Efficient Data Search and Synchronization for Tree-Structured Data on Mobile Devices

by

Mengyu Wang

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Science (MSc)

in

Computer Science

University of Ontario Institute of Technology

Oshawa, Ontario, Canada

(November, 2015)

Abstract

We consider two methods about operating data for tree-structured data sets. The first method is to search particular data items and still retains all the important metadata information; the second method is to allow the data synchronization between two tree-structured files. Both of the methods are based on a special data structure we proposed, called Bloom Filter Tree. It is to compute a bloom filter for each interior node of the tree, essentially building a co-existing BF-tree to enhance the original data tree in order to allow more operations. Using the BF-tree, these two processes become faster by pruning out entire subtrees from being searched and precisely locate the parts we are looking for. Experiments are performed to verify the efficiency of each method. What's more, we deploy the search method as an android application for practical purposes on mobile devices. We focus on a particular tree-structured dataset, DBLP, and select hundreds of records from it and enrich it as a neat XML file for parsing. With hundreds of records, more than 35,000 nodes in the tree structure, our method responds accurately and efficiently.

Acknowledgements

I would like to show my greatest respect and gratitude to my supervisor, Dr. Ying Zhu. Being her student is the best thing ever in my life. She has taught me a great deal about the field of Computer Science by generously sharing with me the creative ideas in her mind. She extends much care and patience on me and stimulates my potentials in the industry. Because of her, I become too much better.

I would also like to thank my parents, Qi Wang and Lijie Ma. Thanks for supporting me making my own decisions. Thanks for making me being in such a wonderful world.

This piece of work is dedicated to them.

Table of Contents

List of Figures

List of Tables

List of Acronyms

UOIT University of Ontario Institute of Technology

Rsync Remote Synchronization

BF Bloom Filter

IBF Invertible Bloom Filter

BF-Tree Bloom Filter Tree

BF-Sync Synchronization based on Bloom Filter Tree

1

Introduction

1.1 Introduction

Mobile devices have completely revolutionized the way we interact with our environment. They have become intertwined with the user, carrying them wherever they go. These changes are evident in both our social and professional environments. Mobile applications were complete packages installed onto the device, capable of completing the tasks as designed. Due to the limits of mobile devices, such as limited storage, low bandwidth etc., the recent trend of mobile application development has favoured approaches to decrease the amount of local storage, save battery life and increase performance.

We consider data that are structured as trees, examples include XML trees and DOM [1] trees. The leaf nodes in the data tree are the data elements; the interior nodes contain metadata as well as possibly data elements. That is, the set of data elements reside in the leaf nodes and possibly in the interior nodes, while the metadata reside in the interior nodes. As an illustrative example, suppose a collection of text data — these could be documents or forum posts or just pieces of text that users contribute to a common repository. This data is naturally organized into a tree, the children of the root node may represent the different content types, the next levels down may represent the timestamps, and the further down the breakdown may be by user ID, etc. The leaf nodes contain the actual text data. There are two mainly operations considered in this research; one is how to search particular elements in this kind of big data tree in a fast way; the other one is if we store two copies of a data tree without having them on the same machine, and changes happened to one of the copies, how to do the data synchronization to have these two copies to be

identical.

With the widely usage of android smart phones and tablets, we are aiming to implement the methods of these two operations onto mobile devices. Thus, the limitations of the mobile devices like low bandwidth, limited local storage, short life span battery, etc. should be considered through the whole development.

1.2 Objective and Methodology

The first objective we consider is the problem of searching for a particular data element *x* in the tree-structured set.

Given a set of data in tree-structure, the basic operation is always about searching for a particular element *x* to see whether it is in the set. Usually, the first solution came up into mind is simply taking all the leaf nodes into a list, and search the list for this particular date element *x*, then we could obtain a boolean answer of whether *x* is a member of the set. However, most of the time, we are not only restricted to the satisfaction of a boolean answer of the existence of a particular data element, but want to explore more information related to this being searched element. The naïve way of doing a breadth-first or depth-first tree traversal would yield *x*'s location(s) in the tree and hence all its associated metadata. The cost of such a traversal is that every node in the tree must be visited and every leaf node also compared with *x*. Since, it is a tree structure, any of the metadata associated with the *x* could be reside in the interior nodes. We would lose vital information encapsulated in the tree organization, and therefore lose its original intent and advantage. To avoid losing the vital information encapsulated in the tree organization, capturing the location and metadata for *x* is of essence.

To realize this, a more efficient method to find *x* and obtain its metadata and location(s) in the

tree is proposed in this research. At each interior node, we compute a bloom filter for the entire subtree rooted at this node. By checking the bloom filter, we can eliminate the possibility that *x* is in that subtree and altogether forego the traversal or search in that subtree. This way, every subtree not having *x* as one of its leaf nodes does not get traversed or searched at all, thus saving the unnecessary cost. Essentially we construct a tree of bloom filters; the tree has the same topology as the data tree, minus its leaf nodes. A bloom filter is simply a bit vector of *m* bits. Each computed bloom filter can easily be stored at the corresponding interior node in the original data tree. We construct the bloom filter tree from bottom up. First, we compute the bloom filters for the lowest-level (topologically speaking) interior nodes, i.e., those with leaf nodes as children. The bloom filters of upper interior nodes are obtained by taking the union of the bloom filters of all their children. Proceeding thus from bottom up, we build the entire bloom filter tree right up to the root. The search for *x*, in contrast, begins at the top. We first check the bloom filter at the root to see if *x* exists in the entire tree. If it does, then we go down one level to check the bloom filters of the children of the root, to see if *x* is in any of these subtrees. If a check yields positive, then we know that most likely *x* is in that subtree and proceed to keep checking in that subtree top-down, in the same fashion. If a check yields negative, then we know with certainty that *x* is not in that subtree and can prune out that subtree. Throughout this top-down pruning search process, we store the path(s) that lead to data element *x*. For every instance of x found in the data tree, the path to it from the root is obtained and this path gives the location and metadata information. The answer to the query for *x* is therefore not just a boolean, but a set of paths (obviously empty set means no instance of *x* found).

The bloom filter tree is constructed from bottom up. The first level interior nodes' bloom filters contain all the information of their children. Thus, all the parent interior nodes associated with these first level interior nodes will contain the same information. Taking advantage of this property is the motivation of synchronizing two tree-structured files by building bloom filter

trees. These two tree-structured files are coming from the same copy and now locate on different machines without access to each other. There are some changes happened to one of the copies and the problem is how to update the old one to be the same as the new version. The ultimate goal is to perform the algorithm on android mobile devices, thus we are assuming that the two machines are connected by a low-bandwidth and preferred to send messages through the communication link at the minimum cost. To really speed things up, taking advantage of the similarity between two files is a wise choice. The BF-Sync algorithm efficiently computes which parts of a source file don't match some part of an existing destination file. These parts could be sent across the link directly. The receiver can then construct a copy of the new version file by updating the differences with the corresponding transferred data. After building up two bloom filter trees, comparing the bloom filters level by level from top to bottom could help locate where the differences are. What's more, during the whole recursively comparing process, if two bloom filters from each side are the same, which indicates there's no difference between their children nodes, then there's no need to go any further. We prone out those subtrees rooted at the equal bloom filters nodes, which really speeds the whole process up.

1.3 Research Contributions

This thesis contains two main research contributions. We investigate the problems of data search in tree-structured data sets and remote synchronization of two data trees over the network. To solve the first problem, we propose an algorithm that uses bloom filters to succinctly represent data elements contained in subtrees and performs a search by scanning these bloom filters in a top-down manner starting at the root node. The search is made efficient by being able to eliminate entire subtrees from the top down. We verify this in our experimental evaluation. We

also present a theoretical analysis of the effect on search performance of the number and positioning of a fraction of bloom filters in the tree (instead of placing one at every node).

For the second problem of remote data synchronization of tree-structured data, we propose an algorithm that generates and maintains bloom filters on each of the two data trees, and synchronizes them by selectively exchanging bloom filters. We also implement the classic *rsync* [2] scheme for remote data synchronization, for comparison and evaluation of our method. We experimented with two variants of data, one with append-only operations and the other with randomly located differences between the two trees. For the second variant, we experimented with a small number of differences. Our assumption is that data synchronization is performed on a frequent regular basis between mobile device and the cloud or between two mobile devices, and given the usage patterns of most mobile devices, not a large amount of data will change during a short interval of time. We use three metrics to measure the performance: number of nodes visited in the tree, number of bytes transmitted between the data sites, and the runtime. In all three metrics, our method has better performance than *rsync*, for both variants of data.

1.4 Thesis Organization

This thesis is organized in five chapters. Chapter 1 is an introduction to the background and the objectives of the whole research.

Chapter 2 illustrates the features of the bloom filter, and how to take advantages of these features. Some variants and applications of the bloom filter are reviewed as the motivation of our BF-Tree construction. Also in this chapter, we clarify the synchronization techniques, especially *rsync*, as the foundations of BF-Sync.

Chapter 3 presents a method to efficiently search for data items in data sets that are organized into hierarchical tree structures by constructing co-existing bloom filter trees. The performance are investigated thoroughly and compared with the naïve method of tree traversal. In order to reduce the amount of overhead associated with computing and maintaining a full bloom filter tree, we also consider the problem of using a sparse bloom filter tree instead of the full one in which every node in the data tree in indexed by a bloom filter.

Chapter 4 we address several issues about using BF-Sync algorithm for data synchronization, also we present two main scenarios of the way changes happen; one is the differences are randomly located in the middle of the whole set; the other one is the differences are only appended in the end. As well, implement and discuss the results of different situations, and finally, analyze the result and the performance of the implemented BF-Sync algorithm.

2

Literature Review

2.1 Why Bloom Filter

Thinking of a general scenario, given a particular element x, it could be any integer, string, etc. We have a data structure, which represents a set *S* of *N* elements and the problem is to determine whether *x* is in set *S*. For this data structure, it should enable of two basic operations, one is the ability to dynamically add elements to the set, and the other is a search operation to determine whether the given element is a member of *S*.

There are many ways. One can think a set of hashed objects, which represents a range of hashed values. This is much more memory efficient than just storing all the elements in a set, especially when the hash function is well chosen. Moreover, the type of the elements in set *S* is not restricted to any specified one. A little more explicit of how this works. We have a set of objects $x_{0,}x_1,...x_{N-1}$, where *N* denotes the number of elements in *S*. For each element, we compute an *m*-bit hash function $h(x_j)$. The hash function takes an element as input and arbitrarily comes up with an output of *m* bits. Thus the set *S is* represents by a set of hash values like $h(x_0), h(x_1),...h(x_{|N|-1})$. This basic hashing approach requires roughly *m*|*N*| bits of memory. This works fine if *N* is not large. Otherwise, there exists the danger of making errors and more memory storage. The danger of making errors is because of the hash collision, $h(x) = h(x_j)$ for some *j*. In this case, the search for *x* will always be positive as long as x_j in the set. This collision

probability of hash function $x \in S$ is directly correlated to the false positive rate. Even though, for many applications, it's acceptable to have a small probability of a false positive. However, to maintain a low false positive rate as *N* increases, the range of $h(x_j)$ must increase and therefore *m* must increase, thus resulting in higher space and time cost. Suppose we want to represent a range of integers of length *n.* As an alternative to hashing or storing directly, we could represent S using a bit vector of *n* bits, like [0,1,… *n*-1]. In this bit vector, bit *x* is set to 1 if $x \in S$ and 0 if $x \notin S$. To check whether *x* is in *S* is a simple look-up of bit *x* in the bit vector. This works fine if the range of integers dictated by the elements in *S* is not too large and the set *S* is not too sparse over the range. Otherwise, it will be too wasteful in space and look-up time. This bit vector is in a small and well-defined range. However, in general world, the elements in the set could be complicated not just a range of well-defined integers. Thus combining the method of bit vector and hashing method, we can use hashing to number the element in set *S.* Suppose $h(\bullet)$ is an *m*-bit hash function and a set *S* of *N* elements are of any type, $x_0, x_1, ...x_{|N-1|}$. We are going to represent this set using a bit vector containing $2^m - 1$ integers. For each x_j, we set the $h(x_j)th$ element in the bit vector to be 1, where $h(x_j)$ is a number in the range of 0 to $2^m - 1$. To check whether *x* is a member of *S,* we only need to check whether the $h(x)th$ element in the bit vector has been set to 1. This scheme is improved but it doesn't avoid the problem of hash collision and even takes more memory storage. In order to avoid hash collision, how about using multiple cells with different hash functions. There comes an idea of using *k* cells, each based on an independent *m*-bit hash function, $h_0, h_1, ...h_{k-1}$. In this data structure, there exists *k* bit vectors and each vector contains 2^m bits. We are adding an element *x* by setting $h_0(x)th$, $h_1(x)th$, … $h_{k-1}(x)th$ in each corresponding bit vector. To check whether *x* is in the set is by simply checking whether all the corresponding bits have been set to 1 in each vector. This idea does decrease the probability of

