based on the definition an element of C, say $c_{i,j}$, is 1 if and only if there is at least one k such that $a_{i,k}$ and $b_{k,j}$ are both set to 1. Note that once we have found one such k there is no need to perform the rest of the operations.

Before explaining our algorithm consider the simpler case which not only the length of the TCAM is n but also the width of TCAM is also n bits. For the general case we will modify our algorithm later. Since the width of TCAM is n we can put the transpose of B which is denoted by B^T in the TCAM. Now we will show that in $O(n)$ every row of the matrix C can be computed.

For instance consider the ith row of C. First, for every bit position set to 1 in the ith row of matrix A we generate a query string which is set to 1 in the same bit position and the rest of the bit positions are set to *. For example if $a_{i,3}$ is one we generate the query string of "$*^2 1 *^{n-3}$". If this query hits the jth line of our TCAM we conclude that $c_{i,j}$ should be set to 1. Now we temporarily disable the jth line of the TCAM and repeat our operation for the rest of the 1-bit cells in the ith row of matrix A. If the query string is not hitting any line of the TCAM we proceed to the next query. In order to avoid setting a bit in matrix C to one more than once, every time that we hit a line in the TCAM we disable the line in the TCAM and every time that the query misses we proceed to the next query. Hence the total number of operations is not more than $2n$ for each row of C. The rest of the bit positions in the ith row of matrix C are zero. Now we enable all the lines in our TCAM which were disabled and repeat the operations for the next row of matrix A. Therefore we achieve an $O(n^2)$ time.

For the case that the width of TCAM w, is much smaller than n we can modify the algorithm as follows. We can put a narrow strip of B^T in the TCAM (i.e. w columns of B^T). Now the queries should be built by the corresponding w bits of each row of matrix A. If a query hits a row in the TCAM, we not only need to disable that line of TCAM but also have to "remember" that row and that line for the other strips which will be loaded into the TCAM later (and there are n/w of them). If we do not store those positions each query can hit $O(n)$ reduntant rows of B^T which makes the time complexity to be $O(n^3)$ and this is same as the naive case. So an extra $\Theta(n^2)$ space is needed to store the so far hit postions for each row of matrix A. Since with every hit we have to disable n/w lines and may have up to $O(n)$ of those hits for every row of matrix A, the total time complexity for every row would be $O\left(n^2/w\right)$. Note if after every set of queries corresponding to w bit of a row we reload the TCAM which requires $\Theta\left(n\right)$ then just the loading time for every row will be $\Theta\left(n^2/w\right)$ and since there are n rows the total time complexity will be $\Theta\left(n^3/w\right)$. Define d to be the maximum number of ones in any row of matrix A. Then the time complexity of this algorithm is $\Theta\left(dn^2/w\right)$ since for every row we need to do at most $O\left(\frac{dn}{w}\right)$ queries. This algorithm for sparse matrices where $d < w$ is $\Theta\left(n^2\right)$ which is the lower bound as well.

The extra $\Theta\left(n^2\right)$ space can be improved further. If each strip of B^T is mapped to a unique ID number with $\lg\frac{n}{w}$ bits and the ID number is attached to every line of its corresponding strip we can load all the B^T strip by strip to the TCAM at once and for every query string that is generated from a row of matrix A perform all the n/w queries at once. This way in order to "remember" the lines in B^T which need to be disables we only need an extra $O(n)$ space.

For example if n is 6 and w is 4 we can put B^T in two strips with one extra bit for as the ID of each strip. If A and B are denoted as follows

$$A_{6,6} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} \\ a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & a_{5,6} \\ a_{6,1} & a_{6,2} & a_{6,3} & a_{6,4} & a_{6,5} & a_{6,6} \end{pmatrix}$$

$$B_{6,6} = \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} & b_{1,5} & b_{1,6} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} & b_{2,5} & b_{2,6} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} & b_{3,5} & b_{3,6} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} & b_{4,5} & b_{4,6} \\ b_{5,1} & b_{5,2} & b_{5,3} & b_{5,4} & b_{5,5} & b_{5,6} \\ b_{6,1} & b_{6,2} & b_{6,3} & b_{6,4} & b_{6,5} & b_{6,6} \end{pmatrix}$$

The table 1.3 shows how B^T is embeded in the TCAM. If in the first row of A, $a_{1,1}$ and $a_{1,6}$ are the only elements equal to 1 then the query strings related to the first row of C will be $11**$ and $0**1$.

1	$b_{1,1}$	$b_{2,1}$	$b_{3,1}$
1	$b_{1,2}$	$b_{2,2}$	$b_{3,2}$
1	$b_{1,3}$	$b_{2,3}$	$b_{3,3}$
1	$b_{1,4}$	$b_{2,4}$	$b_{3,4}$
1	$b_{1,5}$	$b_{2,5}$	$b_{3,5}$
1	$b_{1,6}$	$b_{2,6}$	$b_{3,6}$
0	$b_{4,1}$	$b_{5,1}$	$b_{6,1}$
0	$b_{4,2}$	$b_{5,2}$	$b_{6,2}$
0	$b_{4,3}$	$b_{5,3}$	$b_{6,3}$
0	$b_{4,4}$	$b_{5,4}$	$b_{6,4}$
0	$b_{4,5}$	$b_{5,5}$	$b_{6,5}$
0	$b_{4,6}$	$b_{5,6}$	$b_{6,6}$

Table 1.3: An example for embeding of B^T matrix in TCAM for boolean matrix multiplication where B is 6 by 6 and w is 4.

Chapter 2

The Fixed Universe Successor Problem

2.1 Introduction

In this chapter we address one of the classic problems/data structures in computational complexity namely fixed universe successor problem. The objective is to design a data structure that stores a set of numbers and supports the following operations:

Insert: add a new number to the set

Delete: delete an element from the set

Successor: Given a number find the closest bigger/smaller number from the set

If all the numbers are from the finite universe U then the problem is called the fixed universe successor problem. A variation of the problem which assumes that the set is defined and only Successor operation is executed. This problem is called the static fixed universe successor problem.

A basic solution for the general successor problem is to use a binary search tree. Then all the operations can be done in $O(\lg n)$ in $O(n)$ space. Adding the extra constraint of fixed universe of U, Van Embde Boas in [54] suggested a method solving this problem in $O(\lg\lg U)$ time and $O(U)$ space. Willard in [55] proposed y-fast trees which can achive $O(\lg\lg U)$ time in space of $O(n)$ space. Beame and Fich in [13] proved the lower bound of $\Omega\left(min\left(\frac{\lg\lg U}{\lg\lg\lg U}, \sqrt{\frac{\lg n}{\lg\lg n}}\right)\right)$ for the static case in the pointer machine model assuming the space is in $O\left(n^{O(1)}\right)$. For the dynamic case best lower bound is that of Mehlhorn et al. in [41] who proved the lower bound of $\Theta(\lg\lg U)$ in the pointer machine model.

2.2 Static Fixed Universe Successor problem

The static fixed universe successor problem was solved in [47] in constant time. They also claim that the dynamic case can also be done using TCAM in constant time; however, their method appears to be incorrect. In this section we provide their idea with a slightly different setting to show how this problem can be solved using TCAM in constant time and why this technique can not be generalized for the dynamic case. We defer the dynamic case to the next section.

The approach of [47] is based on Theorem 1. First we need to provide some definitions. Since integers are from the bounded universe U, they are represented by $\lg U$ bits each.

Definition 1 *LCP(a, b): The longest common prefix of the bits of two numbers a and b is LCP(a, b)*

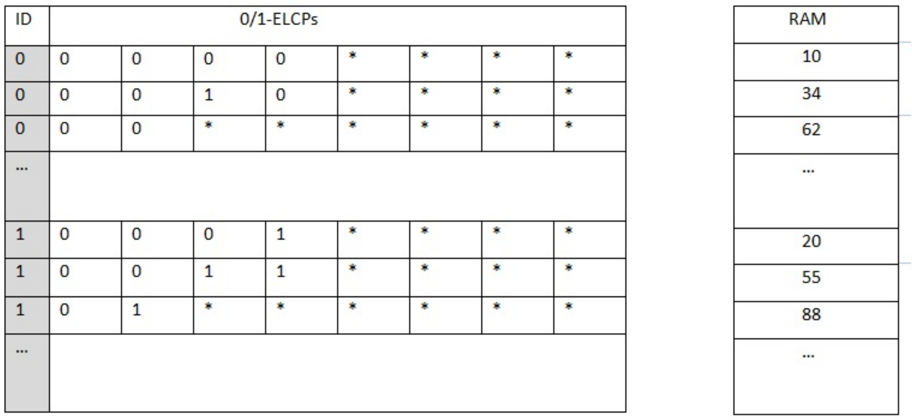
Definition 2 0-*ELCP(a, b) is LCP$(a, b)\,0$ which a zero bit is concatenated with it on the right side and* 1-*ELCP(a, b) is defined similarly.*

Definition 3 *TCAM block: K multiple TCAMs can be simulated by one TCAM which has lgK bits more. First a unique ID number is assigned to each TCAM. In order to address each of these TCAMs we need to concatenat the corresponding ID of the TCAM to the query. We call each of these TCAMs a* TCAM Block.

Theorem 1 *For a sorted set of numbers $S = \{a_1, a_2, ..., a_n\}$ we store* 0-*ELCP(a_i, a_{i+1}) for i from 1 to $n-1$ in one TCAM block and store* 1-*ELCP(a_i, a_{i+1}) for i from 1 to $n-1$ in another TCAM block sorted by their length (and padded with $*$s). Then for an arbitrary number q, the result from querying q to one of these two blocks would be the predecessor or successor of q among the set S.*

Proof. Note that the number a_i which has the longest prefix matching with q is the successor or predecessor of q. If we can prove that one of these two TCAM blocks will hit the longest prefix matching q then the proof is done. The following proof is a minor modification of that in [47].

Let P denote the LCP for a_i and q. Note that q must match one of the ELCPs, $P1*\cdots*$ or $P0*\cdots*$. Without loss of generality assume that it matches $P1*$. We need to prove that there is no longer prefix in TCAM block 1-ELCP matching q. Let us assume the contrary and say that q matches a longer prefix ending with 1. It must be of the form $P1Q1*\cdots*$. Let R‘ denote the range for the ELCP, $P1Q1*\cdots*$. We claim that the range R and R‘ must have at least two points in common, thus getting a contradiction and these two points are $u = P1Q01\cdots1$ and $v = P1Q10\cdots0$. Since P1Q is the LCP for range R, the above points are in R. To prove that they are in R, we will show that they lie between



ID	0/1-ELCPs							
0	0	0	0	0	*	*	*	*
0	0	0	1	0	*	*	*	*
0	0	0	*	*	*	*	*	*
...								
1	0	0	0	1	*	*	*	*
1	0	0	1	1	*	*	*	*
1	0	1	*	*	*	*	*	*
...								

RAM
10
34
62
...
20
55
88
...

Figure 2.1: The set consists of elements 10, 20, 34, 55, 62, 88. The 0-ELCP is recognized by bit ID of 0 and 1-ELCP with 1. The Longest Common prefix of every two consecutive number is calculated and the zero concatenated one is in the 0-ELCP pointing to the smaller number and one concatenated one is in the 1-ELCP block pointing to the bigger number. In this example querying 51 in block 0-ELCP returns 62 and in the 1-ELCP returns 55 and 55 is the successor of 51 which can be identified correctly.

two points in R. Look at $y = P01\cdots1$. Clearly, u and v are greater than y. Also since q matches $P1Q1*\cdots*$, $u < q$ and $v <= q$. So we have: $y < u < v <= q$ and y is in R. Also q is given to be in R. So, u and v must be in R, thus getting a contradiction.

Figure 2.1 illustrates how this technique can be used to solve the static case. In the dynamic case there are two major problems which increase the time complexity to $O(\lg\lg U)$ matching that of the RAM model. The first problem is that in order to insert an element after finding its successor we need to calculate the longest common prefix of the inserted element and its successor. In [47] it is assumed that there exists a special hardware which is doing this operation in constant time. While with the regular logical operations of XOR and bit shifting finding the LCP of two numbers needs at least $\lg\lg U$. In fact there is no need for special hardware to find the LCP of two numbers in constant time. Consider a *block* of TCAM with bit strings of $1*\cdots*$, $01*\cdots*$, $001*\cdots*$,$\cdots$,$00\cdots01$. If we XOR the two numbers and query the result of XOR to this block, one entry of this block will be hit and the number of zeros in that entry corresponds to the LCP of those two numbers. Hence the LCP of any two numbers can be obtained in constant time.

The essential problem which has not been addressed by [47] is how to manage the space between the entries in the 1-ELCP or 0-ELCP TCAM. In order for the algorithm to work correctly an element with more don't care bits should be in a lower place in the TCAM in comparison with an element with less don't care bits. Keeping all these entries ordered and padded with extra space for future inserts and deletes is a very difficult task which has not been addressed by [47] at all.

If for every level of our binary trie for every element we reserve one entry in the TCAM the total space required would be $O(n \lg U)$ which is not linear and in fact very costly. If we do not reserve that many entries then the problem is very complicated given that we need to manage the memory in (amortized) constant time. In this binary trie from level $\lg \lg U$ up to level $\lg U$ we can have up to $\frac{n}{2}$ entries. On the other hand with each insertion we can delete from almost any level and add to almost any level. If the hardware supports a special operation that shifts all the values in the TCAM after one given index downward then the dynamic case is easy. Currently TCAM does not support this operation and we do not know how costly it is to make the TCAM support it.

In the next section we propose two algorithms for the dynamic case without making any of the mentions extra assumptions. The first algorithm achieves the time complexity of $O\left(\frac{1}{\epsilon}\right)$ in the space of $O\left(nU^{\epsilon}\left(\frac{1}{\epsilon}\right)\right)$ and the second algorithm improves the space to linear at the cost of increasing the running time to amortized $O\left(\frac{\lg \lg U}{\lg \lg \lg}\right)$ which is better than the best existing result in the RAM model which is $O\left(\lg \lg U\right)$ (using y-fast tries).

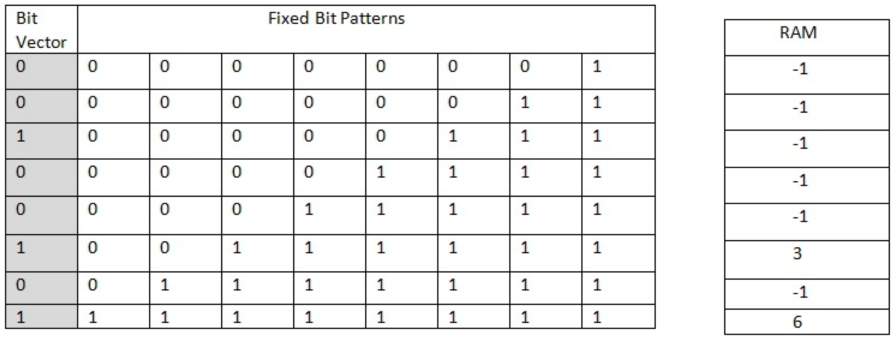
2.3 Dynamic Fixed Universe Successor problem

2.3.1 Solving In Constant Time using $O(nU^{\epsilon}(1/\epsilon))$ lines of TCAM

The first algorithm that we propose needs $O\left(\frac{1}{\epsilon}n\left(U^{\epsilon}\right)\right)$ lines of TCAM. In order to explain it clearly let us first make the extreme assumption that the width of the TCAM is $U+1$ bits and we have U lines in the TCAM. Now assume that the right most bit of each line of the TCAM is a bit vector where inserted elements are set to 1 and the rest of the elements are set to 0.

In order to insert/delete element i from our data structure we need to modify the right most bit of the i-th line of the TCAM. For the rest of the U bits of each line of the TCAM assume that we have the following strings line by line from top to bottom: 111...1, 011...1, 001..1, 00...01. In order to find the successor of element i the query string is a $U+1$ bit vector where the i-th and $(U+1)$-th bits are set to 1 and the rest of the elements are set to $*$.

In order to support predecessor each inserted element should keep a pointer to its predecessor. Whenever that we insert an element, say X, first we find its successor, say $S(x)$, in constant time and assign the value of the predecessor of $S(X)$ to the predecessor of X and modify the predecessor of $S(X)$ to X. For deletion similarly before deleting an element, say X, we store its predecessor in some temporary variable and update the predecessor variable of $S\left(X\right)$ to that temporary variable. Hence all the operations take



Bit Vector	Fixed Bit Patterns							
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	1
1	0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1	1
1	0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

RAM
-1
-1
-1
-1
-1
3
-1
6

Figure 2.2: An example where U is the set of all integers from 1 to 8 and elements of $3, 6, 8$ are inserted. For each line on the TCAM we have a corresponding variable which stores the predecessor of the value if the value is inserted and otherwise stores -1. The query string to find the successor of 4 is 1****1***

constant time in the space of $O\left(U^2 + U \lg U\right)$ bits. Figure 2.2 is an example where U is the set of integers from 1 to 8.