hash collision but if the number of elements increasing a lot, to maintain the low false positive rate, we need to increase the size of all the bit vectors which definitely results in higher space cost. A variation of this multiple bit vectors is using a single bit vector but simultaneously multiple hash functions, which is the bloom filter data structure. Bloom filter offers a better way to control the size of the bit vector while keeping a low false positive rate. The underlying data structure is a bit vector V with $|V| = m$ and k independent hash functions $h_0, h_1, ..., h_{k-1}$ that map items in the data set to the range $|m| = \{1, ..., m\}$. All bits in V are initialized to 0. For each element x in the data set, we simply set all the bits at $h_0(x)th, h_1(x)th...h_{k-1}(x)th$ positions in the bit vector to be 1. Given x, if not all these k bits are 1, then it is known with certainty that $x \notin S$; otherwise it is with high probability that $x \in S$. However, there remains probability that $x \notin S$ but it yields positive. This type of error is known as false positive rate. False positive rate is an error when the element is not in the set but it yields positive. It occurs because of the hash collision, which is due to some different elements map into the same positions. There is another error rate called false negative rate. In contrast, this error rate means the element is in the set but it yields negative. The false negative rate will never happen to the Bloom Filter and we can control the false positive rate by designing the filters on our own.

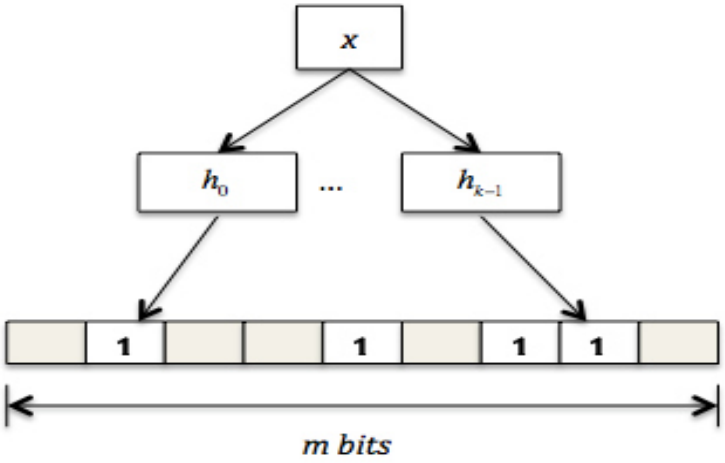

Figure 2.1 Overview of a Bloom filter

Even the structure offers a compact probabilistic way to represent a set that result in false positives, never in false negatives (claiming an inserted element to be absent). Thus this makes bloom filter popular for many kinds of tasks that involve lists and sets. The accuracy of a bloom filter depends on the size of the filter, the number of hash functions used, the number of elements added in the filter. To compute the probability of a false positive rate, suppose we have an empty bit vector *V* and insert an item. As known, all the bits in the *V* at $h_0(x)th, h_1(x)th...h_{k-1}(x)th$ positions are set to 1. Here we assume that a hash function selects each array position with equal probability. Thus, when inserting one element into the filter (*m* is the number of bits in the Bloom filter), the probability that a certain bit is not set to 1 is in equation (2.1) [3]:

$$1-\frac{1}{m} \tag{2.1}$$

There are k hash functions. This is same as independently choosing k bits and setting them to 1. Thus, the probability that a certain bit in V is still 0 is in equation (2.2) [3]:

$$(1-\frac{1}{m})^k \tag{2.2}$$

After n insertions, the probability that a certain bit is 1 is [3]:

$$1-(1-\frac{1}{m})^{kn} \tag{2.3}$$

The process of test for the existence of a particular data element in the set involves hashing this item k times. The false positive occurs if the k times hash functions map into the bit, which are set to 1 already. Thus, the probability of false positives could be [3]:

$$(1-(1-\frac{1}{m})^{kn})^k \tag{2.4}$$

Fig 2.2 shows the false positive probability rate for Bloom filters with the number of inserted elements in a range of 1 to 100.

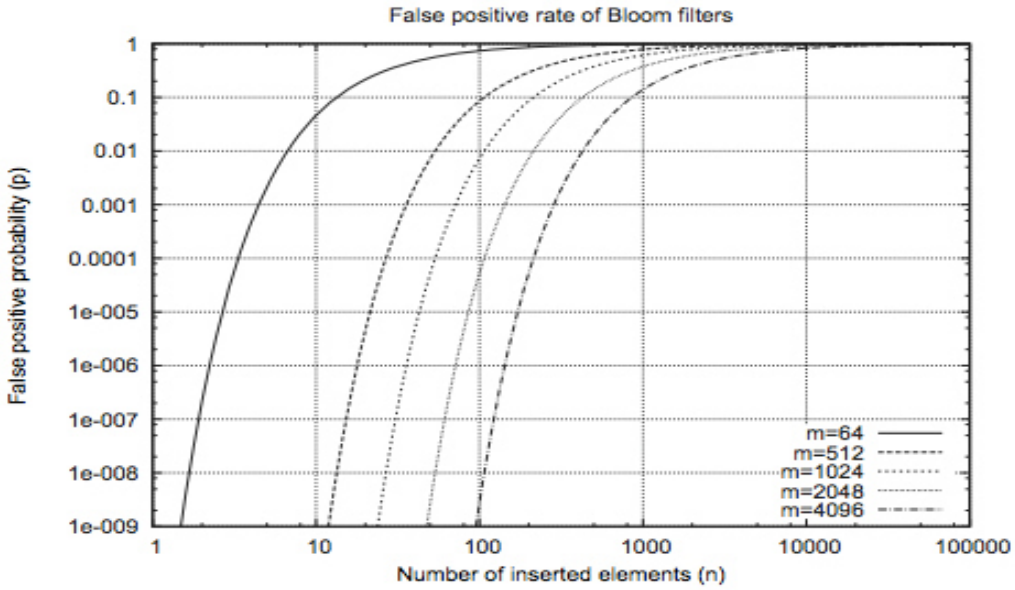

Figure 2.2 False positive probability rate for Bloom filters [4]

According to the equation (2.4), we can see the false positive rate decreases as the size of Bloom filter *m* increases. The false positive rate increases as the size of elements inserted *n* increases. We note that $e^{-kn/m}$ is a very close approximation of $(1-\frac{1}{m})^{kn}$ [3]. Thus:

$$(1-(1-\frac{1}{m})^{kn})^k \approx (1-e^{-kn/m})^k \qquad (2.5)$$

With a pre-specified set size *N* and false positive rate, one can find the required size of bit vector *m*. Even for a low false positive rate of 0.1, *m* is only approximately 10*N* bits. This is considered constant space because *m* doesn't depend on the data size of each element in *S*. The small storage

cost of the bloom filters naturally leads itself to the tree structure of the data set. The Bloom Filter Tree structure proposed in Chapter 3 is just taking advantages of the properties of bloom filters.

2.2 Applications and Extensions of Bloom Filters

Bloom Filter data structure has been widely used to solve many problems and was first introduced in the 1970's by Burton H Bloom [5] for simple text search. Because of its extremely simple randomized data structure for representing a set of data for membership queries, it has found applications in many areas including databases, computer networks, social networks and cryptography. Even its space efficient representation comes at the cost of durable false positive rate, the huge space savings often outweigh a sufficiently low rate.

Bloom filters are found very early uses in databases [3]. Suppose one wanted to determine the students in UOIT that live in cities where cost of living is greater than 5,000 dollars. In a distributed database, one host might hold the information regarding the cost of living; while another might hold the information regarding where students live. Rather than having the first database sending a list of information regarding the cost of living to the second, it can only send a bloom filter of the list. The second can also send back a potential bloom filter. In this way, the communication between two hosts is highly reduced.

Bloom filters can also be used for differential files [6]. Suppose there is a file of record at the beginning of the day, one wanted to know whether there would be some changes happened to this file during the day. Instead of tracking the whole records, one can keep a bloom filter. The list of the changed records can be kept as a bloom filter that helps reduce the storage.

Fan, Cao, Almeida, Broder [7] describe Summary Cache, which uses bloom filters for web cache sharing. A proxy attempts to determine if another proxy cache holds the desired web page. In

summary cache, proxies do not transfer URL lists corresponding to exact content of their caches, but instead bloom filters, which represent the contents of their cache. If a proxy wants to check if the desired web page is included, it only needs to check the bloom filter. This technique is used in the open source Web proxy cache Squid, where the bloom filters are referred to as cache digest.

Because of the space efficiency of bloom filters, it has been used for constructions for peer-to-peer networks. While keeping a list of objects stored at every other node in a peer-to-peer system may be prohibitive, keeping a bloom filter for every other node may be tractable [6]. For example, instead of using a 64-bit identifier for each object, a bloom filter could use 8 or 16 bits per object. A prototype P2P system dubbed PlanetP based on this idea is described in [5]; the filters actually store keywords associated with documents instead of object IDs. In [8], there are ideas that introduce some hierarchy, so that groups of nodes are governed by a leader in the networks. The leaders are meant to be more stable and long lasting. So the leader stores keywords associated with the group of nodes. Thus, the group leader controls routing within a group and other groups.

Byers, Considine, Mitzenmacher, and Rost [9] demonstrate an area where bloom filters can be used in solving approximate set reconciliation problems. This is one of the motivations of our data synchronization method based on bloom filters. They suppose peer *A* has a set of items S_A, and peer *B* has a set of items S_B. Peer *A* and *B* would like to find the items that *B* does not have, which is the set difference, $S_A - S_B$. They propose one approach that is to have *B* to send *A* a bloom filter; *A* then runs through its elements, checking each one against the bloom filter and sending any element that isn't in S_B according to the filter. The determination of the set difference will be faster when the difference is small. As long as we can figure out the set difference, it is also possible to determine the set intersection. Reynolds and Vahdat [10] propose their approach that peer *B* can send a bloom filter representing S_B to *A*; *A* then sends the elements

of S_A. According to the boolean result of bloom filter, we can find out which elements in *A* are in *B*. These are the elements exactly belongs to $S_A \cap S_B$. To handle conjunction queries involving multiple nodes, the set intersection methods are used to reduce the amount of information that needs to be sent to determine the appropriate documents.

The de Brujin graph data structure is widely used in next-generation sequencing in molecular biology and many programs are relying on in-memory representation of this kind of graph, which results in a requirement of a large amount of memory. Chikhi, R., & Rizk, G. [11] encode based on bloom filters to a new, space-efficient representation of the de Bruigin graph.

Some recent best applications of bloom filters related to our daily life are introduced in [12][13]. Facebook uses bloom filters for typeahead search, to fetch friends and friends of friends to a user typed query. The bloom filter is just 16 bits per friend connection (an edge in the facebook social graph) and they have called it "world's smallest bloom". Another application is familiar with the Yahoo email. When you log into Yahoo mail, the browser page requests a bloom filter representing your contact list from Yahoo servers. The bloom filter is compact and easily fits in your browser cache. When you send an email to other people, the browser-side javascript quickly checks the bloom filter in the browser cache for those email addresses.

Section 2.1 describes the mathematics behind bloom filters. There are also some important variations. Suppose we have a set that is changing over time, with elements being inserted and deleted. Inserting elements into bloom filter is easy; just like described in 2.1, hashing the elements *k* times and set these bits to 1. However, the deletion process could not be simply performed in reverse. If we delete one element by changing its corresponding hashing bits to 0, then some other elements stored in the set might be influenced. A bloom filter can easily be extended to support deletions by adding a counter for each element of the data structure. *Counting Bloom filters* [7] was first published to avoid this problem and some improved versions in

[14][15]. The structure works in a similar manner as a standard bloom filter. The array positions are extended from a single bit to n-bit as a counter, which makes the delete and insert operations possible. When an item is inserted, the corresponding counters are incremented; when an item is deleted, the corresponding counters are decremented. There is also one problem with the bloom filter is that it allows any hash function to map to any bit in the vector, thus for some member, more than five hash functions map to the same memory segment, which exceeds the lookup capacity of the memory core. The problem is solving by restricting the range of each hash function a given memory presented in [16] called *Parallel Bloom filters*. Another problem with bloom filters is that you must know an expected capacity in advance. If the prediction is too high, you'll waste space. If the prediction is too low, the rate of false positives will be increased before item insertion. [17] proposed *Scaling Bloom Filters* to solve this problem by starting with a small bloom filter and creating additional ones as needed. Compressed Bloom filter was introduced in [18], which improves performance when the bloom filter is passed as a message, and its transmission size is a limiting factor. This structure is particularly useful when information must be transmitted repeatedly, and the bandwidth is a limiting factor. Shanmugasundaram et al. [19] presented a data structure called *Hierarchical Bloom Filter* to support substring matching. This structure supports the checking of a part of string for containment in the filter with low false positive rates. It works by splitting an input string into a number of fixed-size blocks. These blocks are then inserted into a regular bloom filter. It is possible to check for substrings. What's more, several proposals [20][21] even have suggested providing some basic security during the hashing of bloom filters so that the identities of the set elements represented by the BF are less clearly visible for observers or attackers.

Bloom filters are in many shapes and forms and widely used. This has been reflected in some very recent research and algorithms proposed either directly or indirectly based on bloom filters. [22] proposes an exact set reconciliation algorithm based on bloom filters, in which the

estimation of symmetric difference size is using bloom filters, and multiple polynomial values according to the estimated symmetric difference set are calculated, and then the rational polynomials are interpolated, finally the union of sets is determined through factoring the rational polynomials. In [23], the authors firstly describe an algorithm called *Invertible Bloom Filter*, which is taking an existing IBF [24] and introducing a subtraction operator. And based on IBF, the Difference Digest structure is proposed, that allows two nodes to compute the elements belonging to the set difference. An advanced bloom filters based algorithm for efficient approximate detection of duplicates in data stream was presented in [25]. A generic framework for de-depulication was provided by combing the working principle of reservoir sampling and bloom filters. Another BF-based algorithm called data hiding algorithm aims to contribute in network security in [26], in which the data security issue for many sensor network applications is solved by uniformly and randomly embedding sensitive data into the ordinary data using bloom filters. All these BF-based algorithms are introduced within recent five years; they are making the best use of the features and characteristics of bloom filters, and applying them into specific data structure for specific purposes, which is quite similar with the goal of our research.

2.3 Synchronization Techniques

Data synchronization is the process of establishing consistency among data from a source to a target data storage and vise versa and the continuous harmonization of the data over time [27]. In Computer Science, it refers to the idea of keeping multiple copies of a dataset in coherence with one another.

Given two versions of a data set on different machines, say an outdated and a current one, how can we update the outdated one with minimum communication cost? The obvious solutions are to send all of the current data or compress the current data then send it or only send the compressed

differences between two data sets. Synchronization techniques are considering two versions of data have no access to each other. It is a remote data synchronization technique. There are two aspects of the problem; one is file synchronization, eg. *Rsync,* updating an outdated file to be identical to a current one; the other is set reconciliation. The formalization of set reconciliation problem is as follows: given node *A* and *B,* each with a set of *b*-bit strings, how can both nodes determine the union of two sets with a minimum of communication cost and a minimum of reconciliation latency [28]. The data synchronization talked in this research is between files.

The methods of finding data differences between set reconciliation and file synchronization are also different. The naïve method of set reconciliation is to determine the set difference for each party by sending an encoding of their entire set and then one scanning the other entire list to remove shared elements. Minsky, Trachtenberg, and Zippel [28] discovered a method to use characteristic polynomials for set reconciliation. A key part of this approach is the representation of sets by characteristic polynomials. The key of this data synchronization protocol algorithm is to translate the data characteristic polynomial. It is simply to say that each reconciling host set maintains its own characteristic polynomial. Through the observation of the two host sets, one can evaluate their characteristic polynomials on several sample points and then using a rational function to interpolate these points. The numerator and denominator of the interpolated function are the differences between the two host sets. However, the computational complexity of this technique has two terms; one is from the cost of evaluating the characteristic polynomials at the chosen points; one is from the rational function reconstruction, which make the method is complex to be implemented.

The synchronization techniques of file synchronization update the data of two remote files to be identical and without having them on the same machine. The *rsync* [2] propose a method to locate the differences in a single pass very quickly using special properties of the rolling checksum and the strong multi-alternate search mechanism. The first rolling checksum needs to be very cheap to

compute for all byte offsets but also effective and efficient. And the second checksum needs to have a low probability of collision. The algorithm should not assume any prior knowledge of the source file and the destination file and also should work on random data. *Rsync* doesn't assume one or more files are same on the either end, it's actually using the algorithm to compare every part of each file.

For example, a very large file, first we run a weak rolling checksum across the whole file to compute at every byte offset, and then we only transfer the blocks which are not similar. The role of this fast rolling checksum is to act as a window- rolling filter. In practice, the author found that the probability of rolling checksum matching when the blocks are not equal is quite low. This is important because the strong checksum is expensive and it would only be calculated for the blocks where the rolling checksum matches. For this part, the benefits are saving time and bandwidth.

2.4 Rsync

The *rsync* algorithm proposed by Andrew Tridgell and Paul Mackerras [2] is a classic data synchronization algorithm. *Rsync* does remote synchronization and updates the data of two files to be identical and without having them on the same machine. The algorithm can be used in the condition where the two machines are connected by a low-bandwidth high latency bi-directional communications link. It was published in 1996, the algorithm remains a classic and people nowadays are still using *rsync* in many applications.

Rsync is a remote file (or data) synchronization protocol. It allows you to synchronize files in two different computers or ends, which means it makes both of the copies the same. *Rsync* is capable of synchronization files without sending the whole file across the network.

Imagine you have a file on your local hard drive, and copy of that file on a remote hard server. These two are connected by a slow communication link. Now you make a change to the file on your local hard drive. Since it is only a small change to the whole file, you don't want to send the complete file across the link. You may only want to send the differences. This is how *Rsync* works. The basic idea of *rsync* algorithm is that it can efficiently compute the similar parts of the source file and the destination file. Only parts of the source file which are not matched need to be sent verbatim. It takes advantages of the similarity and only sends the differences.

Without having the two files on the same machine, *Rsync* supplies a method to detect the differences. The most salient parts of the algorithm are the rolling checksum and the strong multi-alternate search mechanism. The first rolling checksum needs to be very cheap to compute for all byte offsets but also effective and efficient. And the second checksum needs to have a low probability of collision. The algorithm should not assume any prior knowledge of the source file and the destination file and also should work on random data. *Rsync* doesn't assume one or more files are same on the either end, it's actually using the algorithm to compare every part of each file.

Suppose two files, *A* is on NEW side and *B* is on OLD side. *B* is an updated version. The OLD divides the old version of the file into blocks of size *S* bytes. The last block may be shorter than *S* bytes. This is done logically, internally in the memory. For each of the blocks OLD calculates two checksums: a weak "rolling" 32-bit checksum and a strong 128-bit checksum, then these checksums are sent to NEW.

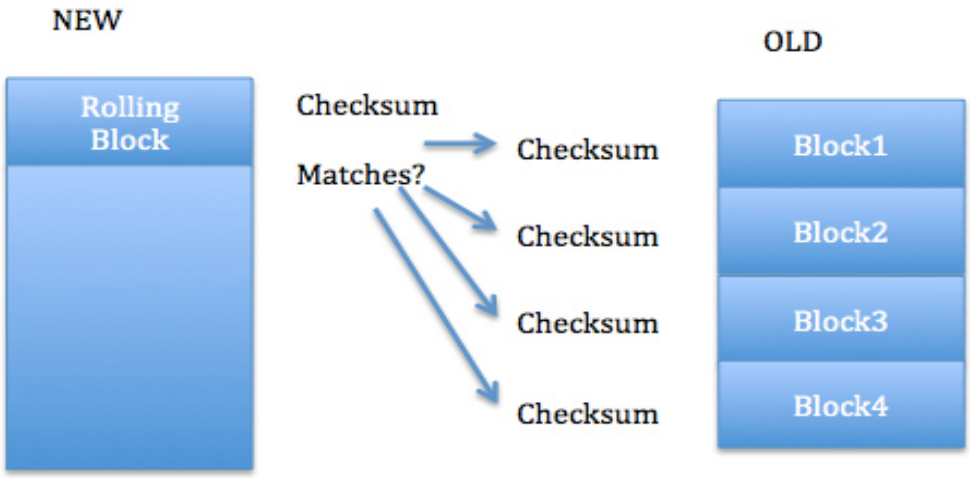

Figure 2.3 Rsync: After receiving the checksums from old version, the new file begins rolling checksum for matches.

NEW searches the newest version of the file for blocks of data that has the same "rolling" checksum as those found in the old version of the file. This is done by fist calculating the "rolling" checksum for the very first block of data (*S* bytes) and can be done in a single pass. If this checksum does not match any checksum in the old file, NEW moves 1 byte down the new file and calculates the checksum for this new block. NEW thus calculates checksums for every possible *S* bytes block in the new file, to search for matches to blocks in the old file. If NEW finds a block with the same checksum as one of the checksums received from OLD, then it considers that block to exist in the old version. NEW now skips to the end of this block and continues searching for checksum matches from there.

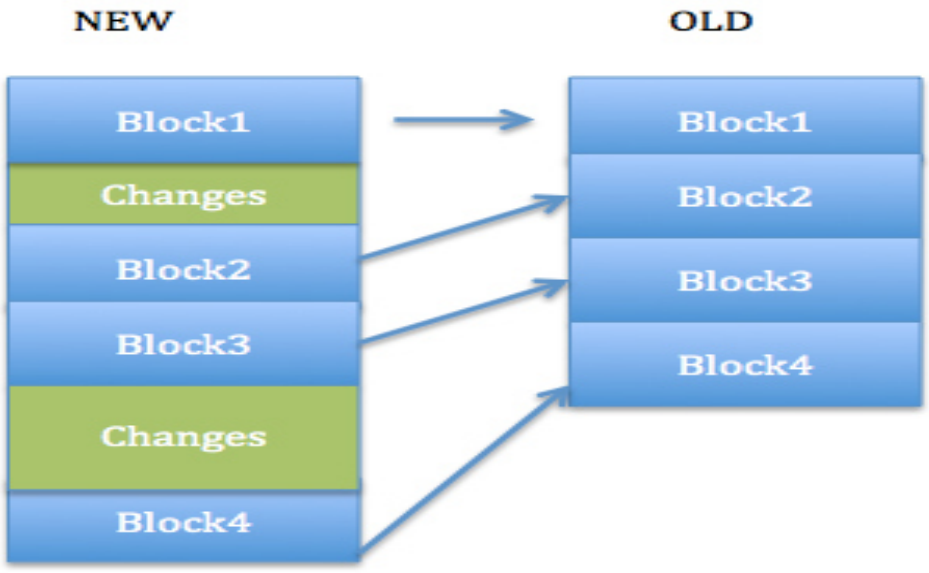

Figure 2.4 The changes are detected using block checksums.