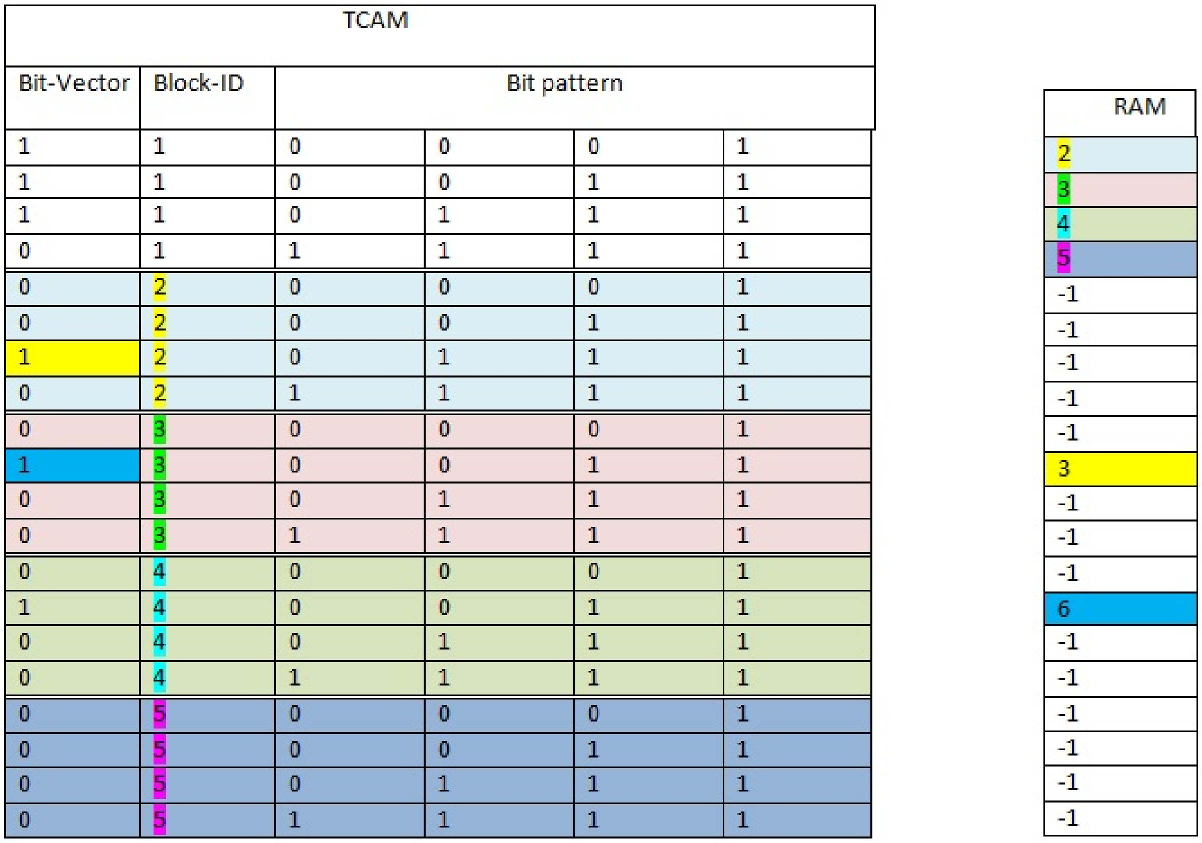
We can improve the width of TCAM to U^ϵ while the time complexity remains small $O\left(1/\epsilon\right)$. The first trick is analogous to what is done in Van Embde boas trees [54]. Let us assume that each line of TCAM has a width of $(ID\_Size) + U^\epsilon + 1$ where ID_Size will be defined later on and ϵ is a constant number less than one. The main idea is to divide the TCAM into blocks of size U^ϵ lines with unique ID numbers attached and consider each block implicitly as a node of a tree with U^ϵ children. This way query resolution at any node of the tree (i.e. each block of the TCAM) will reduce the problem space by a factor of U^ϵ. Hence the height of the implicit tree is $\frac{1}{\epsilon}$. Each line in each block points to a variable in the RAM which is the ID Number of the next block which should be followed if a query hit that line. Figure 2.3 shows an exmaple of this technique were $\epsilon = \frac{1}{2}$ and U is the set of integers from 1 to 16.

The total number of blocks is $1 + U^\epsilon + \left(U^\epsilon\right)^2 + ... + \left(U^\epsilon\right)^{\frac{1}{\epsilon}-1}$. So the total number of bits in order to uniquely identify each block would be

$$\lg\left(1 + U^\epsilon + \left(U^\epsilon\right)^2 + ... + \left(U^\epsilon\right)^{\frac{1}{\epsilon}-1}\right) = \left(1 + 2 + ... + \left(\frac{1}{\epsilon} - 1\right)\right)\epsilon \lg U$$

$$= \left(\frac{1}{2\epsilon} - \frac{1}{2}\right)\lg U$$

and this is the size of the ID_Size in bits. The total number of lines of the TCAM also can be calculated as follows: $U^\epsilon + \left(U^\epsilon\right)^2 + ... + \left(U^\epsilon\right)^{\frac{1}{\epsilon}}$ which is $O\left(U\right)$ lines.



TCAM					
Bit-Vector	Block-ID	Bit pattern			
1	1	0	0	0	1
1	1	0	0	1	1
1	1	0	1	1	1
0	1	1	1	1	1
0	2	0	0	0	1
0	2	0	0	1	1
1	2	0	1	1	1
0	2	1	1	1	1
0	3	0	0	0	1
1	3	0	0	1	1
0	3	0	1	1	1
0	3	1	1	1	1
0	4	0	0	0	1
1	4	0	0	1	1
0	4	0	1	1	1
0	4	1	1	1	1
0	5	0	0	0	1
0	5	0	0	1	1
0	5	0	1	1	1
0	5	1	1	1	1

RAM
2
3
4
5
-1
-1
-1
-1
-1
3
-1
-1
-1
6
-1
-1
-1
-1
-1
-1

Figure 2.3: In this example U is the set of integers from 1 to 16 and ϵ is $\frac{1}{2}$. The inserted elements are $3, 6$ and 10. If we want to find the successor of 9 first we use the 2 significant bits of 9 in the first block. The query string would be "1,001,*1**" and will hit the third line of TCAM which points to block ID 4. Now with the other 2 bits of nine we need to build a query string for the block with ID of 4. The query would be "1,100,**1*" which will hit the 10th line of the TCAM which corresponds to number 10 with predecessor of 6. In this figure cells in the RAM are coloured same as their corresponding regions in the TCAM.

Another improvement is to remove all the empty blocks from the TCAM. Moreover we add a counter for each block which represents the total number of existing elements in the block. Once all the elements of a block are deleted and the counter is zero we can keep a pointer to that block in a list of "free" blocks. This results in a huge improvements in the space needed to store the inserted elements. Assuming the extreme case that each inserted element is occupying different blocks at any level of our implicit tree, it requires $\frac{1}{\epsilon}$ blocks and since there are n inserted element, the total number of lines required in the TCAM is $O\left(n\frac{1}{\epsilon}U^{\epsilon}\right)$. Note that this is an upper bound and if the n numbers preserve some locality this can be a lot smaller. The ID_Size is also reduced to $\lg\frac{n}{\epsilon}$ bits. Note that there exists cases that most of the lines of the majority of the blocks are empty which is a weak point of this algorithm. Moreover the space required here is still far from our goal of $O(n)$ lines in the TCAM. On the other hand this algorithm is very fast and changes to memory is minimal. Note that all the blocks are initialized at the beginning of the algorithm and afterwards we only need to modify one bit of $\frac{1}{\epsilon}$ lines of the TCAM and their corresponding pointer in the RAM.

2.3.2 Solving in Linear space

In this section we consider the case where the space remains linear at the cost of increasing the time complexity. The main idea is to reduce the problem from n elements in universe U to $\Theta(\lg U)$ elements in universe $U/(\lg U)$ with the help of TCAM and modification on y-fast trie data structures[55]. First as a reminder let us have the definition of x-fast and y-fast tries.

Definition 4 *x-fast trie: x-fast trie is an implicit bitwise trie for a set of numbers. All values are stored at the leaves and the internal nodes are stored if there is a number in their subtree or equivalently they are the prefix of an existing element. Each level of this abstract bit-trie is stored in one hash table (using dynamic perfect hashing). Furthermore for every internal node in the bit-trie that does not have a left child we have a pointer to the minimum leaf value in the right sub-tree and for every internal node that does not have a right-child we store a pointer to the leaf element that is the maximum value in the left sub-tree. Since for each element inserted into our set we need to add at most lg U nodes to lg U different hash tables the space complexity of x-fast tries is* $O(n\lg U)$.

In order to do successor or predecessor operations for a given query value t first we need to find the internal node that has the longest common prefix with t. In order to find such a node we need to do a binary search among the $(\lg U)$ *hash tables which requires* $\lg\lg U$ *operations (i.e. first check the* $\left(\frac{\lg U}{2}\right)$*th hash table if it exists then check the* $\left(3\frac{\lg U}{4}\right)$*-th hash table and so on.) In order to do insert/delete we need to update every single level or equivalently every hash table and the internal pointers in* $O(\lg U)$*. Figure 2.4 shows a simple example of an x-fast trie.*

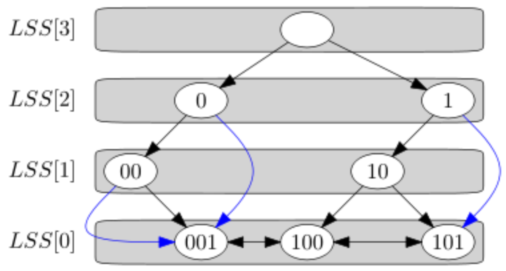


Figure 2.4: An x-fast trie containing the integers 1 (0012), 4 (1002) and 5 (1012). Blue edges indicate descendant pointers. picture is from [1]

Definition 5 *y-fast Trie: The main problems with x-fast trie is that its space is $O(n \lg U)$ and the time complexity of insert and delete is $\Theta(\lg U)$. In order to solve this problem the following approach was suggested by Willard: The elements are divided into groups of $O(\lg U)$ consecutive elements and for each group a balanced binary search tree is created. To facilitate efficient insertion and deletion, each group contains at least $\left(\frac{\lg U}{4}\right)$ and at most $2 \lg U$ elements. For each group one element is chosen as the representative of the group and the x-fast trie is built for $\left(\frac{n}{\lg U}\right)$ representative elements. Since the x-fast trie stores $O(n/ \lg U)$ representatives and each representative occurs in $O(\lg U)$ hash tables, this part of the y-fast trie uses $O(n)$ space. The balanced binary search trees store n elements in total which uses $O(n)$ space. Hence, in total a y-fast trie uses $O(n)$ space. In order to do a successor or predecessor query for value q, first we need to find its predecessor and successor representative elements say r_1 and r_2. Since $r_1 < q < r_2$ the successor or predecessor of q are in one of these two binary trees which corresponds to r_1 or r_2.*

So far we did $O(\lg \lg U)$ operations and with another "binary" search in these two binary trees of size $O(\lg U)$ in time of $O(\lg \lg U)$ we can find the successor and predecessor of q. Note that after every $\lg U$ insertions we might need to do a re-balancing of the trees and add a new representative element to the x-fast trie. Re-balancing of the tree can be done in liner time in the size of tree which is $O(\lg U)$ and the insertion of all the prefixes of the new representative in the x-fast tree requires $O(\lg U)$ insertions in the corresponding hash table. So it is amortized constant time. Figure 2.5 shows a ilustrates the general idea of y-fast trie.

Now let us get back to our own data structure. We store every level of the bitwise trie of the x-fast trie part of the y-fast trie in the TCAM starting from the last level (the leaves). For each level we need $\left(\frac{n}{\lg U}\right)$ entries in the TCAM. For example for figure 2.4 the TCAM will be populated as follows: $001, 100, 101, 00*, 10*, 0**, 1**, ***$ Note one query to the TCAM gives us the longest prefix of the query value in constant time. So instead of $O(\lg \lg U)$ operation we can do this part in $O(1)$. The second part (i.e. binary search in a

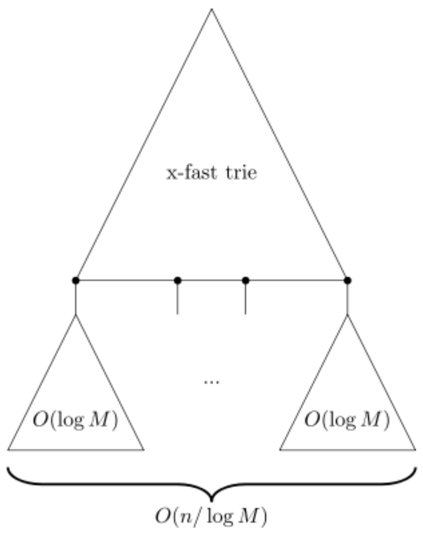


Figure 2.5: a simple ilustration for y-fast trie from [2]

tree of size $\lg U$) still requires $O(\lg\lg U)$ time. If we replace the binary search trees (which is usually red-black trees) with fusion trees [30] the time complexity of the second part can be improved to $O\left(\frac{\lg\lg U}{\lg\lg\lg U}\right)$. After every $\lg U$ insertions we might need to rebalance the fusion trees. Our simple approach is to rebuilt it in a new fusion tree. This requires amortized time of $O\left(\frac{\lg\lg U}{\lg\lg\lg U}\right)$ for rebalancing the trees as well. Now in this data structure updating of the TCAM can be done in amortized constant time and updating the fusion trees can be done in amortized $O\left(\frac{\lg\lg U}{\lg\lg\lg U}\right)$. Moreover predecessor/successor queries can be carried on in $O\left(\frac{\lg\lg U}{\lg\lg\lg U}\right)$ in linear space.

The main open problem of this section is that given $\Theta(\lg U)$ numbers from universe U is it possible to solve the successor problem better than fusion trees in linear space which is $\Theta(\lg U)$? If we can solve this problem in constant time then the dynamic version of fixed universe successor problem can be solved in amortized constant time as well. This can be a starting point for future studies of this problem using TCAM.

Chapter 3

TCAM and Signature Files

3.1 Overview

In this chapter we consider the case that every line of TCAM is a signature of a set (file) in order to build a database. Goel et al in [32] have studied the use of TCAM as a set of Bloom Filters. Since each line of TCAM is relatively small, saving space by using extra information from statistical properties of the sets can be very useful. In this chapter we propose a new indexing data structure based on a novel variation of Bloom filters in order to utilize the information capacity of a TCAM more efficiently. Moreover this data structure can be very useful even in a conventional computer without the TCAM. We called this new data structure COCA Filter, a new type of Bloom filter which exploits the co-occurrence probability of elements in sets to reduce the false positive error. We show experimentally that by using this technique we can reduce the false positive error by about 21 times for the same index size. Furthermore Bloom filters can be replaced by COCA filters wherever the co-occurrence of any two members of the universe is identifiable.

3.2 Introduction

Inverted indices and variants thereof are the preferred data structures currently in use in search engines. However in environments that are very sensitive to index size this method becomes impractical since they require approximately 50% of the size of the corpus for the index file. By compressing the index file and pruning of the less frequent query terms we can reduce the size of inverted indexes down to 10% of the corpus size [57]. In areas where false positive errors are acceptable a more space efficient method called signature files is applicable. With this method it is possible to reduce the size of index file significantly at the cost of precision. Another key advantage of this method is that it can be used in optimizing intersection queries in distributed inverted indices[38, 48]. Parallelizability and the simplicity of the insertion operation are two other benefits of this method that makes it a suitable choice for certain environments.

When using signature files a signature is maintained for each document. A signature is basically a sequence of bits. There are several different methods for computing the signature of a document. One of the most common methods is to use a randomized data structure called Bloom filter. In Bloom filter-based signature files it is implicitly assumed that every pair of words is equally likely to appear in the same document while in practice this assumption is not true.