Eventually, NEW will find multiple blocks of data matching checksums of the old file. These are the data not changed. Between these blocks is the part that doesn't match, which is the changed or new data. Thus, NEW sends OLD a sequence of instructions for constructing a copy of *A*, which includes a list of block references for the sections are not changed and literal data, which doesn't match any of the blocks of OLD. The literal data is sent verbatim.

The Figure 2.5 shows the process of data synchronization between client and server. First, the server side calculates the checksums of the raw data and then sends the checksum stream to the client side. After receiving the checksum stream, the client detects the differences that are the parts not matched. The client sends the differences and merge instructions for the server to construct a copy. The server then construct a copy of the new version file by merging the references to the existing parts and the verbatim material.

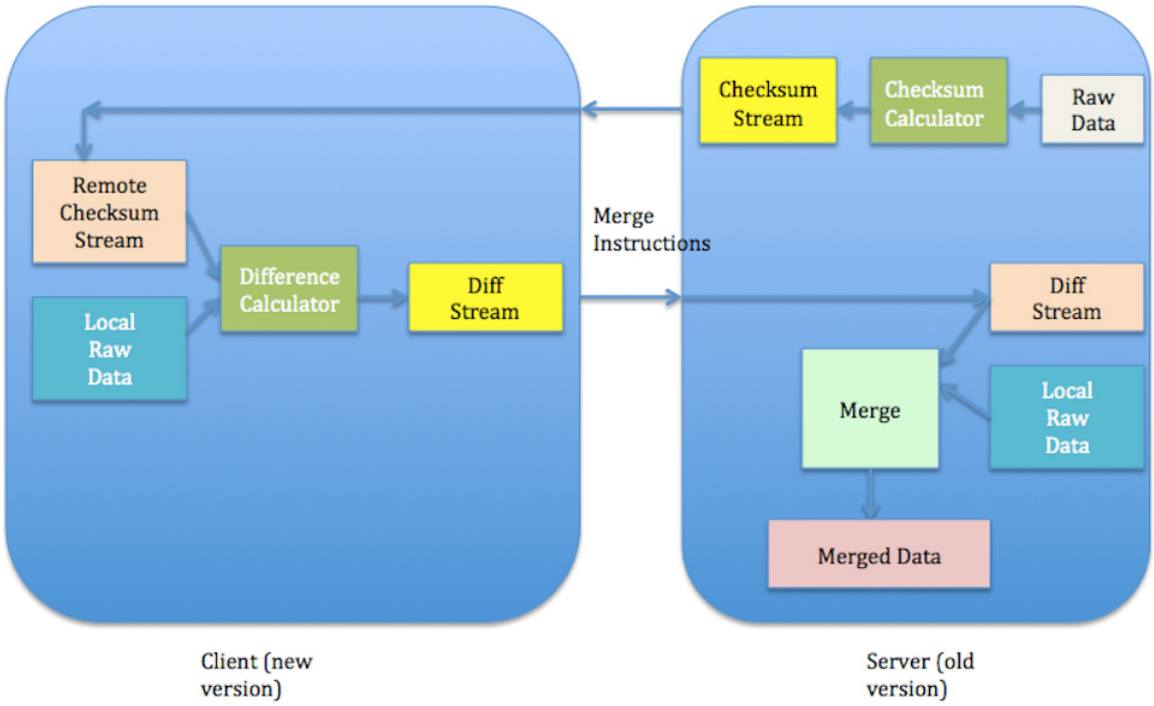

Figure 2.5 The Rsync process

Rsync is widely used to back up data from clients to a server. Of course you can use *rsync* to do restoring if necessary.

Another use case for *rsync* would be to implement resumable uploads and downloads. If you start a file upload via *rsync*, and if it is interrupted for any reason, for example the connection fails halfway through the uploading, you can resume it easily with very little cost. *Rsync* will then compare the full file on the client, to the half file on the server, and find that the last half is missing, and start sending the second half of the file. The same is true for downloads.

Rsync could also be used to implement incremental file versioning. By incremental file versioning we mean keeping different versions of the same file, but rather than keeping the full version of

each file, you could just keep the *rsync* merge instructions needed to go from one file version to another. Thus, you just keep the "incrementation" from one version to another [29].

3

Bloom Filter Tree

3.1 Bloom Filter Tree Structure

Suppose there is a data tree T with n nodes, each node storing data information. The leaf nodes may contain the data elements of specific detailed information and the interior nodes are the metadata including abstracts or the summaries related to the leaf nodes. This kind of tree-structure data set is quite common in practice examples like XML files. Given a set of data of tree-structure, the basic operation is always about searching for a particular element x to see whether it is in the set. However, most of the time, we are not only restricted to the satisfaction of a boolean answer of the existence of a particular data element, but want to explore more information related to this being searched element. Since, it is a tree structure, any of the metadata associated with the x could be reside in the interior nodes. To avoid losing the vital information encapsulated in the tree organization, capturing the location and metadata for x is of essence. To realize this, we need to construct the search on the tree. We proposed a method to construct a co-existing bloom filter tree for further efficiently operations on the data.

The constructed bloom filter tree has the same topology as the data tree, minus its leaf nodes. Since a bloom filter is just a bit vector of m bits, it can just be stored as a new field at the corresponding interior node in the original data tree. The bloom filter tree is constructed from bottom up. First, we compute the lowest-level (topologically speaking) interior nodes, whose children are the leaf nodes. To build up the upper nodes' bloom filters, one of the nice properties of bloom filter has been utilized. The union of a collection of bloom filters can be simply computed by the bitwise OR of each filter. In the other word, a collection of sets' bloom filter can be computed as the bitwise OR result of separated bloom filters for each set. Thus, the bloom

filters of the upper interior nodes are obtained by taking the union of the bloom filters of all their children. Eventually, the entire bloom filter tree is finished building when the union is computed right up to the root node.

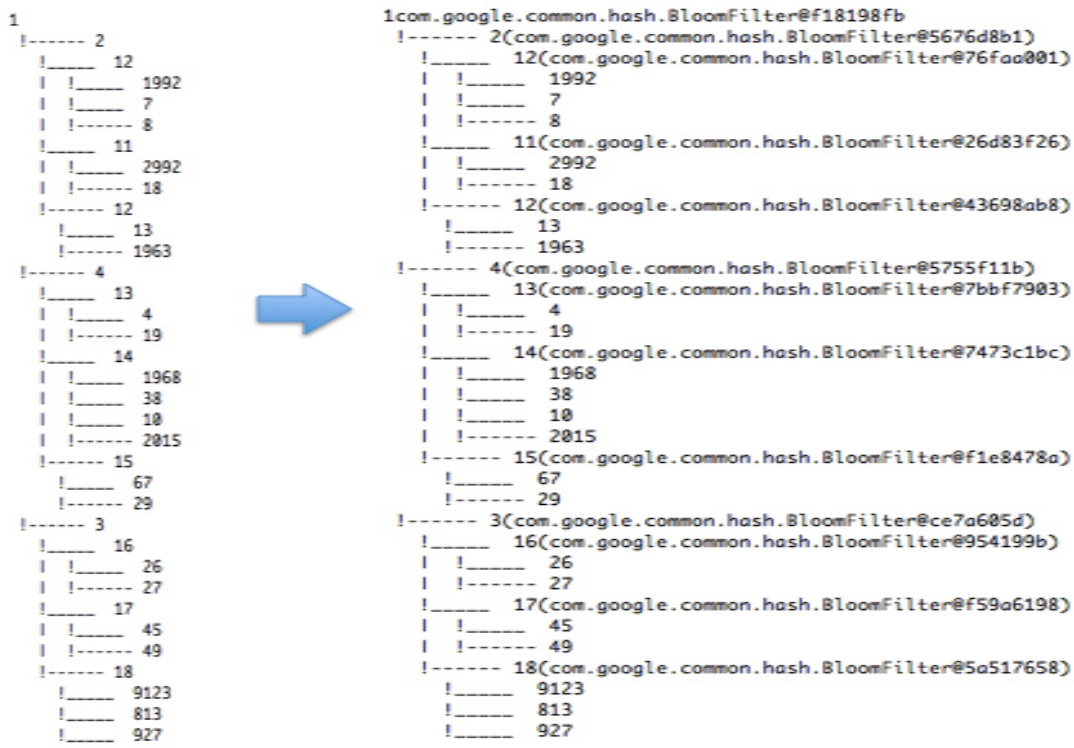

Figure 3.1 BF-Tree construction for an example data tree

The bloom filter tree structure maintains all the salient features and properties of bloom filter, simultaneously makes it easier for searching and exploring through the tree without losing the intents and advantages of the structure.

3.2 Bloom Filter Tree

The construction is based on an assumption of a data tree *T* with *n* elements, including interior nodes and leaf nodes, each leaf node is a data element and each interior node contains metadata or

possibly a data element. Given the number of elements *n* and a desired false positive rate *f*, the number of bits *m* required for the bloom filter to represent the set of elements, using the following equation [30]:

$$m = -\frac{n \ln f}{(\ln 2)^2} \tag{3.1}$$

Knowing *m* and *n*, the optimal number of hash functions *k* can be found by the following equation [30]. Optimality is in the sense of minimizing false positive probability:

$$k = \frac{m}{n} \ln 2 \tag{3.2}$$

3.2.1 BF-Tree Construction

The BF-Tree is build from bottom up. The construction is first started with groups of leaf nodes for those who are sharing the same parent node. The bloom filter for each group of elements with *n* number of has *m* bits and using *k* hash functions, which are generated based on the equations above. And then the bloom filter is stored at parent node as a new filed. After processing all the leaf nodes, parent node of leaves each has a bloom filter, which is constructed from all its leaf nodes. These firstly computed bloom filters are storing at the topologically lowest level of interior nodes. We move one level higher up and build bloom filters for these lowest level interior nodes. For the new level interior nodes, the lower nodes become the children with bloom filters. Those

nodes, who are sharing the same parent, are taken bitwise OR operation of each bloom filter to come up with a union result as the bloom filter of the parent node. Consider such an interior node *u*, let $c_i, i = 1,...,l$ be its *l* children with their respective bloom filters $b_i, i = 1,...,l$. The set of data represented by *u* should be the union of all the data sets represents by $\{c\}_1^l$. The bloom filter for *u* is the bitwise OR of all its children's bloom filters, b_1 OR b_2 OR ... OR b_l. The bloom filter construction is computing for each level of interior nodes of the tree from bottom up until it is computed and stored at the root node.

Algorithm 1: Constructing the BF-Tree

```
Function buildBF (node u) {
u.bf := create-bloom-filter(n,m,k)
if  u's children are leaf nodes then
        for each child c of u do
            add data element c to u.bf
        end
end
else
    u.bf := bitwiseORc child of u buildBF(c,n,m,k)
end
}
```

3.2.2 BF-Tree Search

According to the element data for each node in the co-existing bloom filter tree, check whether *x* exists in the tree by beginning at the top, seeing the root's bloom filter. If it does, in contrast to

the construction, we go down one level to check each bloom filter of root's children, which are topologically the highest level of the interior nodes. At this time, we are not checking whether this node's bloom filter contains *x* but to locate a subtree which contains *x*. If the check yield positive, then we know that most likely *x* is in the subtree and proceed to keep checking in this subtree top-down, in the same fashion. If the check yield negative, then we know with certainty that *x* is not in this subtree and can prune out this subtree. The pruning out subtrees makes the saving of unnecessary cost possible. By simply checking the bloom filter, we can eliminate the possibility that *x* is in a subtree and altogether forego the traversal in a particular subtree. Those subtrees not having *x* doesn't get traversed at all. The construction of co-existing BF-Tree structure doesn't break or waste the original data tree structure, thus throughout this top-down pruning search process, the path that leads to *x* from the root is obtained and this path gives not limited to the location but also the metadata information which may be encapsulated in the tree organization as vital information.

Because of the property of construction, the search for a particular data element can also return the siblings of *x* in a straightforward way. Throughout the whole top-down search process, the final stop is one interior node which is topologically at the lowest level. Since the co-existing BF-Tree is just a new filed of the original tree, after locating this final interior node which most likely contains *x*, it is simple to get the set of all the leaf nodes of this interior node, which are the siblings of *x*.

Algorithm 2: BF-Tree Search

Function lookFor (node u, data element x, result) {
if *u's children are leaf nodes* **then**
    if *x is in u.bf* **then**

```
            append u in result
            return
        end
        else
            return
        end
end
else
        if x is in u.bf then
            append u in result
            for c in u's children do
                lookFor(c,x,result)
            end
    end
end
}
```

Algorithm above is a recursive function of searching for a particular data element *x*. If *x* is in the bloom filter at the interior node *u*, then we recursively for *x* in each of the children of *u*; otherwise, prune out the subtree rooted at *u* and don't search it any further. Every time we find *x* in a bloom filter, *result* is recorded with the visited interior nodes.

3.3 Optimal Placement in a Sparse Bloom Filter Tree

We now consider the problem of using a sparse bloom filter tree instead of the full one in which every node in the data tree in indexed by a bloom filter. In order to reduce the amount of overhead associated with computing and maintaining a full bloom filter tree, we can choose to

index only a subset of the nodes. The problem becomes: Given the number of bloom filters *k* that we wish to generate, how do we find an optimal placement for them in the tree? Optimality is defined in terms of search efficiency, or equivalently, by the total number of nodes scanned or accessed before the target item if sound or deemed to not exist in the tree. Incidentally, our analysis also yields the exact reduction in search complexity achieved by adding *b* bloom filters to the tree.

To calculate the search complexity, we need the parameters: n = total number of nodes in the tree; q = probability of any given data item being searched for is in the tree; α=number of children each interior node has.

The reason we only need to consider the cases of $b = 0, 1, 2$ is because the analysis process can be thought of as being recursive, with each of the subtrees rooted at level 1 nodes (children of the root) being treated recursively as a tree.

We begin with the cases of $b = 0, 1$, that is, the cases of using no bloom filter and using only one. For search complexity, we use the expected number of nodes scanned in the search. For $b = 0$, whether the data item being searched for is in the tree or not, the expected number of nodes scanned is the same:

$$q \cdot \frac{n}{2} + (1 - q) \cdot n$$

For $b = 1$, we first calculate the search complexity of placing the bloom filter at the root node. If the data item is in the tree, the search complexity remains the same as above; but if it is not, then the bloom filter at the root would indicate so and eliminate the search altogether.

$$q \cdot \frac{n}{2} + (1-q) \cdot 0 = q \cdot \frac{n}{2}$$

We observe that by adding just one bloom filter (at the root), the expected number of nodes scanned is reduced by $(1-q) \cdot n$. In the uninformed case of $q = \frac{1}{2}$, search is improved by scanning $\frac{n}{2}$ fewer nodes on average.

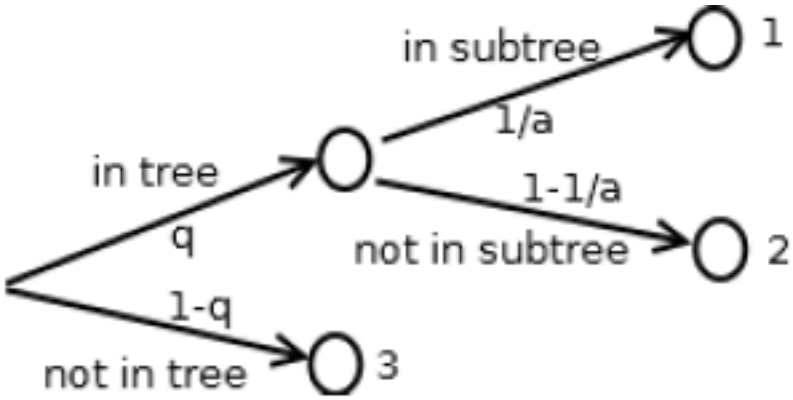

Figure 3.2 Bloom filter at a level-1 node

Suppose instead, we place the bloom filter at a level-1 node (i.e., one of the children of the root). The decision tree in 1 shows the three different possible outcomes, which are at the leaf nodes of this decision tree. For instance, if the data item is in the tree and in the subtree rooted at the level-1 node that has the bloom filter --- outcome 1; this results in a search of only this subtree, the expected number of nodes scanned is hence $1/2$ of the number of nodes in the subtree, $\approx (1/2)n/\alpha$. If it is in the tree but not in the subtree of this level-1 node, then it must be in one of the other $(\alpha - 1)$ subtrees of level-1 nodes --- outcome 2. The edges are labeled by the probabilities of events. For outcome 1, the number of nodes scanned is limited to searching one subtree --- the one rooted at the level-1 node with the bloom filter. For outcome 2, the bloom

filter will indicate that the item is not in that subtree, so all the remaining subtrees will be searched until the item is found. Since the bloom filter is at a level-1 node, outcome 3 will require scanning every single node in all the subtrees buy one, until it is concluded that the item is not in the tree. The expected number of nodes scanned is therefore given below:

$$q \cdot \frac{1}{\alpha} \cdot \frac{n}{2\alpha} + q \cdot \frac{\alpha - 1}{\alpha} \cdot \frac{n(\alpha - 1)}{2\alpha} + (1-q) \cdot \frac{n(\alpha - 1)}{\alpha}$$
$$= q \cdot \frac{n(1 + (\alpha - 1)^2)}{2\alpha^2} + (1-q) \cdot \frac{n(\alpha - 1)}{\alpha}$$

To compare with zero bloom filter, we subtract this from the other to obtain:

$$q \cdot \left[\frac{n}{2} - \frac{n(1 + (\alpha - 1)^2)}{2\alpha^2} \right] + (1-q) \cdot \left[n - \frac{n(\alpha - 1)}{\alpha} \right]$$
$$= q \cdot \frac{n(\alpha - 1)}{\alpha^2} + (1-q) \cdot \frac{n}{\alpha}$$
$$\approx q \cdot \frac{n}{\alpha} + (1-q) \cdot \frac{n}{\alpha}$$

We compare selection of level-1 node with root node for placing the bloom filter, to obtain the difference:

$$-q \cdot \frac{n(\alpha - 1)}{\alpha^2} + (1-q) \cdot \frac{n(\alpha - 1)}{\alpha}$$

If this expression evaluated to a positive number, then placing the bloom filter at the root performs better than placing it at a level-1 node; otherwise, the level-1 node is the better choice. It is interesting to note that with different values of q, it may be one or the other. The root does not always give the better performance. For example, letting $q = 0.8$ and $\alpha = 3$, the above expression evaluates to a negative number, meaning placing the bloom filter at a level-1 node gives better performance that at the root. We draw **the conclusion: If it is know a priori that the search item is likely to be found in the tree, then it is better to place the bloom filter at a level-1 node, not at the root. Otherwise, it is optimal to place the bloom filter at the root. (Conclusion I)**

Generalizing this, we obtain the search complexity for placement at level-*k* node. The probabilities on the right-most two arrows in 1 would change to $1/a^k$ and $1-1/a^k$, respectively. Therefore we have the following:

$$
\begin{aligned}
& q \cdot \frac{1}{\alpha^k} \cdot \frac{n}{2\alpha^k} + q \cdot (1 - \frac{1}{\alpha^k}) \cdot (\frac{n}{2} - \frac{n}{2\alpha^k}) + (1-q) \cdot (n - \frac{n}{\alpha^k}) \\
& = q \cdot \frac{n((\alpha^k - 1)^2 + 1)}{2\alpha^{2k}} + (1-q) \cdot \frac{n(\alpha^k - 1)}{\alpha^k} \\
& = q \cdot n \left[\frac{(\alpha^k - 1)^2}{2} - (\alpha^k - 1) \right] + \frac{n(\alpha^k - 1)}{\alpha^k}
\end{aligned}
$$

For increasing *k*, this quantity clearly increases since the $(\alpha^k - 1)$ term is easily dominated by the $(\alpha^k - 1)^2 / 2$ and $(\alpha^k - 1)/\alpha^k$ terms. In brief, this quantity achieves the minimum value at *k* = 1 over all values for $k >= 1$. **The conclusion is that bloom filter placement at a level-1 node is always better than at a level-*k* node for any $k > 1$. (Conclusion II)**

Next, we consider the case of $b = 2$ bloom filters to be placed in the tree. We compare the search complexity for (i) 1 at root and 1 at level-1, and (ii) 1 at root and 1 at level-2. Using the decision tree reasoning as above, we obtain the following:

$$(i): q \cdot \frac{1}{2} \cdot \frac{n^2}{\alpha^2}\left[1+(\alpha+1)^2\right]$$
$$(ii): q \cdot \frac{1}{2} \cdot \frac{n^2}{\alpha^4}\left[1+(\alpha^2-1)^2\right]$$

Subtracting them, (ii) – (i), gives the difference:

$$n^2\left[\frac{1+(\alpha^2-1)^2}{\alpha^4} - \frac{1+(\alpha-1)^2}{\alpha^2}\right]$$
$$= n^2\left[\frac{2(\alpha^3-2\alpha^2+1)}{\alpha^4}\right]$$

Since $\alpha^3 - 2\alpha^2 + 1 > 0 \forall \alpha \geq 2$, placement (ii) always has a larger search complexity than (i), hence **the conclusion is that it is more optimal to place the first bloom filter at root and the second one at level-1 than to skip a level for the second one. (Conclusion III)**

Now we put our conclusions together. To be most generally applicable, we assume a *q* value that is not too lopsided. Conclusion I requires that we place a bloom filter at the root. Conclusion III indicates that the next bloom filter should be placed at a level-1 node for optimal search efficiency. Applying Conclusion II, the next bloom filter should be placed at another level-1 node, and this is applied until all but one of the level-1 nodes have bloom filters assigned (the last

one does not need one, since the complement of all its siblings serve the same purpose). Then, we apply these recursively to each of the subtrees rooted at the level-1 nodes. Because the equivalent q value for each level-1 node (root of their own subtree) is known to be small (it is size of one subtree over size of all subtrees), Conclusion I dictates that we place bloom filters at the level-2 nodes (all except one). And so on and so forth, until we reach the quota of b bloom filters in total to be placed. Based on our analysis, this incurs the minimum search complexity and thus optimum search efficiency.

3.4 Experiments

The experiments are implemented in Python. For the data tree, we randomly generated data elements for the leaf nodes and the interior nodes of the tree using parameters of tree height h, number of children for each interior node at level less than h, and the number of leaf nodes for each interior node at level h. Then using the generated data tree as input, the co-existing bloom filter tree was constructed as the construction algorithm. For comparison, the naïve method of tree traversal, breadth-first [31] and depth-first [32] methods were implemented.

The first one is comparing the search time of using the bloom filter tree and of doing the naïve tree traversal. For each data tree, we randomly selected 1,000 data elements and run the search for them by using both methods. For this experiment, the tree size was around 20,000 nodes with height of 7. The search time histogram is shown in Figure 3.2.

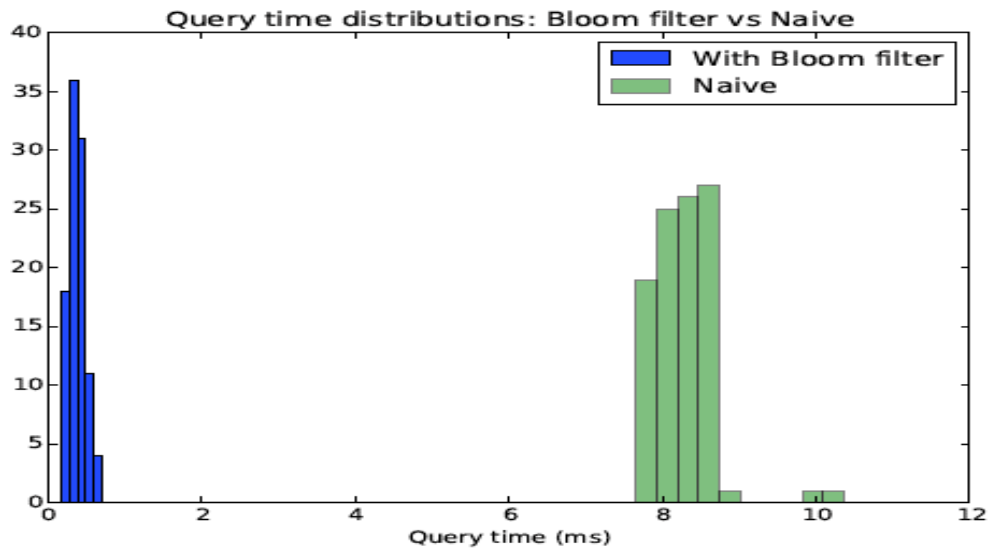

Figure 3.3 Histogram of query time distribution, with bloom filter versus naïve method

As can be seen in Fig. 3.2 the search time for the BF-Tree is consistently and substantially more efficient than the naïve method.