In this Chapter we introduce a new variant of Bloom filters named co-occurrence aware Bloom filters or COCA Filters for short. COCA Filters utilize the probability of the co-occurrences of the words in documents to improve the false positive error. We show through experiments that COCA Filters can reduce the space by up to 75% for the same false positive error or equivalently reduce the false positive error by up to 21 times for the same index size.

Reducing the size of the signature file index or equivalently its false positive error makes COCA Filters ideally suited for applications which are extremely sensitive to the size of the storage.

The rest of this chapter is organized as follows: Section 3.2 reviews related work and background. Section 3.3 describes the details of our approach and our proposed methods. Section 3.4 presents the evaluation and analysis of our proposed methods and finally we conclude our work in section 3.5.

3.3 Previous Work and Background

Inverted file indices and signature files are two well established indexing methods which have been proposed for large text databases [21, 28, 57]. Although using the inverted files is more favorable because of its wide range of useful properties in comparison to signature files, Carterette and Can in [21] showed that signature file indexes can be as good as inverted file indexes in special environments where memory is scarce and a given false positive rate is acceptable. Library catalogues, multimedia files with many attributes, medical cross references, and a large lexicon or lists of streets for a GPS system are examples of text databases in which signature file can work faster with less storage. However, the high false positive error rate is one of the critical problems of the signature file method which makes it impractical for many applications.

Signature files are a forward index method which stores a signature for every document. Hashing every single term of a document and concatenating the hash values of the terms can be considered as a simple signature for that document. Alternatively, superimposed coding can be used to create a signature of a document. In this method, hashing every word of the document yields a bit pattern of size m, with k bits set to 1, in which m and k are design parameters. The bit patterns are superimposed (OR-ed) together to form the document signature. Searching for a set of words is handled by creating the signature of each word and OR-ing them together to build the query signature and returning all documents with a matching pattern.

To avoid having document signatures that are flooded with 1s, long documents are divided into smaller blocks, that is, pieces of text that contain a constant number of unique words. Each block of a document gives a block signature and block signatures are concatenated to form the document signature.

There exists different ways of managing signature files in the main memory. These different ways can be categorized to three main groups of sequential storage, vertical partitioning and horizontal partitioning. In sequential storage methods, as it is suggested by its name, all signature files are concatenated sequentially. In vertical partitioning methods, the signatures are stored column-wise. Since for answering a query just a certain number of files should be searched, this technique improves the query time at the cost of insertion time. Bit-sliced [27] [49] and frame-sliced are two variations of vertical partitioning methods.

In bit-sliced partitioning (figure 3.1), the width of each column is equal to 1 bit. In other words, in this method, the i-th bit of all signatures (i=1 to m) is stored in the ith file.

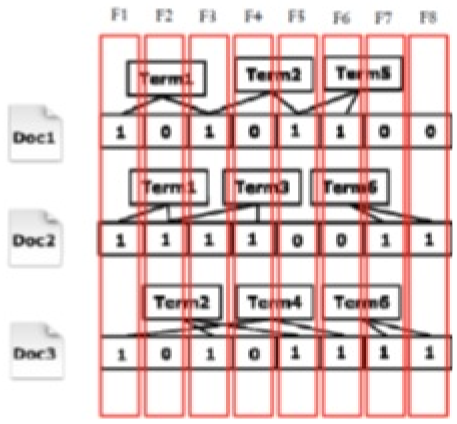


Figure 3.1: Bit Slice partitioning of Signature files

In frame-sliced partitioning (figure 3.2) [39], the width of each column is more than 1 bit. This method forces each word to hash into bit positions that are close to each other in the document signature. Then, these columns of the signature matrix are stored in the same file and can be retrieved with few random disk accesses. This method is a trade-off between query time and insertion time.

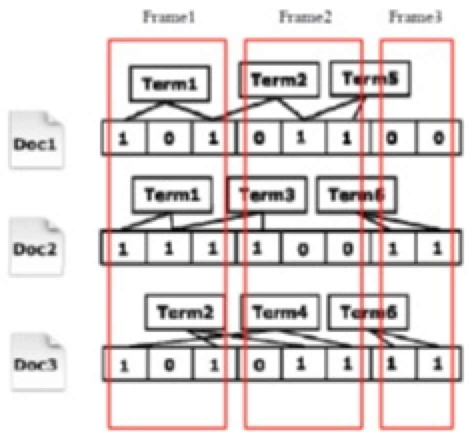


Figure 3.2: Frame Slice partitioning of Signature files

The main motivation behind horizontal partitioning is to achieve better than linear search time while in the above mentioned methods we need to scan all the signatures. In this method, signatures are grouped into sets. The grouping can be done based on some priori rules to partition signature files or in a hierarchical way (like a B-tree). Note that in TCAM the query time is linear in the size of output and insertion and deletion can be done in constant time so none of these problems can happen while using TCAM.

Although not explicitly stated in the literature, superimposed coding is a variation of Bloom filter, a well-known randomized data structure first suggested in [15]. A Bloom filter is a probabilistic data structure used to check whether an element belongs to a set with possible false positive error but zero false negative error. It consists of a bit vector of size m and k independent hash functions $h_1, h_2, ..., h_k$ with ranges of $1, ..., m$. All the bits are initially set to zero. These hash functions can be interpreted as uniform random number generators over the range of $1, ..., m$. For every element of the set, say x, the bits $h_i(x)$ for $(1 \leq i \leq k)$ are set to 1. Some bits of the array might be turned on more than once, but this will not affect the status of the array. To check if an item, say y, is a member of S, the k positions of $h_i(y)$ for $1 \leq i \leq k$ in the array should be checked and if one or more of the k positions are set to 0, it can be assured that y has not been inserted to this array and consequently is not a member of S. If all k positions are set to 1, it is assumed that y is in S. However, there is some probability that this assumption is wrong. Therefore, a Bloom filter may result in a false positive error, also known as false drop error.

Bloom filters have been used in a wide variety of applications in recent years. They are used as spell-checkers [42], as a means of succinctly storing a dictionary of unsuitable passwords for security purposes [52], to speed up semi-join operations [43], for Web cache sharing [29] and in many other areas. In order to support multi-sets, Cohen and Matias introduced spectral Bloom filters [24]. Chazelle et al. in [23] introduce a similar data structure which is called a Bloomier filter in order to approximate functions.

Bloom [15] proved that the false positive probability of the Bloom filter is about $f = (1 - \frac{e^{-kn}}{m})^k$. Recently Bose et al. in [16] showed that the Bloom's approximate formula for false positive is not accurate and gave a proper formula. They also demonstrated that for large enough values of m (size of Bloom filter) with small values of k (number of hash functions), the difference between Bloom's formula and the actual false positive rate is negligible.

To obtain an estimate of the efficiency of Bloom filters, it is good to know the information theoretic lower bound of the size of any data structure that can represent all sets of n elements from a universe with false positive for at most a fraction t of the universe but allows no false negative. Broder et al. [17] showed that to achieve a false positive rate less than t, we must have $m > n\ lg(\frac{1}{t})$ bits. Furthermore, they showed that this lower bound in Bloom filters is $m > n\ lg(e) \cdot lg(\frac{1}{t})$ and consequently argued that space-wise Bloom filters need more than a $lg(e) \simeq 1.44$ factor of the information theoretic lower bound. In [46] Pagh et al. introduced a more complicated data structure that achieved this lower bound.

One of the key observations in both of the aforementioned proofs is the assumption that there is no correlation between any two members of the universe, and any subset of the universe with cardinality of n is equally likely. Now assume that some members of the universe are strongly correlated (i.e. given that one of them belongs to a subset, the probability that the other is also a member of that set is very likely). Intuitively this property of possible subsets can be exploited by using special hash functions which produce more "similar" bit patterns (i.e. with smaller hamming distances) for more correlated members of our set and vice versa. For doing so, we use locality sensitive hash (LSH) functions which are customized to hash similar items to the same hash value with high probability [22]. The LSH algorithm has been used in numerous applied settings from bio-sequence similarity search [20] to audio similarity identification [56] and many other areas [31, 33, 45]. Min-wise independent permutations is a locality sensitive hashing scheme for a collection of subsets with the similarity function defined as follows:

$$Pr_{h \in F}[h(A) = h(B)] = sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In this setting, hash of a set A is defined as $h(A) = min_{a \in A} \pi(a)$ where π is a permutation which was chosen randomly from a min-wise independent permutation family F. A permutation family F (subset of all n factorial permutations) is *min-wise independent* if for any subset X of $[1...n]$, and any $x \in X$, when π is chosen randomly from F we have $Pr(min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|}$ [19].

One of the applications of min-wise independent hash functions was suggested by Broder in [18] to detect near duplicate documents over a large set of documents. Broder suggested to consider a set of shingles (contiguous subsequences of words) for each document and choose a set of t independent random permutations $\pi_1, \pi_2, \pi_3, ..., \pi_t$. For each document D, calling its set of shingles SD, he defined the sketch of Document D as $(min_{a \in SD}\pi_1(a); min_{a \in SD}\pi_2(a); ...; min_{a \in SD}\pi_t(a))$. Then, he argued that the sketch of two documents can be used to estimate their resemblance by computing how many corresponding elements in their sketches are equal. In the next section a similar approach is taken in order to reduce the false positive errors of signature files.

3.4 COCA Filters

Considering the concept of signature files over human readable texts, some terms (members of the universe) are more likely to exist in the same document (set); in other words positively corelated.

In order to exploit this non randomness, it is preferable to modify the k hash functions of the Bloom filter such that “similar” words (i.e. with high co-occurrence ratio) have “similar” bit patterns (i.e. with less hamming distance). For example if two words occur in almost the same set of documents their bit patterns can be designed such that they differ in a few places. More importantly by using these bit patterns, after inserting these two words there would be more bit positions available for the rest of the words in the Bloom filter causing reduction in the average false positive error. This observation can be formalized as the following optimization problem.

Consider two keywords of x and y from the universe of all the words W with corresponding posting lists of X and Y. Furthermore assume that the k bit positions of each of these two terms are stored in the sets of $H(x)$ and $H(y)$. Rather than having k random numbers between 1 to m, the proposed objective is to design hash functions such that:

$$\forall x, y \in W, \frac{|H(x) \cap H(y)|}{|H(x) \cup H(y)|} = \frac{|X \cap Y|}{|X \cup Y|}$$

Note that the right hand side is determined by the corpus and so is fixed. So this problem can be characterized as given N^2 rational numbers p_{ij} design a bipartite graph with m vertices on one side and N vertices the other side such that for every two vertices

V_i and V_j where $0 < i, j \leqslant N$ the following constraints hold:

$$\frac{|Neighbour(V_i) \cap Neighbour(V_j)|}{|Neighbour(V_i) \cup Neighbour(V_j)|} = p_{ij}$$

We conjecture that this problem is NP-hard when we are given p_{ij} as a pair I_{ij} (intersection size) and U_{ij} (union size). Here we propose the following ad-hoc probabilistic approach. Define k co-occurrence-aware hash functions of x to be k of the min-wise independent permutations over the set of X. So, the probability that each hash of two distinct terms x and y be equal to each other is $\frac{|X \cap Y|}{|X \cup Y|} = sim(x, y)$. This new data structure is named co-occurrence-aware Bloom filters or in short *COCA filter*.

Assuming that the probability that two different hash functions produce the same bit position for two different words is negligible, the expected value of the difference between the left and right hand side of the objective function for every two term can be calculated as follows:

$$E\left[\left|\frac{|H(x) \cap H(y)|}{|H(x) \cup H(y)|} - \frac{|X \cap Y|}{|X \cup Y|}\right|\right] \simeq \tag{3.1}$$

$$\left|\frac{k \times sim(x, y)}{2k - k \times sim(x, y)} - sim(x, y)\right| = \tag{3.2}$$

$$\left|\frac{sim(x, y) \times (sim(x, y) - 1)}{2 - sim(x, y)}\right| \tag{3.3}$$

The approximation from (1) to (2) is based on the assumption that pairs of sets with large intersections on average have large unions but obviously this is not true in general. Since $sim(x, y) = \frac{|X \cap Y|}{|X \cup Y|}$ and is in $[0, 1]$, with simple algebraic calculations it can be shown that this value is less than 0.172 and more importantly for the pairs of x and y such that $sim(x, y)$ is close to 0 or 1 this value is close to 0. So over the sets that most of the members are strongly co-related or are not related to each other at all this approach can perform very well. Note that the reason this formula is not dependant on k, the number of hash functions, is the implicit assumption that the bit vector is large enough such that it is quite unlikely for two different hash functions to produce the same bit position for two different words.

Note that by doing so, the reduction in the false positive probability for all of the terms in documents happens at the cost of increasing the false positive probability over random terms which do not exist in any of the documents. In the next section we describe three experiments over three different English corpora comparing COCA Filters to traditional Bloom Filters.

In order to implement COCA filters, k min-wise independent permutations should be picked randomly from a min-wise independent family. Since min-wise independent families are too big for practical applications (in fact it is known that their size is at least $lcm(1, 2, ..., n)$ [19]), variant notions of min-wise independence have been introduced in the literature [40, 50].

In our experiments in order to keep the implementation relatively simple two-universal hash functions has been employed to replicate the behaviour of k independent permutations. For a large prime value of p, k random pairs of (a_i, b_i) are generated where $1 \leqslant i \leqslant k$. The hash of each document ID, say x can be calculated by $(a_i * x + b_i) mod p$. Each hash of the document IDs corresponds to one permutation over the set of all document IDs. This procedure is repeated for all the k random pairs so that there are k different permutations. Consequently, for each permutation the minimum value of each posting list is the hash of the corresponding keyword.

In algorithm 1, the pseudocode as explained in the last paragraph shows how to calculate the k hash functions of the COCA filter and store them in k hash tables $h_1, h_2, ..., h_k$.

Algorithm 1 Hash Calculator For COCA Filter

input: Assume documents are numbered from 1 to N and m is the size of the bit vector. The posting list of each term t can be accessed as a set by posting-list(t). k random pairs of (a_i, b_i) where $1 \leqslant i \leqslant k$ are generated.

Output: k hash tables $h_1, h_2, ..., h_k$

```
 1: for $i = 1$ to N do
 2:     for $j = 1$ to k do
 3:         $perm[i][j] \leftarrow (a_j i + b_j) mod P$
 4:     end for
 5: end for
 6: for every term $t$ in the corpus do
 7:     for $x = 1$ to k do
 8:         $h_x[t] = [min_{num \in posting-list(t)}(perm[num][x])] mod(m)$
 9:     end for
10: end for
```

3.5 Experimental Results

In the first phase of our experiment we show why k min-wise independent hash functions were chosen. Why not using a mixture of random functions and min-wise independent hash functions? Why not using less than k min-wise hash functions and repeat some of them multiple times? Because of the uncertainty of the input (i.e. the Corpus) and the complexity of calculating the average false positive error for each of these scenarios first we conduct these experiments in these different scenarios and as we will show, k min-wise independet remains one of the best alternatives in almost all of the experiments. Table 3.5 describes the details of the computation steps of our proposed hash functions, the pure random method, and the theoretic formula.