Another way to do the evaluation is to measure the average query time over thousands of queries. We varied the tree size from 10,000 to 30,00 and measured the average query time over 1000 queries. And then we plot the lines for both methods as displayed in Figure 3.3. One may observe that with bloom filter index, the query time is improved by 1 to 2 magnitudes.

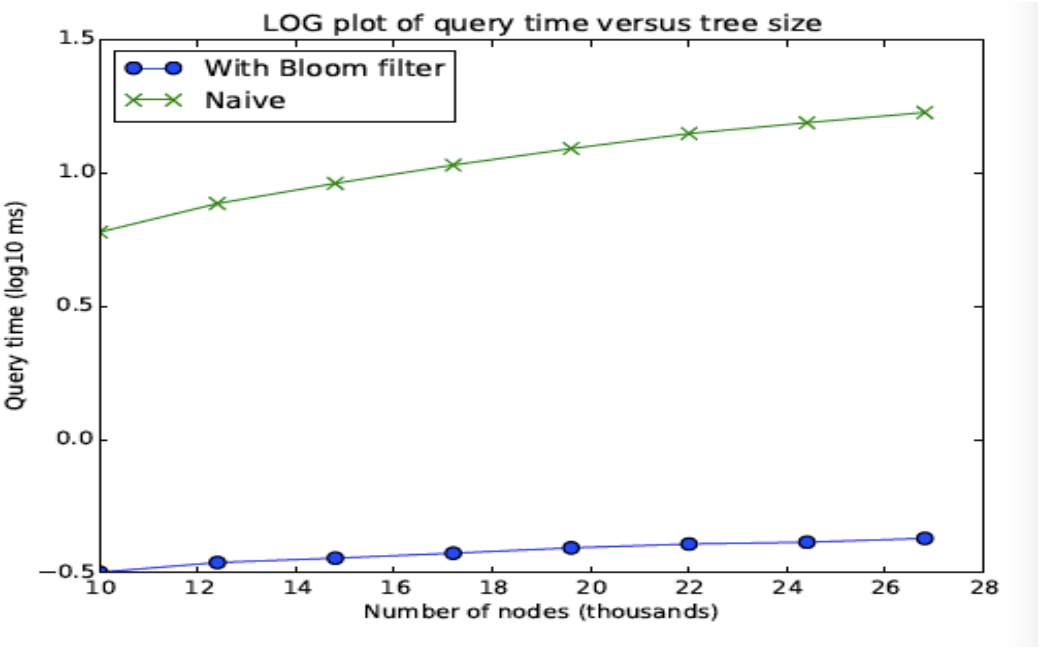

Figure 3.4 Varying tree size versus query time

To investigate the robustness of our method, we measured the average query time for trees of varying structures. The experiment was conducted with shallow trees to deep trees, varying the height from 2 to 10. The log plot of query time is shown in Fig. 3.3. Not only is our method more efficient in query time, but also as the tree height increases, the gap grows even more rapidly.

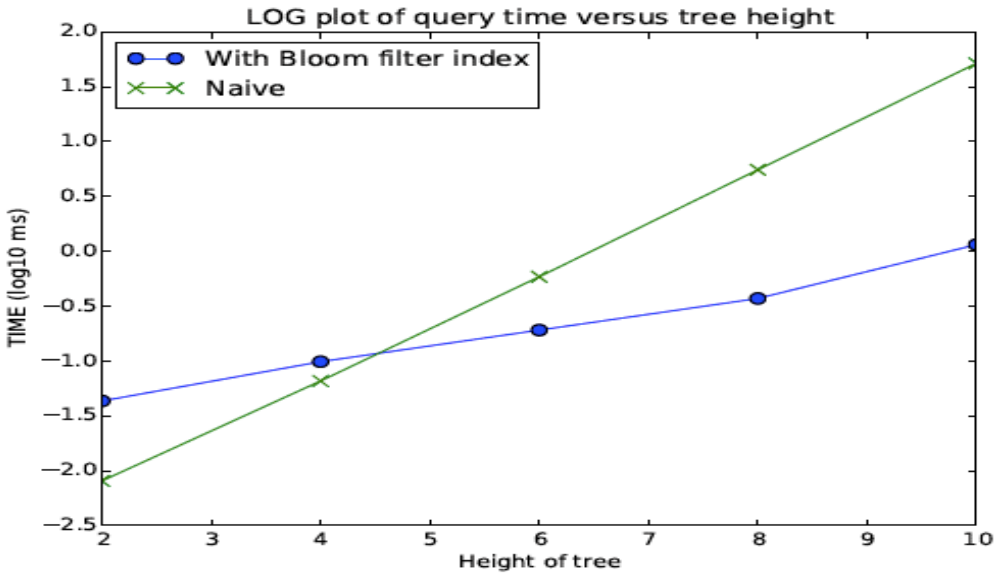

Figure 3.5 Effect of tree height on query time

We also considered the scenario of selectively placing bloom filters in the tree. If the tree is really deep and dense, there is no need to compute all the bloom filters for interior nodes for every level. The construction for the BF-Tree starts from the building the lowest-level interior nodes, which already contains all the information of the leaf nodes. Thus, if the concentration of the application is not strictly about the full path of finding a particular element, selectively placing bloom filters in the tree is a brilliant choice. We investigated the performance when we change the proportion of nodes that have bloom filters. The proportion various from sparsely indexed to densely indexed trees, numerically from 0 to 1. As we can see in Fig. 3.5 even with only 40% of the nodes indexed, our method outperforms the naïve one; and as in the previous experiment, the improvement grows with the density increase.

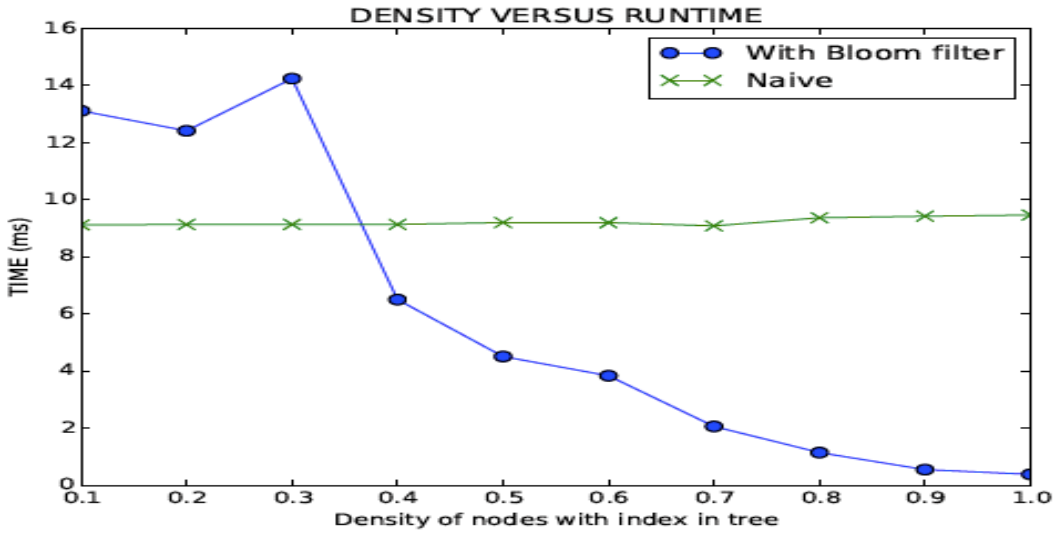

Figure 3.6 Density of indexed nodes versus query time

Figure 3.7 Effect of types of querying the work load on query time

In the experiments presented so far, all the data items for searching are already known in the tree. In real life, we probably miss the queries, namely queries that do not match any data in the tree.

Given the bloom filter property, we have greater pruning power with missed queries, thus yielding better performance.

Fig. 3.6 shows the performance for workloads consisting of various proportions of missed queries. As expected, the higher the ratio of missed queries, the more efficient the search time our method exhibits.

3.5 Search Method Implementation as an Android app

To better test the search method based on our BF-tree, we implement the algorithm onto an android application for a practical purpose. The design of this search app is about searching a particular single keyword in a tree-structured text data set and to get the paths and the whole text data that are associated with the keyword.

As an illustrative example in real life, suppose a collection of text data. These could be documents or forum posts or just pieces of text that users contribute to a common repository. The data is naturally organized as a tree structure, the children of the root node could be the different content types, the next levels down may represent the timestamps, and further down the breakdown may be by ID, etc. The leaf nodes contain the actual text data. To search a particular text items, one can build a BF-tree and use the BF search method.

The DBLP records are stored in huge XML file and are open to be downloaded for testing. Digital Bibliography and Library Project or DBLP has been evolved from a small experimental Web server in 1993 to a computer science community which consists of millions of bibliographic records; these records represent bibliographic details of variety types of articles which have been published in journals, conferences, Web pages … etc [33]. DBLP dataset records have eight different elements of publications and for each main element there are sub elements which provide information regards to any publication such as authors, year, title, journal, pages … etc.

These eight elements are all connected to an article, thus we decide to add one more sub element to store the actual text data and let these existing elements to be the interior nodes levels.

According to the XML format of DBLP, we enrich the DBLP with one more element tag called “abstract”, in which we put related abstracts. An example has been given in Figure 3,7. The contents under this element tag will be the leaf nodes in the tree structure.

```
<Article id="6" group="6">
    <author>You-Chiun Wang</author>
    <title>Mobile Sensor Networks: System Hardware and Dispatch Software.</title>
    <pages>12:1-12:36</pages>
    <year>2014</year>
    <volume>47</volume>
    <journal>ACM Comput. Surv.</journal>
    <number>1</number>
    <url>http://doi.acm.org/10.1145/2617662</url>
    <abstract>Wireless sensor networks (WSNs) provide a convenient way to monitor the physical environme
</Article>
```

Figure 3.8 An example in XML file

The XML file is stored under the Asset folder in the android application. Even the XML file has already been in the tree structure, we still need to parse the file in order to manipulate the data as needed. There are variant XML parsing models used to read and manipulate XML files. Such these parsers are SAX, DOM, StAX [34], and VTD parser [35]. The parsing process goes through three-step processing (Character conversion, Lexical Analysis and Syntactic Analysis) [36]. We choose SAX parser since it meets the application requirements in terms of better parsing process stages, smaller memory usage and larger file size. The SAX parser is able to parse the XML file element by element, thus after parser each element has been set as the interior nodes in the tree and the contents under “abstract” element tag are stored as the leaf nodes.

After SAX parsing, every element in the XML file finds its corresponding position in the tree structure and the leaf nodes contain the actual text data. By building the bloom filters for this tree, we can efficiently search for a particular text item and find out both the paths to this particular data item and the whole abstract contents that contain this data item. There is one thing should be clarified here is that the abstract contents are not stored in the leaf nodes directly, but firstly trimmed in a form as word by word after parsing. Since we are aiming to search one word and find out all the siblings associated with this word, we need to record all the sibling leaf nodes those under the same parent interior node as the searched one in the search algorithm.

There are three simple page layouts in the application. The first page is with a EditText widget which allows inserting input words and a Button widget which is for searching purpose. After clicking the Search Button, the current page will be forwarded to the second page. The mainly purpose of the second page is to display the paths where the keyword stores. The third page contains a TextView widget in order to display all the leaf nodes sets which contain the searched word. Our application is able to return multiple results.

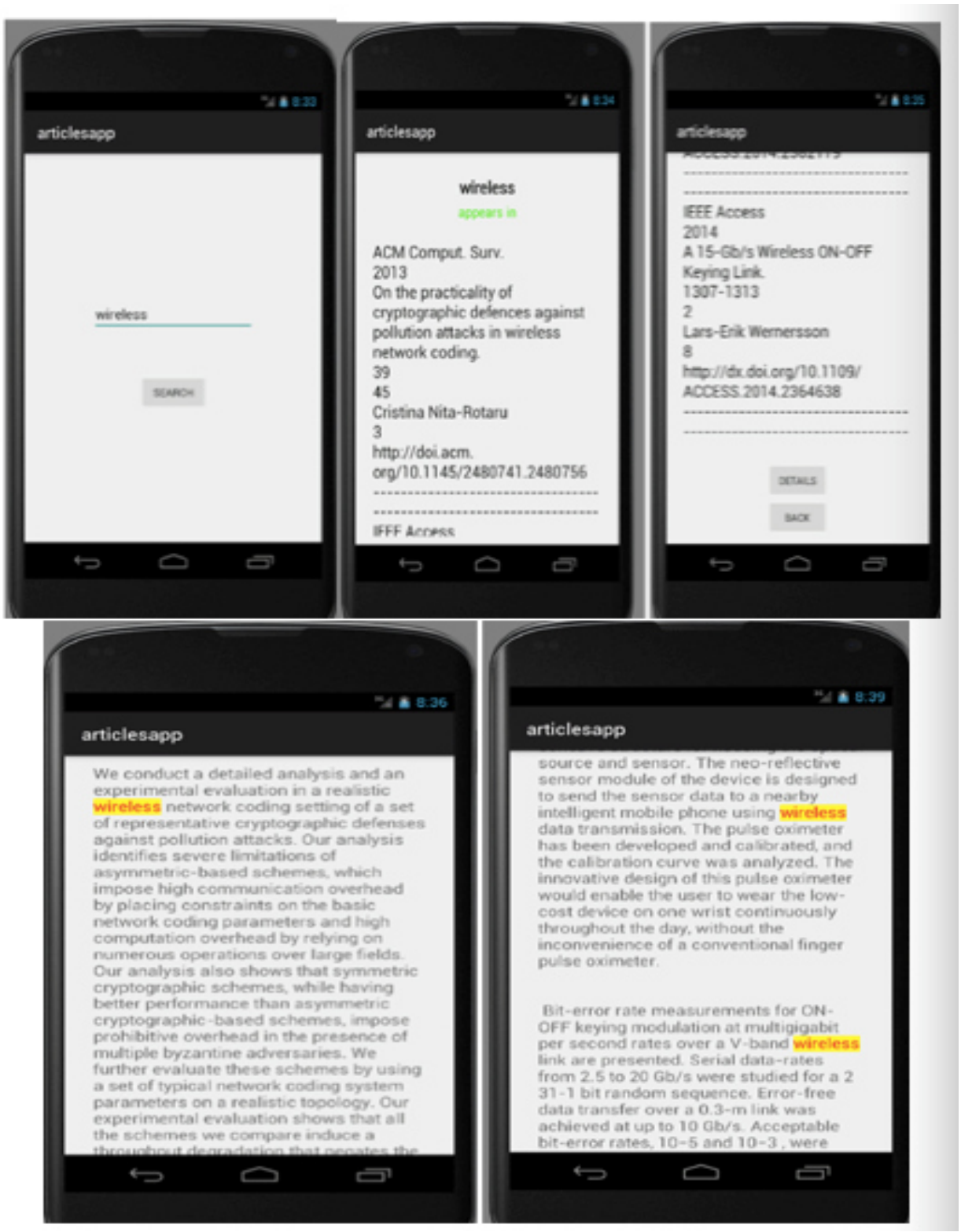

Figure 3.9 Android app screenshot

Figure 3.8 shows the screenshot of our search method android application. In the given example, the search keyword is “wireless” which is a very common word in computer science industry. After clicking the Search button, it turns out that “wireless” is recorded in more than one article and all the paths are displayed. These paths are the element tags as described in the XML file. If you click “ Details” button, you’ll get all the abstracts contains “wireless” with the searched keyword highlighted.

In the experiment, we inserted hundreds of articles in the XML file. These hundreds of articles are selectively from different research areas. We could not take all the records in DBLP dataset. It is too large for a mobile device. With the hundreds of articles recorded in the XML file, the number of total nodes in the whole tree is approximately 35,000 nodes. We have fast responses and accurate search results.

4

Tree Structure Data Synchronization using BF-Tree

4.1 The Problem

Imagine you have two tree-structure files, *A* and *B*. There are only small differences between them, and you wish to update *B* to be the same as *A*. These two files could be on different ends connected by a slow communication link. The obvious method think of is to scan all the leaf nodes of each end and translate them into two files, and then use the rsync algorithm to do the data synchronization. Suppose we have a big data tree with approximately fifty thousands nodes, ten thousands interior nodes and forty thousands leaf nodes respectively. In practice, most of time, the number of interior nodes is way smaller than the number of leaf nodes. Now assume, *A* and *B* are quite similar, perhaps both derived from the same original file. The number of nodes which have differences comparing to the number of all leaf nodes is like one out of ten thousands. This is reasonable if the synchronization between two files is performed frequently. Rsync could be a right way but scanning forty thousands nodes to locate small differences is such a cost.

We are considering two possible scenarios; one could be randomly insertion or deletion of several data element in the middle of the whole leaf nodes sets; the other is only appended differences at the end of the whole leaf nodes sets. In both scenarios, the number of changed nodes compared to the entire tree is like nothing. Thus, data synchronization using constructed BF-Tree is taking advantages of the interior nodes.

4.2 BF-Tree Sync

The synchronization starts from the comparison from top of two trees, the bloom filter of root nodes at two ends. If it yields positive, no more comparisons, we can conclude two files are identical. At this point, only the root bloom filter is transmitted through the communication link. If it yields negative, we move down to the children of root, comparing each node's bloom filter at this level. We only continue exploring the subtrees when the bloom filters of two roots of the subtrees are different. For the ones whose bloom filters are the same, we are with highly certainty the entire two subtrees are the same. Thus, only scanning bloom filters in the first level of interior nodes helps us prune out several big subtrees, which avoids the further visiting and comparing of every nodes in those subtrees.

The algorithm efficiently computes which parts of the leaf nodes of a source data tree don't match some part of an existing destination data tree by comparing the bloom filters of interior nodes level by level top-down and gradually narrow down to the comparisons of certain subtrees. Only the leaf nodes whose parent nodes' bloom filters are not matched need to be set verbatim through the communication link. The parts need not be sent across the link have been pruned out by only transmitting its root nodes' bloom filters.

Suppose we have two general tree-structured files *A* and *B*. *A* and *B* are identical at first, and then some changes happened to *B*. However, *A* and *B* are still quite similar, and they are at two different ends and could get access to each other. We want to find the differences and make *A* to be the same as *B*. There is a slow communication link between two ends.

The data synchronization algorithm based on BF-Tree consists of the following steps:

1. Both two tree-structured data elements in *A* and *B* are parsing through the BF-Tree construction algorithm in chapter 3 to generate co-existing Bloom filter trees. And the bloom filter for each interior node is stored as a new field.
2. The root bloom filter of *B* is sending to *A*.
3. *A* takes out its bloom filter at the same location as the bloom filter received and does the comparison dynamically.
4. For each interior node at *B*, the bloom filter keeps sending to *A* top-down level by level only when the comparison for its parent node yields negative.
5. Step 3 and step 4 operate recursively until the bloom filter comes to the topologically lowest level of interior nodes.
6. The leaf nodes elements of the different interior nodes at the lowest level in *B* are sending directly to *A*, and the leaf nodes at the same location in *A* are replaced with the received data elements for constructing a copy of *B*.

Algorithm 3: BF-Tree Data Synchronization

```
Function dataSync (node u, node v) {
if  u.bf equals to v.bf  then
   return
end
else
        for each child c ,d of u,v do
```

```
            if  c.bf equals to d.bf  then
               println
             end
             else
              if  c.child is leaf || d.child is leaf  then
                  transmit the leaf nodes of c to d
             else
                  dataSync(c,d)
              end
              end
        end
end
}
```

Algorithm 3 is the pseudocode of the data synchronization process. First, we begin with the root from each end, if two roots' bloom filters are identical, then we are with highly certainty to stop further comparisons. Otherwise, we continue exploring the children at the first level of each root. Here, the comparison only happens to the nodes at the same location from each end. The bloom filter of the leftmost child of the root from one end only can be compared with the bloom filter of the leftmost child of the other end. If it yields positive, we prune out this node rooted subtrees in each end tree. If it yields negative, we proceed to keep checking in that subtree top-down recursively, in the same fashion. Unlike the search method, the procedure proceeds to be in each subtrees when the result yields negative. Meanwhile, if the node's children is leaf nodes, which means we have located the different sets, thus just transfer the whole set to the other end, and then data synchronization can be done.

4.3 Implementation and Evaluation

We test the theoretical framework above by examining two specific scenarios. The considered scenarios, one is that the differences only happen at the end of the whole tree --- this is the application scenario of append-only data sets, and the other one is that the differences are located randomly in the middle --- this is scenario of random access data insertion or modification. For comparison, we implemented the classic rsync algorithm for the data synchronization of the same groups of trees. The data synchronization algorithm based on BF-Tree is implemented in Java and the performance is evaluated with the number of nodes from approximately 5,000 to 60,000. The comparisons of these two algorithms are operated in terms of three aspects, the number of nodes visited, the number of bytes transmitted and the running time of data synchronization.

In the entire data synchronization process, the bloom filter plays a dual role. It is both a data structure being constructed, and a message being passed for comparison. When we use the bloom filter as a data structure, we may tune its parameters for optimal performance. The main tradeoffs of bloom filters are the number of hash functions used, which drives the computational overhead, the size of the filter and the collision rate. As the equations shown in Chapter 2&3, we've known that given the number of elements *n* and a desired false positive *f,* we can find the filter size *m* using $m = -\frac{n \ln f}{(\ln 2)^2}$. The size of the filter would influence the transmission size during the comparing. And given m and n, the number of hash functions *k* can be found by $k = \frac{m}{n} \ln 2$.

Throughout our experiments, we use the false positive probability *f* of 0.1 and the number of elements *n* is from around 5,000 to 60,000. For each interior nodes in the tree structure has *m* bits. As the number of elements increase, the size of the filter is increased. The bloom filter is also being passed and compared, thus in the experiment we optimize the transmission size by

calculating the SHA-1 value of the bloom filter. SHA-1 [37] uses 160-bit keys, which optimizes the transmission size and won't result in more collisions.

4.3.1 Data with Append-only Operation

During the whole data synchronization process based on BF-Tree, the number of the visited nodes depends on the number of interior nodes scanned. In contrast, the *rsync* algorithm scanned all the leaf nodes as input, thus the number of the visited nodes equals to the number of leaf nodes. In reality, a tree-structure file or a naturally organized data tree, the children of the root node may represent the different content types, the next levels down may represent the timestamps, and the further down may be some information like user ID, and the leaf nodes contain the actual text data. Thus, the number of the leaf nodes should be greater than the number of the interior nodes at most of the time. In the experiment, we built groups of trees with the total number of nodes from approximately 5,000 to 60,000, and the ratio of the number of interior nodes to the number of leaf nodes is 1:10.

For the data trees, we randomly generated thousands of data elements as the nodes, then using the generated data trees as input to compute co-existing BF-Trees as described in the construction algorithm in Chapter 3.