Method Name	Method descriptions
Pure Random	1) Divide the word to 4-character parts 2) Build a 32-bit integer by concatenating 8-bit code of 4 characters 3) XOR all resulting 32-bit integers 4) Use the resulting 32-bit integer as the seed of a uniform random generator 5) Generate a sequence of numbers between 0 and m-1 using the random generator 6) Use these randomly generated numbers as position of bits that should be set to 1 in the bloom filter for this word.
1-Min-Others-Random	1) $s = min\{docID \mid word \in docID\}$ 2) Use s as the seed of a uniform random generator 3) Generate a random number using this random generator between 0 to $m-1$ 4) Use the generated number as the position of the first bit that should be set to 1 in the bit-pattern of the word in the bloom filter 5) Use the pure random method for generating the position of the other bits that should be set to 1 in the bit pattern for the word in the bloom filter
1-Max-Others-Random	1) $s = max\{docID \mid word \in docID\}$ 2) Use s as the seed of a uniform random generator 3) Generate a random number using this random generator between 0 to m-1 4) Use the generated number as the position of the first bit that should be set to 1 in the bit-pattern of the word in the bloom filter 5) Use the pure random method for generating the position of the other bits that should be set to 1 in the bit pattern for the word in the bloom filter

1-Min-1-Max-Others-Random	1) $s1 = min\{docID \mid word \in docID\}$ 2) $s2 = max\{docID \mid word \in docID\}$ 3) use si as a seed for uniform random generator Ri (i=1, 2) 4) Generate a random number pi between 0 to m-1 using random generator Ri (i=1, 2) 5) Use pi as the position of the ith bit that should be set to 1 in the bit-pattern of the word in the bloom filter (i=1, 2) 6) Use the pure random method for generating the position of the other bits that should be set to 1 in the bit pattern for the word in the bloom filter
1-(Min+Max)-Others-random	1) $s1 = min\{docID \mid word \in docID\}$ 2) $s2 = max\{docID \mid word \in docID\}$ 3) $s = s1 * 256 * 256 + s2$ 4) Use s as the seed of a uniform random generator 5) Generate a random number using this random generator between 0 to m-1 6) Use the generated number as the position of the first bit that should be set to 1 in the bit-pattern of the word in the bloom filter 7) Use the pure random method for generating the position of other bits that should be set to 1 in the bit pattern for the word in the bloom filter
All-(Min+Max)	1) $s1 = min\{docID \mid word \in docID\}$ 2) $s2 = max\{docID \mid word \in docID\}$ 3) $s = s1 * 256 * 256 + s2$ 4) Use s as the seed of a uniform random generator 5) Generate a sequence of numbers between 0 and m-1 using the random generator 6) Use these randomly generated numbers as position of the bits that should be set to 1 in the bloom filter for this word

2-First-Mins-Others-Random	1) $s1 = first - min\{docID \| word \in docID\}$ 2) $s2 = second - min\{docID \| word \in docID\}$ 3) use si as a seed for uniform random generator Ri (i=1, 2) 4) Generate a random number pi between 0 to m-1 using random generator Ri (i=1, 2) 5) Use pi as the position of the ith bit that should be set to 1 in the bit-pattern of the word in the bloom filter (i=1, 2) 6) Use the pure random method for generating the position of other bits that should be set to 1 in the bit pattern for the word in the bloom filter
3-First-Mins-Others-Random	1) $s1 = first - min\{docID \| word \in docID\}$ 2) $s2 = second - min\{docID \| word \in docID\}$ 3) $s3 = third - min\{docID \| word \in docID\}$ 4) use s_i as a seed for uniform random generator $R_i(i = 1, 2, 3)$ 5) Generate a random number pi between 0 to m-1 using random generator $R_i(i = 1, 2, 3)$ 6) Use pi as the position of the ith bit that should be set to 1 in bit-pattern of the word in the bloom filter (i=1, 2, 3) 7) Use the pure random method for generating the position of the other bits that should be set to 1 in the bit pattern for the word in the bloom filter
All-K-First-Mins	1) $s_1 = first - min\{docID \| word \in docID\}$ 2) $s_2 = second - min\{docID \| word \in docID\}$ 3) ... 4) $s_k = kth - min\{docID \| word \in docID\}$ 5) use s_i as a seed for uniform random generator $R_i(i = 1, 2, ..., k)$ 6) Generate the random number pi between 0 to m-1 using random generator $R_i(i = 1, 2, ..., k)$ Use pi as the position of the ith bit that should be set to 1 in the bit-pattern of the word in the bloom filter (i=1, 2, ..., k)
Theoretic Formula	$(1 - (1 - \frac{1}{m})^{kn})^k$

In order to evaluate the effectiveness of each method in reducing the false positive error, we test them experimentally on three collections. The first corpus is a collection of Wikipedia articles [5]. This collection consists of 2000 high quality pages selected by a team of volunteers. For indexing purposes we stripped all HTML tags, Java scripts, comments and other non-related elements from the html files and removed all numbers and words shorter than 3 characters. The total size of the html files is 244 Megabytes and after cleaning the files and removing the duplicates of the words in each file the total size is reduced to 20.4 Megabytes. According to the statistics provided in [7], in Wikipedia, the average number of words per document is about 400. Based on this assumption, each document of the test collection is divided into partitions of size 400 words. After partitioning each document, the size of its last partition will be less than or equal to 400 words. To address this problem, the number of words in all fragments of each document has been balanced. For example, after partitioning a 700 word document, there will be 2 partitions of size 350 words. The output of this step is $7,500$ partitions with the average size of 350 words per document and 212568 unique terms in total. In this experiment, the false drop ratio for each of our proposed hash functions is computed using the following scheme:

```
1: for Each proposed method $M_i$ do
2:   for $\frac{m}{n} = 2$ to 10 do
3:     for $k = 1$ to 6 do
4:       Create a signature file index given $\frac{m}{n}$ ratio and $k$ and use the $M_i$ for hashing
         words to bit patterns
5:       Compute the average false drop ratio of all bloom filters in the created index by
         using all the unique terms which were used in our collection
6:     end for
7:   end for
8: end for
```

Using the above scheme, for each method which mentions in 3.5 and every $\frac{m}{n}$ and every k, 54 average false drop ratios were computed. Figures 3.3 3.4, 3.5, 3.6, and 3.7, show the comparison of false drop ratio of our proposed methods for $\frac{m}{n}$ ratios of $2, 4, 6, 8$ and 10 respectively. In all of the figures the x-axis corresponds to different experiments with different hash functions and the y-axis shows the average false positive error of that experiment.

From these experiments we can point out the following key observations:

- In all of the diagrams, for all the values of $\frac{m}{n}$ and k, the false drop ratio of pure random method is just slightly more than that of the theoretic formula. It confirms

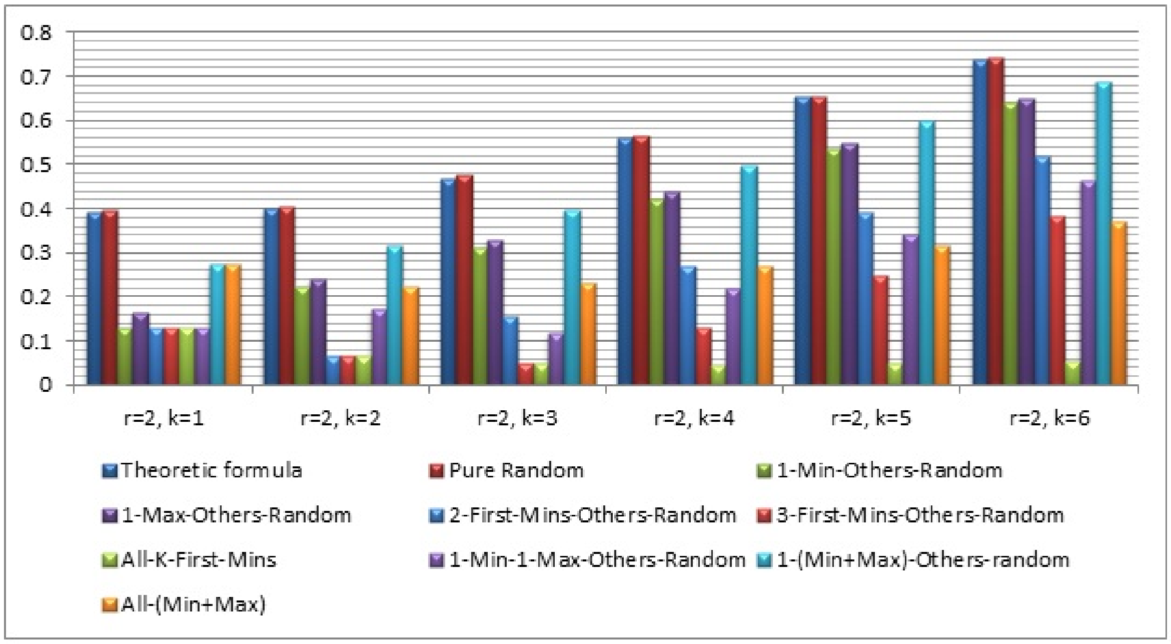


Figure 3.3: The comparison of the false drop ratio of the proposed method when $\frac{m}{n}$ (r) is equal to 2

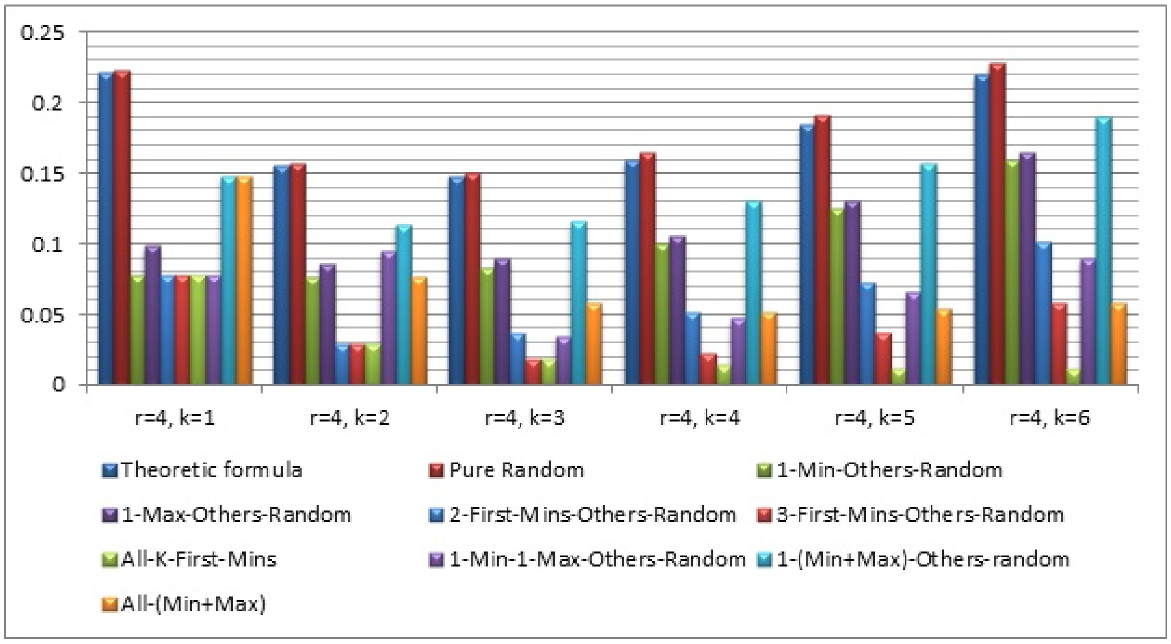


Figure 3.4: The comparison of the false drop ratio of the proposed method when $\frac{m}{n}$ (r) is equal to 4

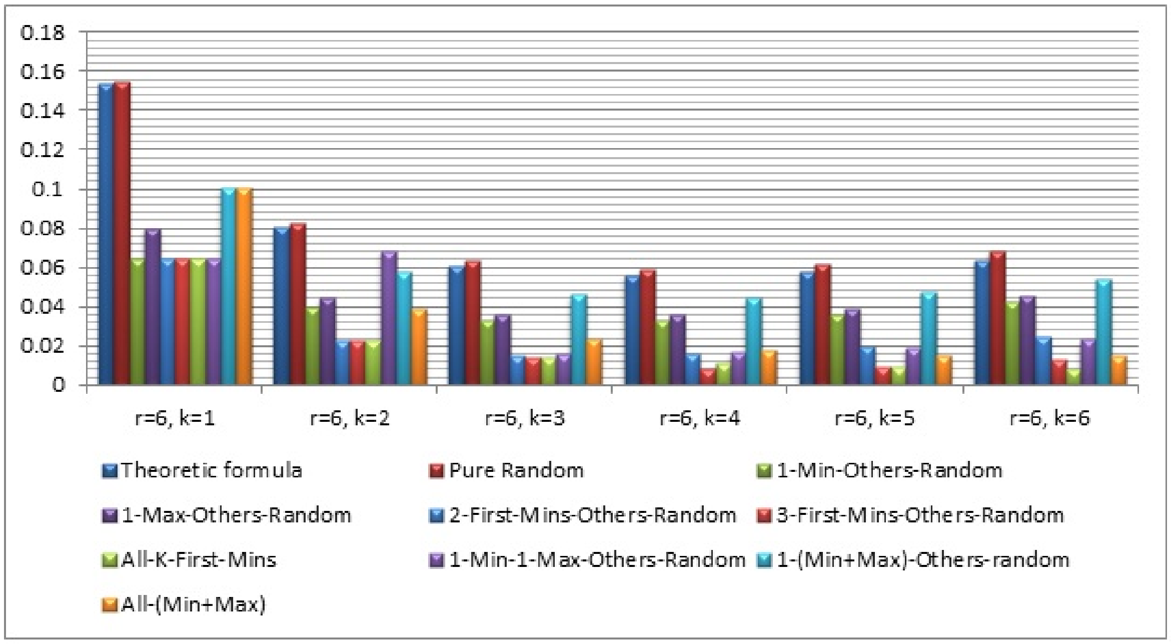


Figure 3.5: The comparison of the false drop ratio of the proposed method when $\frac{m}{n}$ (r) is equal to 6

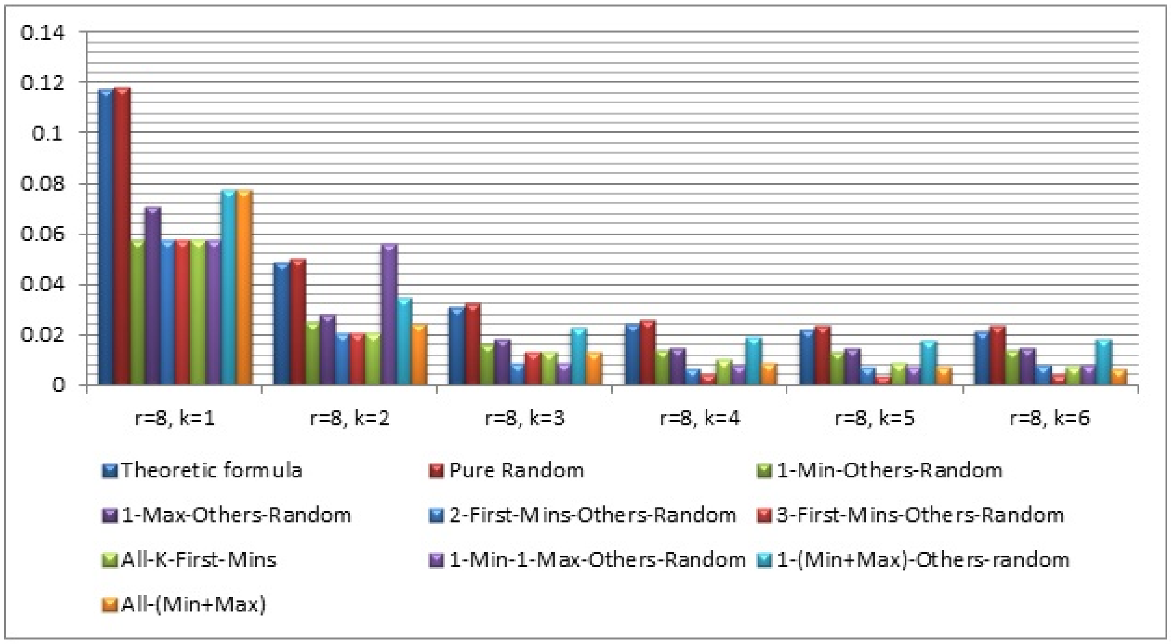


Figure 3.6: The comparison of the false drop ratio of the proposed method when $\frac{m}{n}$ (r) is equal to 8

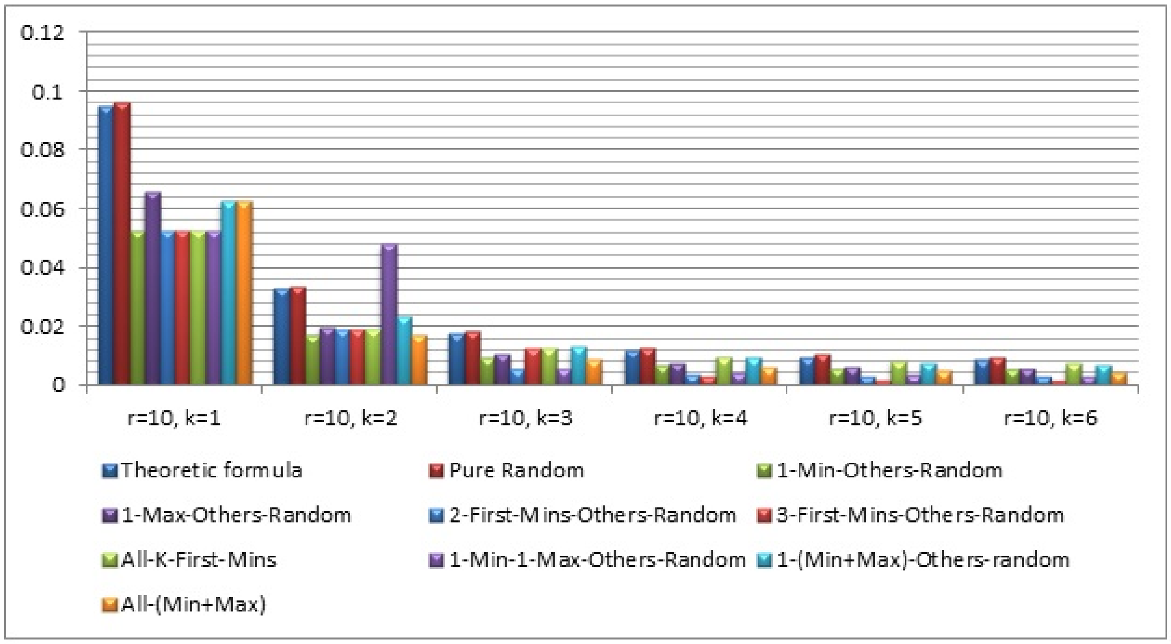


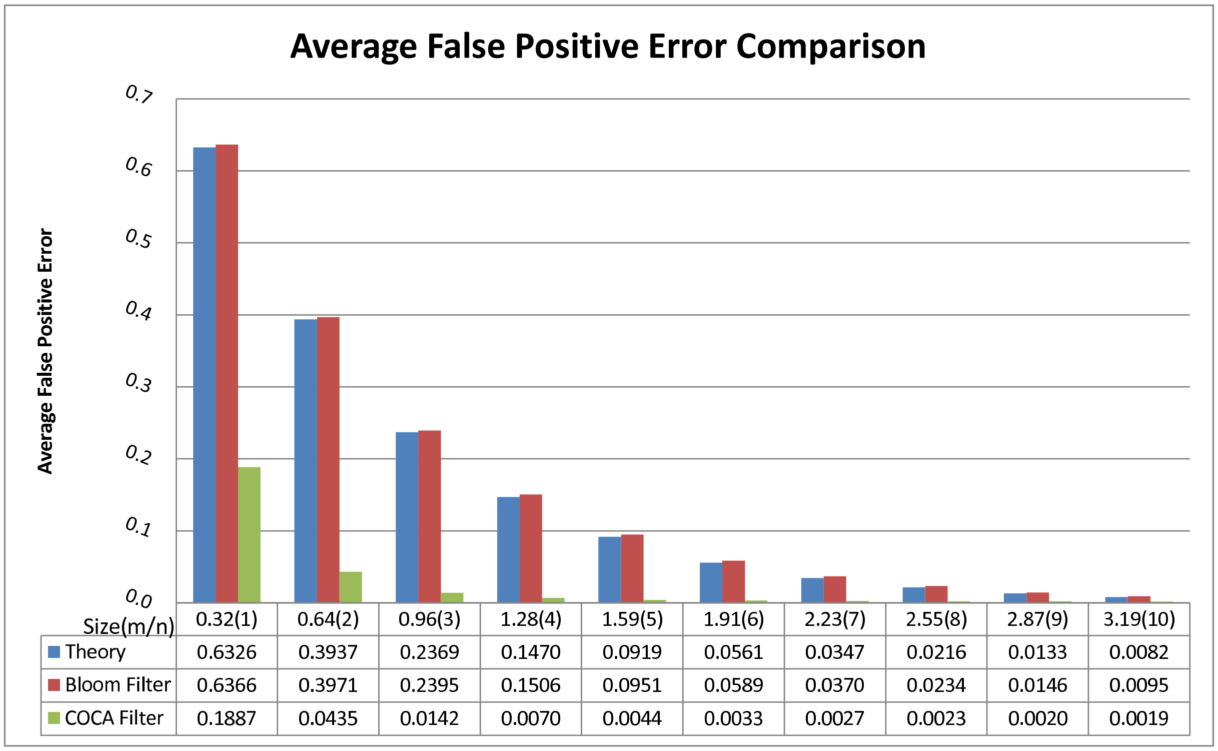
Figure 3.7: The comparison of the false drop ratio of the proposed method when $\frac{m}{n}$ (r) is equal to 10

the argument of Bose in [16] that blooms formula provides only a lower bound for false drop probability.

- In all of the diagrams, for all values of $\frac{m}{n}$ and k, the false drop ratio in all of the proposed methods (except the ALL-(Min+Max) method) is significantly better than the false drop ratio in Pure Random method and theoretic formula.
- In some cases we have up to 21 times improvement in false drop ratio. For example, in r=4, k= 6, false drop ratio of All-K-First-Mins method is 21 times better than that of Pure Random method.
- Only in One-Min-1-Max-Others-Random method, and just for a few combinations of m-over-n and k, the false drop ratio is getting slightly worse than Pure Random method.
- In our proposed methods, the optimum k for each $\frac{m}{n}$ differs from the optimum k in Pure Random method. Since in these Proximity-Aware bloom filters we are using more similar bit patterns for the terms of each document, the throughput (number of 1 bits in the bloom filter) of these bloom filters is less than the throughput of Pure Random method. On the other hand, we know that for maximizing the information

content of each bloom filter the throughput must be 50%. So we can conclude that in Proximity-Aware bloom filters, for minimizing the false drop ratio, we need more hash functions than what was suggested by Bloom in [15].

- Among our proposed methods, in most of the cases, All-K-First-Mins (i.e. COCA filter) outperforms the others. This method gives the same hash values to the words that their first K minimum occurrences are the same as explained in 1 Algorithm. From this point whenever we are talking about COCA filters we are referring to this particular case.



Size(m/n)	0.32(1)	0.64(2)	0.96(3)	1.28(4)	1.59(5)	1.91(6)	2.23(7)	2.55(8)	2.87(9)	3.19(10)
🟦 Theory	0.6326	0.3937	0.2369	0.1470	0.0919	0.0561	0.0347	0.0216	0.0133	0.0082
🟥 Bloom Filter	0.6366	0.3971	0.2395	0.1506	0.0951	0.0589	0.0370	0.0234	0.0146	0.0095
🟩 COCA Filter	0.1887	0.0435	0.0142	0.0070	0.0044	0.0033	0.0027	0.0023	0.0020	0.0019

Figure 3.8: The comparison of the average false positive error of the COCA filter method with the conventional Bloom filter and the theoretic formula for the sampled Wikipedia corpus. The x-axis shows the index size in Megabytes and the value in the parentheses indicates the $\frac{m}{n}$ ratio.

In the second phase of our experiment the goal is to compare the average false positive error of the COCA filter (i.e. All-k-first-mins case) with that of the theoretic formula and the conventional Bloom filters. Let W be the set of all words. $FP(d)$ is defined as the number of words in W which are not in document d but its corresponding Bloom filter falsely claims that they are. For each document, the false positive error of its corresponding Bloom filter is defined as $FPE(d) = \frac{FP(d)}{|W|}$. Thus, the average false positive error of a signature file method is $\frac{\sum_{d \in D} FPE(d)}{|D|}$.

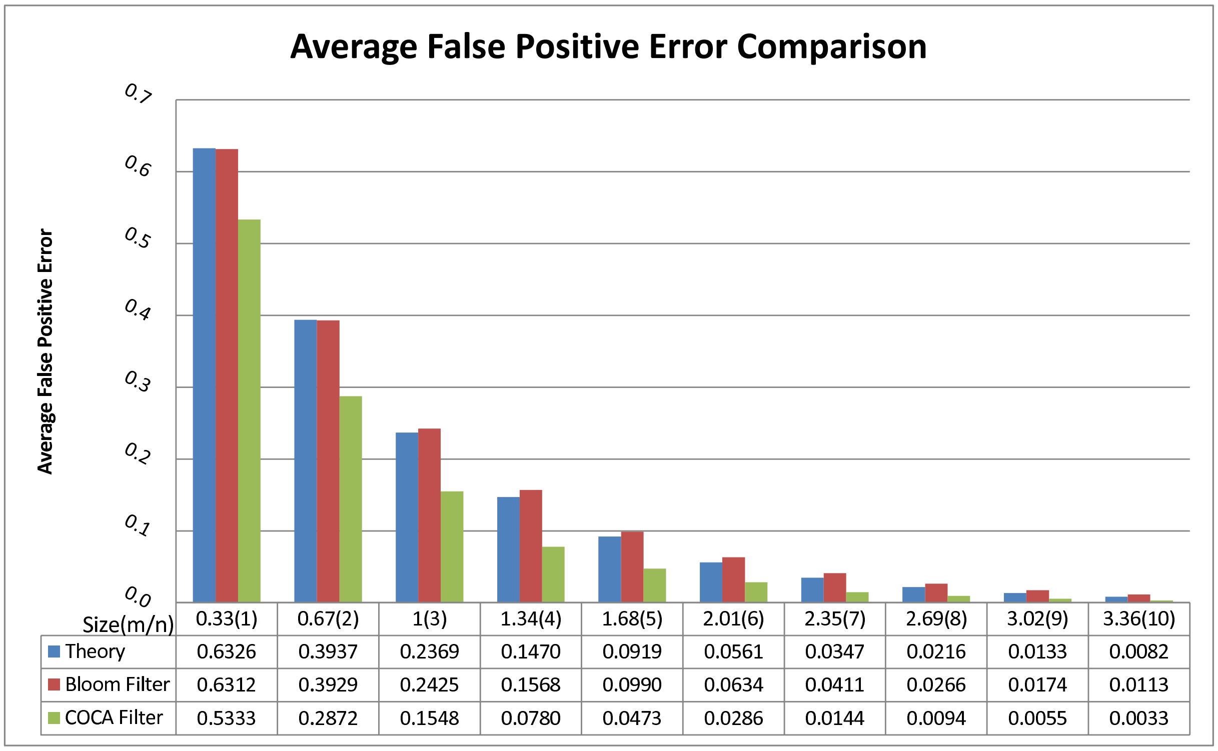
In figure 3.8 , the x-axis shows the result of this experiment for different $\frac{m}{n}$ ratios and y-axis shows the corresponding average false positive error. In each method, for each $\frac{m}{n}$ ratio, only the result for the k value which minimizes the false positive error is shown. The total size of the signature file along with the average false positive error is also included in the table below the figure.

Note that in some cases there is about 21 times improvement in the average false positive error. For example, in $\frac{m}{n} = 6$, the average false positive error of the COCA filter is about 21 times better than the Bloom filter. In some cases there is up to a 75% reduction in the size of the index for the same average false positive error. For example if the objective is to achieve an average false positive error less than 0.19 in conventional Bloom filters the $\frac{m}{n}$ ratio should be at least 4 while in COCA filter with the $\frac{m}{n}$ ratio of 1 the average false positive error of 0.19 is achievable. In other words for every bit that is used in the COCA filter, 3 extra bits are required in a conventional Bloom filter. Note that when $\frac{m}{n} = 1$ the index size is only 1.6% of the polished corpus.

Our proposed approach is based on the co-occurrence of the words in documents and therefore is sensitive to the correlation of documents. In order to investigate the relationship between the degree of correlation among documents and the improvements in average false positive error in COCA filters, the previous experiment was repeated over a collection of weakly-correlated web pages. This collection is a random selection of $900,000$ web pages released by Google in 2002 for a programming contest [4]. We chose about 13500 samples from this collection randomly and performed the previous experiment on the resulting collection. We used the same method as the first experiment for cleaning the documents and fragmenting them. Due to the smaller average size of documents in this collection, we divided the documents into partitions of size 100. After partitioning, the collection had about 35200 documents with the average size of 80 terms and 228715 terms in total.

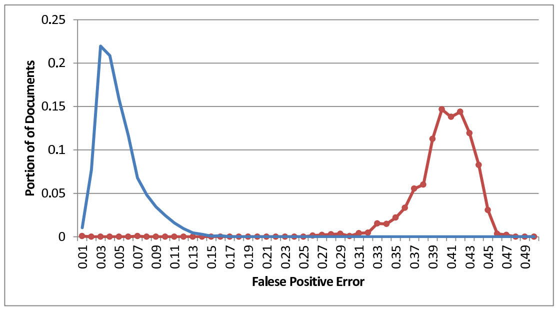
Figure 2 shows the result of this experiment. Although COCA Filter is still better than conventional Bloom Filter, the improvement in this experiment is not as good as the first experiment. The result of this experiments confirms the sensitivity of our proposed method to the correlation among the terms of the documents.

In the first experiment we chose a collection of high quality articles of Wikipedia which are all coherent in writing and have a scientific theme. It is normal to encounter many synonyms of a word instead of a repetition and there is also a somewhat predictable set of antonyms to follow. On the contrary, in the second corpus documents are not coherent neither in meaning nor in the style which results to have lots of random terms

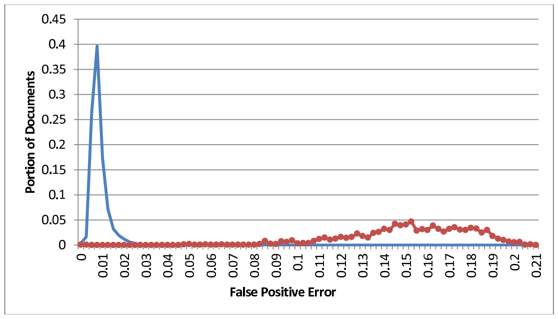


Size(m/n)	0.33(1)	0.67(2)	1(3)	1.34(4)	1.68(5)	2.01(6)	2.35(7)	2.69(8)	3.02(9)	3.36(10)
🟦 Theory	0.6326	0.3937	0.2369	0.1470	0.0919	0.0561	0.0347	0.0216	0.0133	0.0082
🟥 Bloom Filter	0.6312	0.3929	0.2425	0.1568	0.0990	0.0634	0.0411	0.0266	0.0174	0.0113
🟩 COCA Filter	0.5333	0.2872	0.1548	0.0780	0.0473	0.0286	0.0144	0.0094	0.0055	0.0033

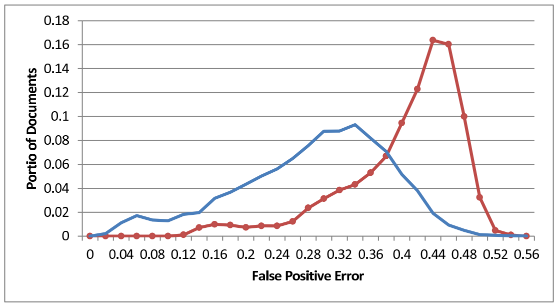
Figure 3.9: The comparison of the average false positive error of the COCA filter method with the conventional Bloom filter and the theoretic formula for the sampled Google corpus. The x-axis shows the index size in Megabyte and the value in the parentheses indicates the $\frac{m}{n}$ ratio.



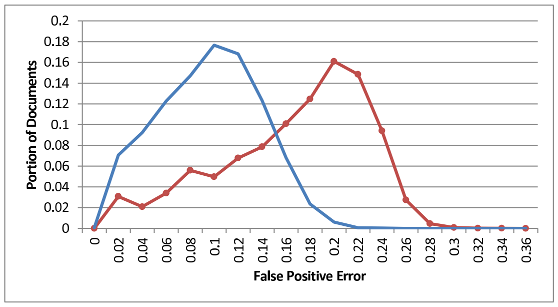
(a) Distribution of false positive error for $\frac{m}{n} = 2$ for the Wikipedia collection



(b) Distribution of false positive error for $\frac{m}{n} = 4$ for the Wikipedia collection



(c) Distribution of false positive error for $\frac{m}{n} = 6$ for the Google collection



(d) Distribution of false positive error for $\frac{m}{n} = 6$ for the Google collection

Figure 3.10: Comparison of the distribution of the false positive error of the COCA Filter and the conventional Bloom filter. In each graph the left curve corresponds to the COCA filter and the right curve corresponds to the conventional Bloom filter.

from street names and addresses to gene sequences and peoples' names in them. Moreover the diversity of the topics that these terms are covering is higher than the first collection and this diversity reduces the probability of the co-occurrence of the words in documents and consequently reduces the effectiveness of our proposed method.

To ensure that the size of corpus does not have a negative effect on the quality of COCA Filters we repeated the first experiment on two larger corpora. The first one is a similar but larger collection of Wikipedia documents [6]. This collection is a more comprehensive version of the first collection and consists of 6500 high quality pages selected by a team of volunteers for school students. The size of this collection is more than 3 times the size of the first collection but it is very similar to the first collection in terms of coherency and writing style. We used the same method as the first experiment for cleaning the documents and fragmenting them. After partitioning, the collection had 21543 documents with the average size of 350 words per document and 321500 unique terms in total. Table

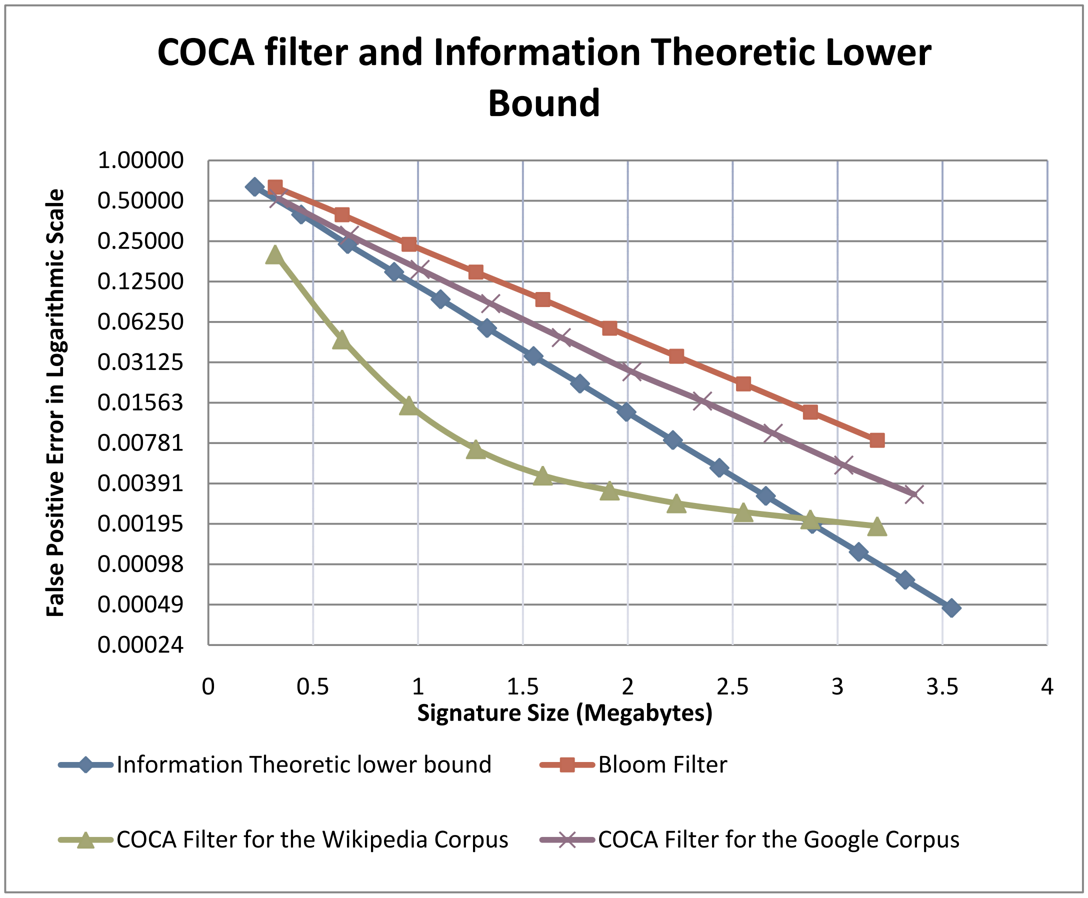


Figure 3.11: The Comparison of Information Theoretic lower bound with COCA filters over two different corpora

Table 3.2: Comparison of the average false positive error of COCA filter over two wikipedia corpora with different sizes

m/n	1	2	3	4	5	6	7	8	9	10
Wikipedia(6500)	0.18913	0.04369	0.01193	0.00476	0.00264	0.00179	0.00142	0.00116	0.00099	0.00087
Wikipedia(2000)	0.18869	0.04346	0.01415	0.00703	0.00439	0.00331	0.00272	0.00234	0.00204	0.00185

1 compares the result of this experiment with the result of the first experiment. It shows that increasing the size of the collection does not increase the average false positive error given that the coherency and style of the corpus remains the same. It confirms that the higher average false positive error of COCA Filters on Google collection is not because of the larger size of this collection and it is only due to the non-coherent, random nature and diversity of that collection.

In the second experiment we used the BioMed Central's open access full-text corpus [8] with 62,347 articles and 5GB of data. Our first reason for choosing this collection was its suitable size. Another reason for selecting this collection was testing a collection that its content and structure is not the same as those of Wikipedia. After eliminating the non-related contents from the documents, the size of our collection reduced to about 1.5GB. To make our comparison to the previous collection as fair as possible, we used the same fragment size as the previous experiment. After fragmenting the collection we had about 511,300 fragments. This is about 70 times larger than the first collection. Since the run-time of the experiment on a large collection of documents is huge, we decided to compute just the false drop ratio of COCA filters with $\frac{m}{n} = 2$ and k=2, 3 and 4. The reason for choosing this method is that this method outperformed all the other proposed methods in the previous experiment. Figure 3.5 shows the result of this experiment.

One area of concern is whether in COCA Filters the average false positive error decreases at the cost of having many bloom filters with low false positive error and many with high false positive error (i.e. having a bimodal distribution). Figure 3.103 illustrates a comparison between the distribution of the false positive error in the COCA Filter and the conventional Bloom filter for $\frac{m}{n} = 2$, 4 over the Wikipedia and Google corpora. In each graph the left curve corresponds to the COCA filter and the right curve corresponds to the conventional Bloom filter. In all graphs, in both curves, by increasing the distance from the average, the frequency of documents decreases rapidly. It shows that there are only a few documents with a false positive error significantly greater than (or smaller than) the average false positive error. It can be seen that in the Wikipedia corpus which has higher correlation even the worst false positive error of the COCA filter method is significantly better than the best false positive error of the conventional Bloom filter method while in

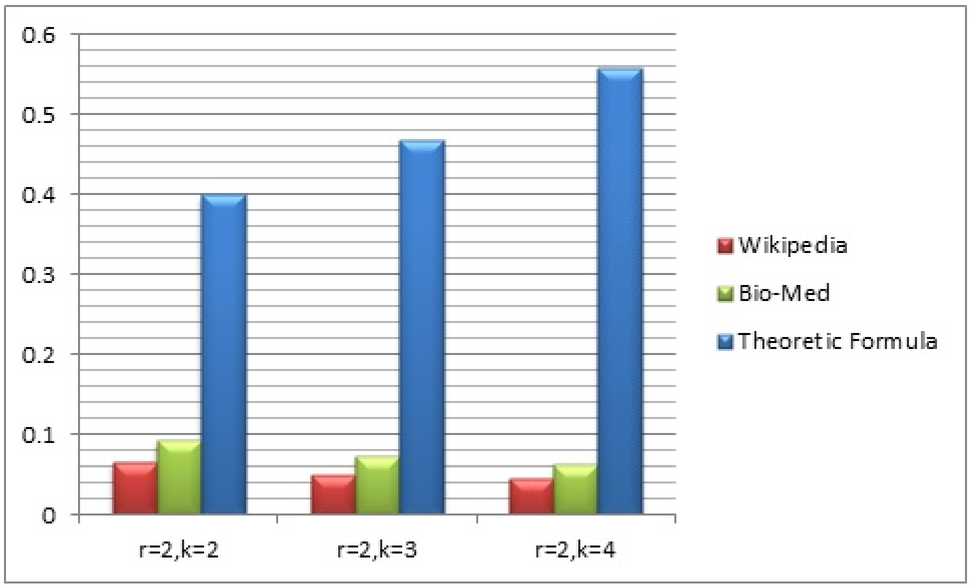


Figure 3.12: The comparison of the average false positive ratio of the COCA filter method when $\frac{m}{n}$ (r) is equal to 2 for BioMed and Wikipedia corpora and the theoretic formula

the Google corpus this property does not hold. Moreover, in the Wikipedia corpus the deviation of the COCA filter curve from the average is much smaller than its corresponding Bloom filter curve while in the other corpus this is not easily observable.

Another interesting comparison is between the COCA filter and the information theoretic lower bound on the three corpora as suggested in [17]. In other words we want to compare our method in terms of space efficiency with the best possible randomized data structure which does not utilize the co-occurrence probability of the words. Figure 3.11 illustrates this comparison. Note that the y-axis is the average false positive error in logarithmic scale in order to demonstrate the difference more clearly. While the COCA filter for the Google corpus never beats the information theoretic lower bound, the COCA filter for the Wikipedia corpus beats it in most of the cases by a significant margin. Note that as the false positive error gets closer to zero the distance between the curves shows a smaller difference. Interestingly as the correlation among the terms of the corpus gets stronger the rate of the decrease in false positive error tends to be hyper-exponential (as in Wikipedia corpus) rather than exponential (in Google Corpus) but as the index size increases the improvement rate decreases until it becomes very close to Bloom filter. This shows that for these applications where the elements of the corpus are highly correlated, utilizing the extra information about this correlation can be very valuable.

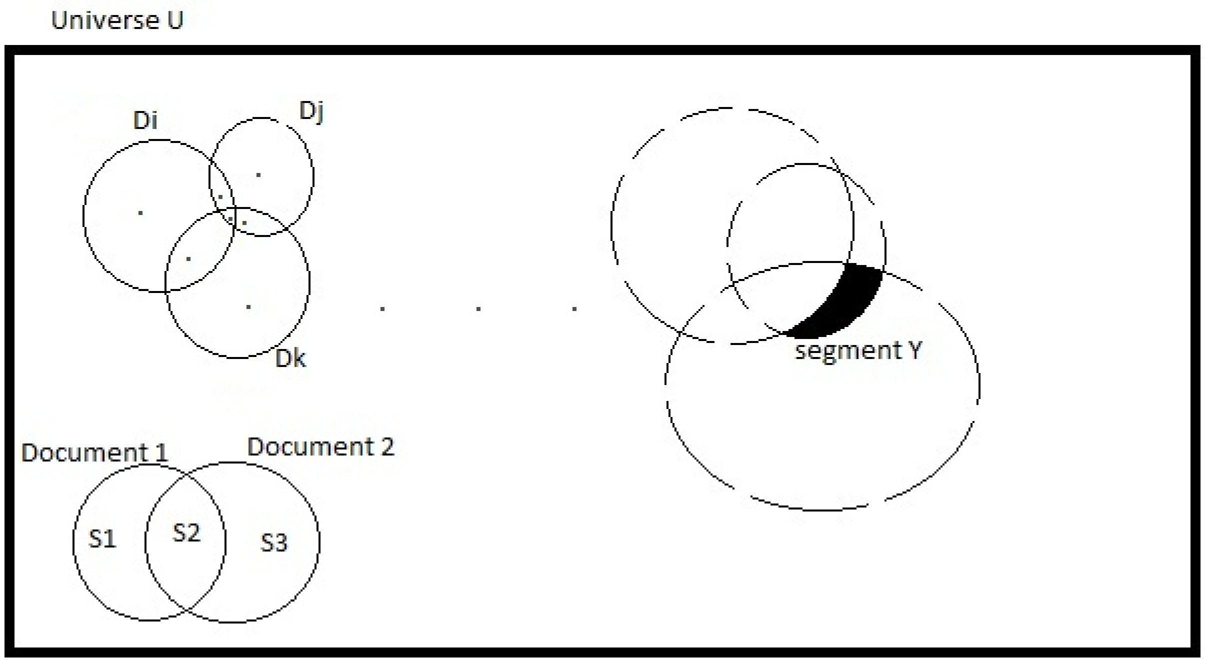


Figure 3.13: An illustration for the problem of perfect COCA filters where each circle represents a set and dots represent an element from the universe and each intersection of two circles has a unique element which can be identified with.

3.6 Some thoughts on perfect COCA filters

In this section we formulate the case that the false positive error is zero in the same setting. In other words given a set of sets with members from the universe U, we want to devise a boolean vector of size m for every member of U, such that the bit vector of every set which is produced by superimposing every member of the set can determine if every member of U is in the set or not with 100% accuracy.

Figure 3.13 provides a good illustration for this problem. Note that every "segment" has at least one member (word) which uniquely identifies that segment (i.e. we can not have more sets such that their intersection contains that unique word). If in any segment there is more than one element which uniquely identifies that segment we can consider all of those elements as one member with one unique bit-vector of size m because those elements are occurring together all the time and there is no need to have different bit patterns for each of them. Furthermore we claim that each segment should have a unique bit-vector. Proof: Assume toward contradiction that two different segments have the same bit-vector. By definition of segment, one of these segments must belong to a set which is not the superset of the other segment. So all the members of this segment while are not occurring in the other superset are conforming to the bit vector of that superset which means the accuracy is not 100% and is a contradiction. So every segment must have a unique bit-vector.

Since each segment should have a unique bit-vector, we need at least $(\lg(\#of segments))$ bits for every member of the universe U. On the other hand each segment requires at least one member which means we need to have at least $(\lg U)$ bits in every bit vector. This lower bound for English corpora or more generally human readable texts is very small since the number of different terms is very small. For example the number of all the different words and slangs and expressions and special sings in the English language is no more than one million and if you consider all the different numbers and sings in the whole wikipedia the number is still less than a billion. This means that for a perfect COCA filter this lower bound is at most $(\lg 10^9)$ which is less than 30 bits.

The other important constraint in this setting is that for every segment and every set that the segment does not belong to, the bit pattern of that segment should not conform to the bit pattern of that set. For example in the figure 3.13 if the bit-vectors corresponding to the two segments of the $S1$ and $S2$ are named $X1$, $X2$ the following two constraints are held between segment y and those two segments.

$$y.(X1 + X2) < y.y$$

and

$$y.(X1 + X2) < (X1 + X2).(X1 + X2)$$

Where "+" is the boolean summation and "." is the dot product of the vectors. Note that we have at most $(number of segments) * (number of documents)$ of these constraints and having an assignment for the bit-vectors of all the segments such that all these constraints are satisfied is necessary and enough in order to have a perfect COCA filter.

Assuming that each segment S_i has a boolean bit vector of size m with name of y_i and each Document D_j has a corresponding bit vector of size m with name of y_{D_j} here we formulate all the constraints:

$$\{\forall D_j, \forall S_i : S_i \subseteq D_j | y_{D_j}.y_i = y_i.y_i\}$$
$$\{\forall D_j, \forall S_i : S_i \cap D_j = \phi | y_{D_j}.y_i < y_i.y_i\}$$

This is a boolean semi-definite programming problem with the objective of finding the minimum value of m such that there exists an answer with the mentioned constraints. While there exists polynomial solutions for the gerneral problem the boolean version is NP-hard. there has been studies on how to solve this problem approximately. One avenue for future research is how to solve this problem approximately to improve the average false positive of bloom filters.

3.7 Conclusion and Future Work

In this chapter the problem of false positive error of Bloom filters has been addressed and a novel technique to reduce the false positive error is proposed. The effectiveness of this approach was evaluated by conducting two experiments and our experimental results showed that up to 21.6 times improvement in false positive error or equivalently up to 75% reduction in space is achievable. Although this improvement is surprisingly good it is important to note that this technique is very sensitive to the correlation among the terms of the documents in the set.

In the current definition of the similarity function the size of each posting list can not affect the similarity of any two words as long as the ratio of the intersection and the union of their corresponding posting list is the same. It would be interesting to investigate similarity functions which are sensitive to the size of the posting lists as well.

Finding the information theoretic lower bound for the minimum number of bits required for a Bloom filter, given the extra information of the co-occurrence probability of each pairs of the members of the universe is another avenue for research. The formulations in the previous section might be an starting point to achieve this goal.

TCAM has been used to replicate a set of Bloom filters in order to solve the subset query problem for small sets [32]. Another potential opportunity is to explore the possible positive effect of COCA filters in areas where TCAM can be used as a group of Bloom filters.

Chapter 4

TCAM and Network Flow Classification

4.1 Overview

Since TCAM is a prevalent device in networking equipments, taking advantage of it in computer-networks related problems is not adding extra cost and potentially can be very useful. In this chapter we focus on one of these problems namely traffic classification problem.

Real time traffic classification is one of the fundamental problems in computer networks and has been studied thoroughly. As encryption techniques are getting more common across different network protocols, there is an increasing trend towards classification techniques that are based on statistical patterns of different network protocols. One area of study is how to increase the speed of these classifiers at the cost of reducing their precision or recall. In this chapter we will introduce a novel approach for real time Internet flow classification using TCAM. This approach is based on the work of Shinde et al. [51] which solves the approximate nearest neighbor search problem using TCAM. While this technique is not beating the best machine learning classifiers in terms of f-factor, the low computational cost of this approach makes it practical for network flow classification with order of magnitude higher throughput.

4.2 Introduction

Real time traffic classification is one of the major and fundamental problems that Internet service providers (ISPs) and network equipment vendors are faced with today. Traffic classification is used mainly with two broad objectives of Quality of Service (i.e. QoS) and Security. If network flows can be classified in real time, the classification information can be used to guarantee a threshold of quality(i.e. time delay, bandwidth, etc.) over a range of certain types of protocols (traffic shaping). Furthermore this information can be used as a core part of intrusion detection systems [3], in anti-virus and anti-worm applications and to detect patterns indicative of denial of service attacks or for eavesdropping over particular data channels.

In order to define traffic classification, first we need to define *network flow*. A stream of IP packets with the same 5-tuple of

- protocol type(TCP or UDP)
- source IP number
- source port number
- destination IP number
- destination port number

is considered to be a network flow. Determining which classes of applications are communicating over this flow is the objective of traffic classification. One basic heuristic for traffic classification is to guess the controlling application based on the port number of one of the end points. However, as many applications are using unpredictable port numbers [35], this technique is getting inaccurate.

Consequently, more sophisticated techniques look for application specific data in the packets of a flow. These techniques generally are called deep packet inspection (DPI) techniques. There are two implicit assumptions in DPI. Firstly, the payload of a flow are visible to third parties in a decrypted way and secondly the protocols that the two applications are using to communicate with each other is public knowledge. Assuming that these assumptions remain true, having every packet of a flow checked for a particular signature or regular expression makes DPI techniques very costly. Moreover using random ports and encrypting the payload or applying obfuscation techniques as is used in peer to peer protocols, makes it almost impossible to classify the network flow accurately.

Hence there is a strong trend on classification schemes which are based on statistical patterns in general features of the traffic such as inter-packet arrival times and packet lengths. There are many researches based on these attributes with different machine learning techniques. A detailed survey on different machine learning techniques and their performances can be found in [44]. In the next section we will briefly go over the different machine learning techniques which are used for traffic classification and describe a few heuristic approaches.

The rest of this chapter is organized as follows: Section 4.3 reviews related work and provides a short survey on machine learning techniques used in order to classify networks. Section 4.3.1 discusses the best features and classification techniques. Section 4.4 describes the details of our approach and the proposed methods. Section 4.5 presents the evaluation methods and the experimental results and finally we conclude our work and disscuss future directions in Section 4.6.

4.3 Machine Learning Techniques for Traffic Classification

Many papers recently showed that using machine learning techniques can be a practical approach for traffic classification mainly because they only require to inspect the first few packets of the flow and therefore they can be used in traffic shaping applications. In each machine learning algorithm first we need to define a set of features which will be used to differentiate different network protocols. Not including the features related to the content of the packets, these features are usually some combination of the followings: inter-packet arrival time, minimum and maximum length of packet and flow duration. Once the features are defined, different classifiers are used over a training set to train the classifier. A testing set is used to compute the precision and recall of each classifier. The main metrics which are used to compare different classifiers to each other are false positive, false negative, true positive, true negative, precision and recall.

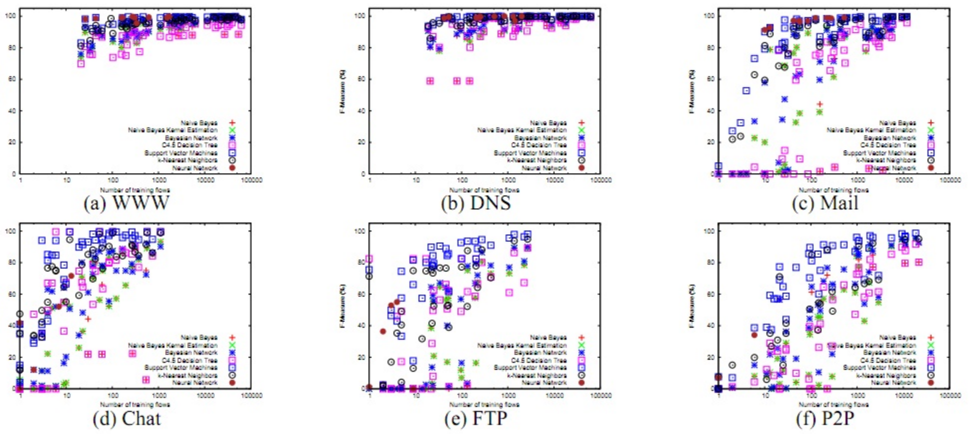


Figure 4.1: Comparison of 7 classifiers over 6 classes of protocols with different training sizes in terms of f-factor. The figure is from [37]

Extensive work was done in [44] to compare different classifiers in terms of their accuracy, recall and f-factor. Figure 4.1 from [37] illustrates the differences between the most common machine learning techniques in terms of f-factor. Note that support vector machines are one of the best techniques in this comparison which are followed by k-nearest neighbor and neural networks. As mentioned in [44], k-nearest neighbor search is too slow and hence can not be applied in practice and neural networks could not be trained for more than 10000 flows as they did not converge to a stable solution. One rational approach is to implement the SVM method using special hardware in order to do classification in high throughput. While there are works toward implementing the SVM method using special hardwares like field-programmable gate array (FPGA) circuits[10] to the best of our knowledge there has not been successful implementation of SVM for flow classification in practice. Since TCAM is prevalent in many network equipments (mainly for packet forwarding purposes) utilizing TCAM in order to classify network flows can be useful in this area because there is no need to provide extra new hardware.

4.3.1 Features for Network Flow Classification

One of the basic ad-hoc approaches in flow classification is using the port number to predict the protocol. As network applications are getting more diverse in their behavior, the port number alone is a fairly weak feature for classification.

Another important feature is the packet length of all the packets in a flow or some function of that, similar to mean or variance of the packet lengths. If we use all the packet lengths then the classifier implicitly requires us to pass all the packets first in order to classify the flow. This approach can not be used for security or QoS purposes and can be applied only for passively monitoring network flows. As Bernaille et al. suggested in [14] the first few packets after the TCP handshake are a good indicator of the network flow since these first packets are the negotiating phase of each protocol and are different among different protocols. So using the size of the first few packets can be useful in order to do an early detection of the network flow.

Another set of features are related to the time it takes for a flow to finish or the inter-arrival time between consequent packets in a flow. Because of the many routers and access points that each packet should traverse in order to get to the other end point this number can vary from time to time among different end points and can not be reliable in general.

Another set of features are related to the number and time of network flows that are initiated toward or from a particular end point. These numbers along port numbers can be used in a heuristic way in order to predict the behavior of the end point based on similar end points which were classified before. [36] uses a variation of this technique.

Another interesting technique is based on building the graph of all the peers which are connected to each other and try to find a sub-graph with a particular pattern and use this

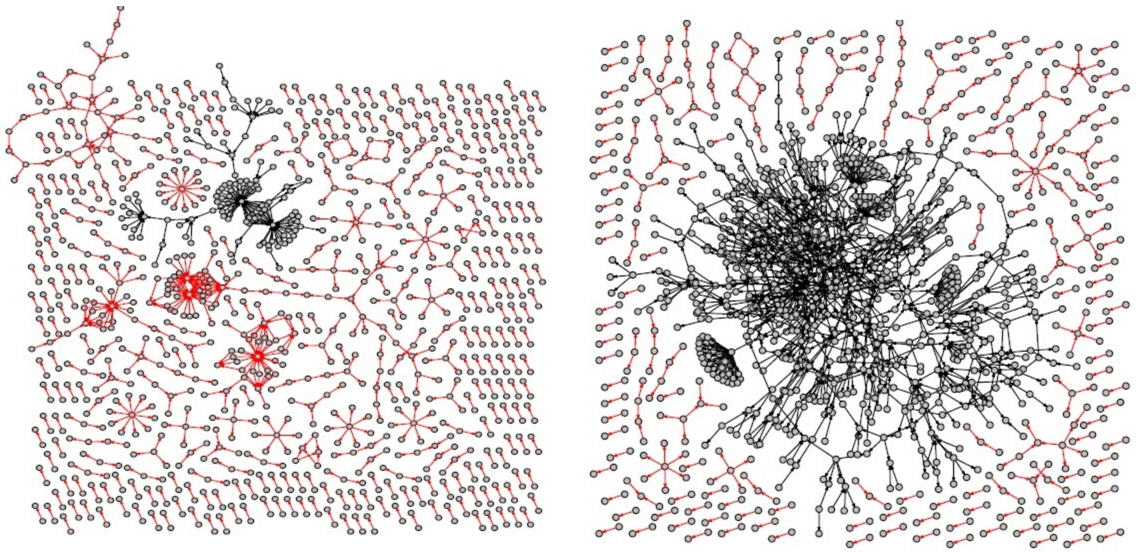


Figure 4.2: Two directed graphs related to two different network protocols. The left one is the graph of HTTPS connections between servers and their clients and the right one is the graph of connections which were established with FastTrack protocol which is used for peer to peer file sharing applications. Both these graphs were built from the viewpoint of a backbone. The figure is from [34].

information to predict the protocol. This idea was suggested by Iliofotou et al. in [34] in order to distinguish file sharing protocols among the rest of protocols. Figure 4.2 shows two samples of these graphs.

As Bernaille et al. suggested in [14] the first few packets are playing a key role in flow classification and as was mentioned by Este et al. in [26] packet sizes are among the few features which can be used in a scalable way for the network flow classification on the fly. [26] showed that using the size of the first three packets of each bi-directional flow and using the support vector machine they can receive very promising results. In the next section we will discuss the possibility of using TCAM in order to solve the approximate nearest neighbor problem and equivalently using this technique to classify network flows based on their first few packet sizes.

4.4 Using TCAM to Solve Approximate Nearest Neighbor Search

In [51] Shinde et al. introduced a novel approach to solve the approximate nearest neighbor search problem using TCAM. Interestingly their approach is based on a variation of locality sensitive hash functions (LSH) [9] which are used to solve c-Approximate Nearest Neighbor Problem(c-ANNS). First similar to LSH they define a set of ternary locality sensitive hash functions which hashes points to ternary numbers such that if two numbers are close enough then with "high" probability they are T-equal to each other(i.e. bit patterns conform to each other).

We briefly explain a formal definition of the problem as stated in [51] and their technique to solve it and then discuss its properties in order to solve our own problem namely network flow classification problem.

Definition 1. c-Approximate nearest neighbor search problem: Given a set of n points from $\mathbb{R}^d$ design a data structure such that given a query point q will return a point which is at most c times further from the closest point to q.

Note that in a classification problem we want to solve the decision problem to check whether a given query point is close *enough* to one point in a set of points and if yes we want one of those close points. So we require to solve the following problem:

Definition 2. (l,c)-near neighbor search problem: Given a set of n points from $\mathbb{R}^d$ design a data structure such that if for a query point q there is point with a distance less than l returns "YES" and a point which has a distance less than cl and otherwise returns "NO".

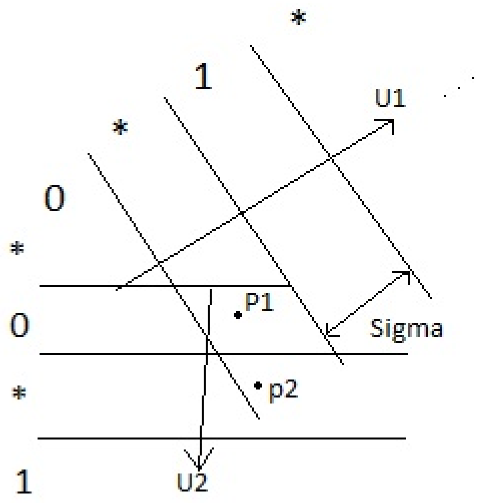


Figure 4.3: An illustration of how TLSH function works with two bits. U_1 and U_2 are two random vectors and hash two points p_1 and p_2 to *0 and ** respectively

The TLSH family is defined by a set of random hyperplanes in $\mathbb{R}^d$. Each hyperplane repeats in the space parallely in some fixed distances, say δ, and partitions the space into alternate regions and hashes points in the regions alternately to 0,*,1,*,0,*,1,*... as shown in the figure 4.3. So having w random hyperplanes, each point can be mapped to a w-bit ternary value. In [51] they showed that the (1,c)-NN problem can be solved by having

$$w = O((\frac{1}{\epsilon} log \frac{n}{\epsilon})^{\frac{c^2}{c^2-1}}((log(\frac{1}{\epsilon}) log \frac{n}{\epsilon})^{\frac{3}{2}})(c^2-1)^{\frac{-3}{2}})$$

with the error probability at most ϵ and using only one TCAM look up. Notice that here again the assumption is that the input data and query points are given randomly but as we describe in the next section this is not the case for our special problem. As explained in the previous section the size of the first three packets after the TCP hand shake is the signature for each flow. Analogous to other classification problems we have two sets in order to train our classifier (finding the optimum δ with respect to precision or recall or some other criteria) and testing our classifier. Each of these data sets are just the size of the first three packets of the flow after the TCP hand-shake along with the corresponding protocol(as an ID). By using 144 random vectors we hashed three quarters of unique members of the training set in the TCAM and test them against the other quarter in order to find optimal δ values with respect to precision and recall of every single protocol. Finally we test our classifier against the testing set in order to have an estimate for the precision and recall of this technique in practice. One of the good characteristics of this technique is that every point query does not require extensive computation and a single TCAM query can be executed for a large number of network flows in parallel. In the next section first we describe the data sets that were used for testing our classifier and then discuss the experimental results.

4.5 Data Description and Experimental Results

We did our experiments over a random subset of one of the data sets used in [26] in order to illustrate the accuracy and recall of TLSH in TCAM as a classifier. The data set is called the UNIBS data set. The packet traces were collected from the router of the faculty network of the University of Brescia. In this trace they captured the first 400 bytes of every packet and with pattern matching techniques and manual inspection they classified all the TCP flows. Hence the set can be considered relatively reliable.

The data was collected from about a thousand workstations with different operating systems and has a relatively diverse set of protocols. The training and the evaluation sets have protocol classes belonging to different network applications namely web browsing (http), mail services (SMPT), P2P(torrent) and file transferring (ftp) and IM (msn). These network applications can be considered the majority of the Internet traffic and because of their variety they are suitable to compare different classification techniques.

Protocol	Flow	bytes	payload bytes
http	65.35%	79.55%	79.89%
smpt	5.92%	1.63%	1.54%
pop3	0.1.74%	0.25%	0.23%
ftp	0.13%	0.01%	0.003%
bittorrent	0.81%	1.78%	1.74%
msn	0.24%	0.04%	0.04%

Table 4.1: Composition of the traffic gathered at UNIBS border router

Since flows whose capture times are close are more likely to be from the same application the training and testing sets were chosen randomly from a much larger trace set. Table 4.1 shows the six protocols which were used in the training set and the percentage of the payload which belongs to each protocol. Next to each protocol name is the percentage of flows with that protocol and the portion of bytes with that protocol and lastly the percentage of the bytes in the application layer for that protocol.

Once for all the data sets, the unique three dimensional points are built there were 6347 entries for the training phase and 4677 entries for the testing phase. The percentage of each protocol follows the data which is described in table 4.1.

4.5.1 Experimental Results

Table 4.2 shows the accuracy of the SVM method which was used in [26] over the six common protocols on top of the Internet protocol. In all the protocols both precision and recall are relatively good but the problem is the computational cost of SVM method itself. Note that some of these protocols have very similar sizes because of the similarity in their protocol. For example pop3 and ftp protocols are very similar because first the server and the client are required to exchange user name and password. The main difference between these two protocols is the statistical difference between the length of the user name and passwords which are used for any of these protocols.

In our method depending on how many points from how many classes are kept in the TCAM, different results would be obtained. Which set of points should be hashed and kept in the TCAM is in itself an important problem. For the sake of simplicity once the training phase is done all the entry points are kept in the TCAM. The reason for this decision is that the number of these points in the training set is always very small (less than 10000) and it is doable in practice. In the training phase first three quarters of the flows from each class (i.e. protocol) are hashed using TLSH functions with different σ values and the optimal σ which maximizes the overall f-factor over the remaining flows in

	http	smtp	pop3	ftp	bittor	msn	unknown
http	94.9%	-	-	-	-	-	5.1%
smtp	-	93.3%	.1%	.7%	-	-	6.8%
pop3	-	0.4%	88.2%	3.7%	-	-	7.7%
ftp	-	-	0.5%	97.7%	-	-	1.8%
bittor	1.8%	-	-	-	96.8%	-	1.4%
msn	-	-	-	-	.1%	91.2%	8.5%

Table 4.2: Classification results for the UNIBS data set using SVM method

	http	smtp	pop3	ftp	bittor	msn	precision	recall	f-factor
http	99.99%	-	-	-	0.01%	-	99.99%	52.1%	68.4%
smtp	-	86.45%	5.8%	3.3%	-	4.3%	86.45%	96.76%	91.3%
pop3	-	15.2%	72.5%	5.07%	0.1%	7.03%	72.5%	99.83%	0.839%
ftp	-	15.1%	13.68%	63.9%	-	7.2%	63.9%	94.47%	76.28%
bittor	-	1.2%	6.9%	1.8%	90.1%	0.6%	90.1%	33.03%	48.3%
msn	-	1.9%	0.1%	0.6%	-	96.4%	96.4.5%	98.08%	97.2%

Table 4.3: Classification results for the UNIBS data set using SVM method with an average accuracy of 84.9 and recall of 79.01 and f-factor of 81.86

the set is picked. For the optimal σ, say σ_{opt}, we hash all the training entries in the TCAM and test them against the testing set. It is possible that one query point matches with multiple classes. In this case we assume that it is equally likely to be matched with every single of those matches. For example if an entry of ftp is matched with 3 http points and 7 ftp points for that entry we assign a 0.3 false positive and 0.7 true positive. If the query is not matched with any of the entries in the TCAM then we have a false negative and if a query of a class which does not exist in the TCAM is not matched with any of the entries in the TCAM we have one true positive. Table 4.3 shows the result of this experiment.

Since the majority of the flows are HTTP and in HTTP protocol the URL is transfered in the first three packets, the entries of this protocol are understandably more diverse than the rest of the protocols. As the classifier tries to cover more points of this protocol (by increasing of σ) and increase the recall for this class, the accuracy for this protocol faces a huge bump and it reduces so fast that the optimal result is the case that only 52.1% of http protocols are classified but almost all of them are classified correctly. While msn protocol has a better precision and recall in comparison to the SVM method the rest of the protocols suffer from a low precision or recall. We did not expect that this method will beat the SVM method because of its simplicity and lack of flexibility for having different

σ values for different areas in the euclidean space. Still as it is shown in this experiment it can be used as a good method to filter out certain set of protocols with a good recall.

One of the logical next steps in the experiments is to check the case where the entries in the TCAM are only from one of the protocols and the objective is to use the TCAM as a binary classifier. This way the network operator can filter out one type of flow or for every single flow perform a query on multiple TCAMs with different TLSH functions in order to cover all the protocols. For this classifier the σ value can be tuned with respect to precision or recall or f-factor in order to operate for different scenarios. For instance if the network operator wants to do traffic shaping for one single protocol such that it is not affecting the shape of the traffic for the rest of the protocols, the classifier should be tuned such that the precision is very high (almost 100%). On the other hand if the flow signature belongs to a virus and the objective is to filter out all those flows even at the cost of dropping some of the other unnecessary packets then the objective is to have a high recall. In order to have a good combination of recall and precision some variation of the f-factor can be used. Table 4.4 illustrates all of these scenarios for every single protocol. In this table every row belongs to a different experiment. In each row every column of different protocols shows the false positive of the TCAM for that protocol. For example in the experiment of "http (best f-factor)" 10.4 percent of the bittorrent protocols are mistakenly classified as an http protocol. In order to calculate these numbers for every binary classifier true/false positive/negative rates are calculated first. If false positive, false negative and true positive and true negative are respectively denoted by fp, fn, tp, tn then the precision and recall were calculated by the following formula.

$$Precision = \frac{tp}{tp + fp}, Recall = \frac{tp}{tp + fn}$$

Other than bittorrent the rest of the flows can be tuned to obtain a relatively high f-factor (above 83%). Note that bittorrent is the only flow type which gets a pretty bad recall which affects the rest of the metrics as well. One of the likely results is that the number of training points is too low or are from different torrent clients than the testing data.

As we increase the recall for every flow type there is an increase in false positives for the other flows but the decrease is not as sharp as the previous experiment. This can be explained by the fact that we have to increase the σ value in order to cover all the desired points but in a binary classifier the desired points are far less than the general classifier and hence it is more likely to cover the points faster.

In most of the cases we can achieve good recalls while keeping the precision more than 50%. This suggests that this technique can be useful as a preclassifier in order to filter out and monitor a certain set of flows. For example we can recall all the msn flows at the cost of covering 43.7% unrelated flows. So we effectively pruned most of the other flows and the

	http	smtp	pop3	ftp	bittor	msn	precision	recall	f-factor
http (best f-factor)	95.7%	0.2%	0.3%	0.6%	10.4%	-	99.04%	95.7%	97.3%
http (best precision)	65.45%	-	-	-	.4%	-	99.99%	65.45%	79.1%
http (best recall)	65.45%	100%	100%	96%	87%	99.8%	60.4%	99.9%	75.2%
smtp (best f-factor)	-	71.5%	0.3%	1.2%	-	-	99.1%	71.5%	83.1%
smtp (best precision)	-	58.1%	-	-	-	-	100%	58.1%	73.1%
smtp (best recall)	-	100%	70.8%	72.3%	1.8%	37.3%	43.9%	100%	60%
pop3 (best f-factor)	-	-	90.6%	0.6%	-	-	99.6%	90.6%	94.9%
pop3 (best precision)	-	-	80.9%	-	-	-	100%	80.9%	89.4%
pop3 (best recall)	-	77.5%	100%	76.6%	3.6%	81.7%	22.3%	100%	36.4%
ftp (best f-factor)	-	2.4%	1.9%	80.9%	-	-	88%	80.9%	84.3%
ftp (best precision)	-	-	-	68%	-	-	100%	68%	80.7%
ftp (best recall)	-	83.4%	98.3%	100%	7.2%	86.7%	10.9%	100%	19.6%
bittor (best f-factor)	-	0.4%	2.9%	-	35.7%	1.3%	79%	35.7%	49.2%
bittor (best precision)	-	-	-	-	19.4%	-	100%	19.4%	32.5%
bittor (best recall)	13.3%	100%	100%	95%	97.2%	100%	9.6%	97.2%	17.5%
msn (best f-factor)	-	5.4%	0.9%	2.4%	0.4%	93.6%	94.7%	93.6%	94.2%
done msn (best precision)	-	-	-	-	-	58.8%	100%	58.8%	74%
done msn (best recall)	-	57.7%	42%	41%	6.7%	99.5%	57.5%	99.5%	72.9%

Table 4.4: Using TCAM as a binary classifier for every single protocol

remaining flows can be classified at a much lower computational cost using a traditional stateful classifier.

4.6 Future Work and Conclusion

In this chapter we focused on one of the main problems in computer networks in order to use the existing TCAM in networking devices (mostly routers). The problem that has been addressed is real time traffic classification based on packet sizes. We used a novel algorithm for approximate nearest neighbor search using TCAM which was proposed in [51] and compared our method with the best existing algorithm in this area namely SVM method. While our method is not better than the SVM method, because of its minimal computational cost can be order of magnitude more scalable. Our results show that this technique can be useful for filtering out a certain set of protocols from a large set of network flows in real time.

Note that in our hash functions the value of σ was always static. Making σ dynamic (e.g. such that in more dense areas of the euclidean space σ is smaller and in more sparse areas of the space σ is larger) can improve this technique and is an avenue for future research. Moreover similar to SVM method mapping the three dimensional signature of each network flow to higher dimension in a smart way might result in performance boost as well. Working with better data and more reliable features like TCP flags of the packets can be some other directions for future works.

Chapter 5

Discussion and Conclusion

Ternary Content Addressable Memory is a special memory which can be used as a powerful computing device because of its parallel operations. In this thesis we just scratched the surface of the possibilities of exploiting its computational power and many problems remain open. We believe as TCAMs get cheaper and their memory capacity expand while their power consumption remain low; there would strong demands to use it in more general environments alongside a CPU or a GPU in order to speed up algorithms which can be parallelized. In this research we focused on three fundamental problems and one practical problem. The important fundamental problems were boolean matrix multiplication, approximate set membership using bloom filters and fixed universe successor problem.

For boolean matrix multiplication a simple algorithm was suggested that achieves the time complexity of $O\left(dN^2/w\right)$ where N is the length of the square matrix and w is the with of TCAM and d is the number of 1s in a row of the matrix with the highest number of ones.

For approximate set membership using bloom filters we modified the hash functions of the bloom filter in order to exploit the co-occurrence probability of possible members in order to reduce the false positive probability. Furthermore we showed that our modified bloom filter which we named COCA filter can reduce the false positive probability more than 21 times and it can reduce the number of bits required for the bloom filter by three quarters while keep the false positive probability constant.

In the problem of fixed universe successor problem we proposed two data structures which both beats their best possible data structures in the RAM model in terms of time complexity. The first data structure requires space of $O\left(\frac{1}{\epsilon}nU^{\epsilon}\right)$ and can perform all the operations of a priority queue in constant time. The second data structure keeps the space linear while achieving the time complexity of $O\left(\frac{\lg lgU}{\lg\lg\lg U}\right)$ which beats the best possible data structure in the RAM model with linear space namely y-fast tries.

Since the main industrial use of TCAM is in networking equipments we worked on one of the fundamental problems in computer networks namely real time network flow classification and showed that TCAM has a great potential in order to solve this problem with reasonable false positive and negative rates in real time.

References

[1] http://en.wikipedia.org/wiki/X-fast_trie. [Accessed September-2011]. ix, 14

[2] http://en.wikipedia.org/wiki/Y-fast_trie. [Accessed September-2011]. ix, 15

[3] http://www.snort.org. 42

[4] http://www.google.com/programming-contest, 2002. [Online; accessed 24-January-2011]. 32

[5] http://www.wikipediaondvd.com/site.php, 2007. [Online; accessed 24-January-2011]. 27

[6] http://schools-wikipedia.org, 2008. [Accessed January-2011]. 34

[7] http://en.wikipedia.org/wiki/Wikipedia:Words_per_article, 2009. [Online; accessed 24-January-2011]. 27

[8] http://www.biomedcentral.com/info/about/datamining, 2009. 36

[9] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:459–468, 2006. 46

[10] D. Anguita, A. Boni, and S. Ridella. A digital architecture for support vector machines: theory, algorithm, and fpga implementation. *Neural Networks, IEEE Transactions on*, 14(5):993–1009, sept. 2003. 44

[11] V.; Dinic E.; Kronrod M.; Faradzev I. Arlazarov. On economical construction of the transitive closure of a directed graph. *(in Russian), Dokl. Akad. Nauk. English Translation in Soviet Math Dokl.*, (194):11, 1970. 3

[12] Michael D. Atkinson and N. Santoro. A practical algorithm for boolean matrix multiplication. *Information Processing Letters*, 29(1):37–38, 1988. 3

[13] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, 2002. 7

[14] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *SIGCOMM Comput. Commun. Rev.*, 36:23–26, April 2006. 44, 45

[15] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970. 19, 20, 31

[16] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel H. M. Smid, and Yihui Tang. On the false-positive rate of bloom filters. *Inf. Process. Lett.*, 108(4):210–213, 2008. 20, 30

[17] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002. 20, 37

[18] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *CPM*, pages 1–10, 2000. 21

[19] Andrei Z. Broder. Min-wise independent permutations: Theory and practice. In *ICALP*, page 808, 2000. 21, 23

[20] Jeremy Buhler and Martin Tompa. Finding motifs using random projections. *Journal of Computational Biology*, 9(2):225–242, 2002. 20

[21] Ben Carterette and Fazli Can. Comparing inverted files and signature files for searching a large lexicon. *Inf. Process. Manage.*, 41(3):613–633, 2005. 17

[22] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002. 20

[23] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA*, pages 30–39, 2004. 20

[24] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 241–252, New York, NY, USA, 2003. ACM. 20

[25] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 1–6, New York, NY, USA, 1987. ACM. 3

[26] Alice Este, Francesco Gringoli, and Luca Salgarelli. Support vector machines for tcp traffic classification. *Computer Networks*, 53(14):2476–2490, 2009. 45, 47, 48

[27] C. Faloutsos and R. Chan. Fast text access methods for optical and large magnetic disks: Designs and performance comparison. In *14th International Conf. on VLDB*, pages 280–293. 18

[28] Chris Faloutsos and Stavros Christodoulakis. Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2:267–288, October 1984. 17

[29] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8:281–293, June 2000. 20

[30] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. 15

[31] Bogdan Georgescu, Ilan Shimshoni, and Peter Meer. Mean shift based clustering in high dimensions: A texture classification example. In *ICCV*, pages 456–463, 2003. 20

[32] Ashish Goel and Pankaj Gupta. Small subset queries and bloom filters using ternary associative memories, with applications. *SIGMETRICS Perform. Eval. Rev.*, 38:143–154, June 2010. viii, 2, 16, 40

[33] Taher H. Haveliwala, Aristides Gionis, and Piotr Indyk. Scalable techniques for clustering the web. In *WebDB (Informal Proceedings)*, pages 129–134, 2000. 20

[34] Marios Iliofotou, Hyun chul Kim, Michalis Faloutsos, Michael Mitzenmacher, Prashanth Pappu, and George Varghese. Graph-Based P2P Traffic Classification at the Internet Backbone. In *INFOCOM Workshops 2009, IEEE*, pages 1–6, April 2009. x, 45

[35] T. Karagiannis, A. Broido, N. Brownlee, K.C. Claffy, and M. Faloutsos. Is p2p dying or just hiding? [p2p traffic measurement]. In *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, volume 3, pages 1532–1538 Vol.3, nov.-3 dec. 2004. 42

[36] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: multilevel traffic classification in the dark. *SIGCOMM Comput. Commun. Rev.*, 35:229–240, August 2005. 44

[37] Hyunchul Kim, KC Claffy, Marina Fomenkov, Dhiman Barman, Michalis Faloutsos, and KiYoung Lee. Internet traffic classification demystified: myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 11:1–11:12, New York, NY, USA, 2008. ACM. x, 43, 44

[38] Jinyang Li, Boon Loo, Joseph Hellerstein, M. Kaashoek, David Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. In *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 207–215. Springer Berlin / Heidelberg, 2003. 17

[39] Z. Lin and C. Faloutsos. Frame-sliced signature files. *IEEE Trans. on Knowl. and Data Eng.*, 4:281–289, June 1992. 18

[40] Jiri Matousek. On restricted min-wise independence of permutations, 2002. 23

[41] Kurt Mehlhorn, Stefan Näher, and Helmut Alt. A lower bound on the complexity of the union-split-find problem. 17(6):1093–1102, 1988. 7

[42] J. K. Mullin and D. J. Margoliash. A tale of three spelling checkers. *Softw. Pract. Exper.*, 20:625–630, June 1990. 20

[43] J.K. Mullin. Optimal semijoins for distributed database systems. *Software Engineering, IEEE Transactions on*, 16(5):558–560, May 1990. 20

[44] T.T.T. Nguyen and G. Armitage. A survey of techniques for internet traffic classification using machine learning. *Communications Surveys Tutorials, IEEE*, 10(4):56–76, quarter 2008. 42, 43, 44

[45] Zan Ouyang, Nasir D. Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *WISE*, pages 257–268, 2002. 20

[46] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *SODA'05*, pages 823–829, 2005. 20

[47] R. Panigrahy and S. Sharma. Sorting and searching using ternary cams. *Micro, IEEE*, 23(1):44–53, jan/feb 2003. 8, 9, 10

[48] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 21–40, New York, NY, USA, 2003. Springer-Verlag New York, Inc. 17

[49] C.S. Roberts. Partial-match retrieval via the method of superimposed codes. *Proceedings of the IEEE*, 67(12):1624–1642, dec. 1979. 18

[50] Michael Saks, Aravind Srinivasan, Shiyu Zhou, and David Zuckerman. Low discrepancy sets yield approximate min-wise independent permutation families. *Information Processing Letters*, 73(1-2):29–32, 2000. 23

[51] Rajendra Shinde, Ashish Goel, Pankaj Gupta, and Debojyoti Dutta. Similarity search and locality sensitive hashing using tcams. *CoRR*, abs/1006.3514, 2010. 3, 41, 45, 46, 47, 51

[52] Eugene H. Spafford. Opus: Preventing weak password choices. *Computers & Security*, 11(3):273–278, 1992. 20

[53] STRASSEN V. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969. 3

[54] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10:99–127, 1976. 10.1007/BF01683268. 7, 11

[55] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space [theta](n). *Information Processing Letters*, 17(2):81–84, 1983. 7, 13

[56] Cheng Yang. Macs: music audio characteristic sequence indexing for similarity retrieval. In *Applications of Signal Processing to Audio and Acoustics, 2001 IEEE Workshop on the*, 2001. 20

[57] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006. 16, 17