In the first experiment, we considered the scenario of appending the differences at the end of the whole leaf nodes set. For all the data trees, the BF-Sync procedure starts from the root and gradually narrows down to smaller certain subtrees that contain the differences. Eventually, all the visited nodes are the interior nodes related to the differences. However, the rsync algorithm scan the whole set of leaf nodes. The histograms of the number of nodes visited for these two algorithms are shown in Fig. 4.1. It is obvious that as the data tree becomes bigger, the rsync algorithm needs to visit much more number of nodes than the BF-Sync.

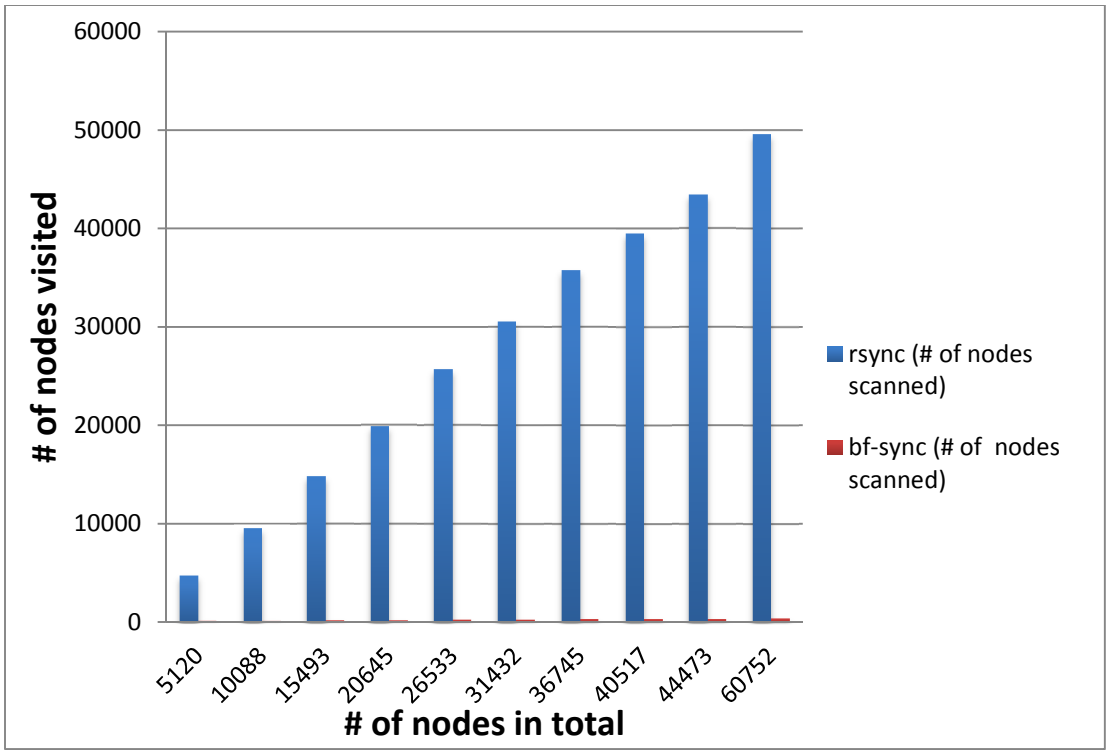

Figure 4.1 Histogram of the number of nodes visited, varying the number of total nodes from 5,000 to 60,000, rsync versus bf-sync

Under the same circumstances, the second aspect we researched is the number of bytes transmitted through the communication link theoretically. According to the above analysis, the number of bytes transmitted depends on the number of bytes of the visited interior nodes' bloom filters (here, we calculate the SHA-1 values) and the bytes of the block size where the differences locate (here, the appended differences locate in the last block). For *rsync*, the things transferred are the unmatched parts verbatimly plus the data for checksums and block indexes. We tracked and computed the number of bytes and displayed the records in Table 4.1.

Table 4.1 Table of the number of bytes transferred, varying the number of total nodes from 5,000 to 60,000, rsync versus bf-sync

# of Nodes	rsync(# of bytes transmitted)	bf-sync (# of bytes transmitted)
5120	920	602
10088	1264	717
15493	1590	867
20645	2155	962
26533	2693	1078
31432	3532	1130
36745	4062	1223
40517	4663	1271
44473	4980	1318
60752	5386	1391

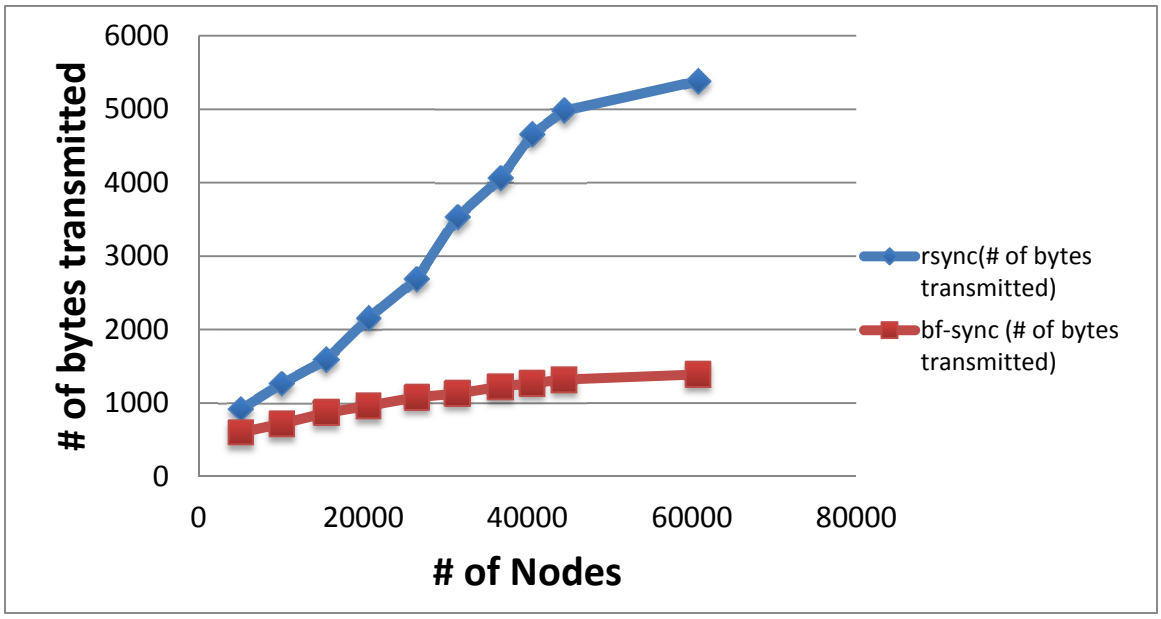

Figure 4.2 The number of bytes transferred, varying the number of total nodes from 5,000 to 60,000, rsync versus bf-sync

Table 4.1 and Figure 4.2 show the number of bytes transferred for variant number of randomly generated nodes, as a result we can notice that as the tree size becomes bigger, bf-sync has the tendency to be more stable than *rsync*.

Another way to evaluate the bf-sync algorithm is by calculating the time required to synchronize the data. The runtime records are shown in Table 4.2 & Fig. 4.3. Both of the methods perform well, but our method is more efficient as the bloom filter tree structure brings in pruning subtrees, so that less number of nodes visited results in less time spent.

Table 4.2 Table of runtime in ms, varying the number of total nodes from 5,000 to 60,000, rsync versus bf-sync

# of Nodes	rsync(runtime in ms)	bf-sync (runtime in ms)
5120	5	3
10088	7	3.333
15493	10	4.000
20645	12	3.667
26533	12	3.333
31432	14	4.000
36745	17	5.000
40517	19	5.000
44473	16	5.333
60752	17	5.667

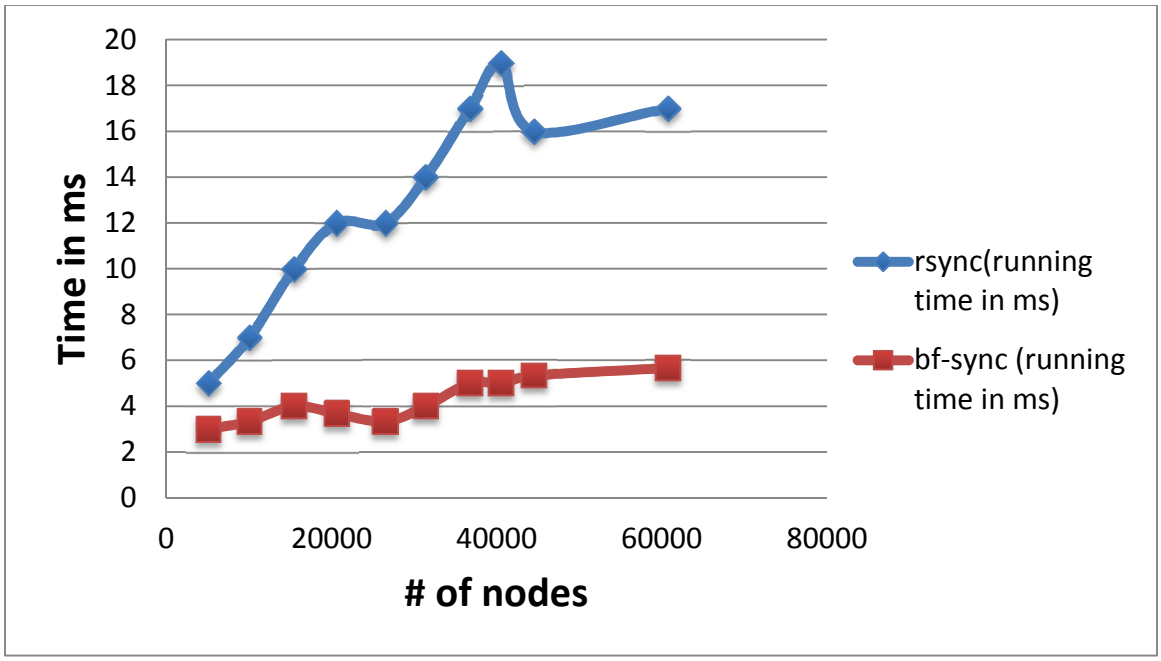

Figure 4.3 The runtime in ms, varying the number of total nodes from 5,000 to 60,000, rsync versus bf-sync

4.3.2 Data with Randomly Located Differences

The second set of experiments we did is considering the situation where the differences are randomly inserted in the middle of the whole leaf nodes list. We did the experiments from the same three aspects, the number of nodes visited, the number of bytes transmitted and the runtime.

We tested with five differences located in the middle of the whole leaf nodes set. The number of nodes visited is the number of interior nodes associated to the differences. Since the changes are only at five different locations, the number of nodes visited in bf-sync is way much smaller than the number of leaf nodes visited in *rsync*. Thus, the histogram for the number of nodes visited is quite similar to the Fig.4.1.

The number of bytes transmitted is related to the number of nodes visited. For the interior nodes, all the visited nodes' bloom filters are transmitting through the communication link for comparison in order to prune out subtrees. After locating the differences, the whole blocks where the differences located are directly transmitted. As described, we optimize the transmission size by computing the SHA-1 values of bloom filters. Thus, the number of bytes transmitted are total bytes of SHA-1 values of bloom filters of the visited interior nodes plus the bytes of all the leaf nodes sets which contain the differences. As can be seen in Fig. 4.4 and Fig. 4.5, both of the two methods have a good speed, however, with the same changes to do the data synchronization, the number of bytes transmitted for the BF-Tree is consistently and substantially more efficient than the rsync method. Especially, when the number of nodes in the tree is increasing, the BF-Tree could finish the synchronization in a fast way with less number of bytes transferred through the communication link.

Table 4.3 Table of the number of bytes transferred, varying the number of total nodes from 5000 to 60,000, rsync versus bf-sync

# of Nodes	rsync(# of bytes transmitted)	bf-sync (# of bytes transmitted)
5120	5097	566
10088	4463	711
15493	6881	831
20645	8512	926
26533	9102	1022
31432	6731	1094
36745	7791	1167
40517	7897	1215
44473	9291	1262
60752	9590	1335

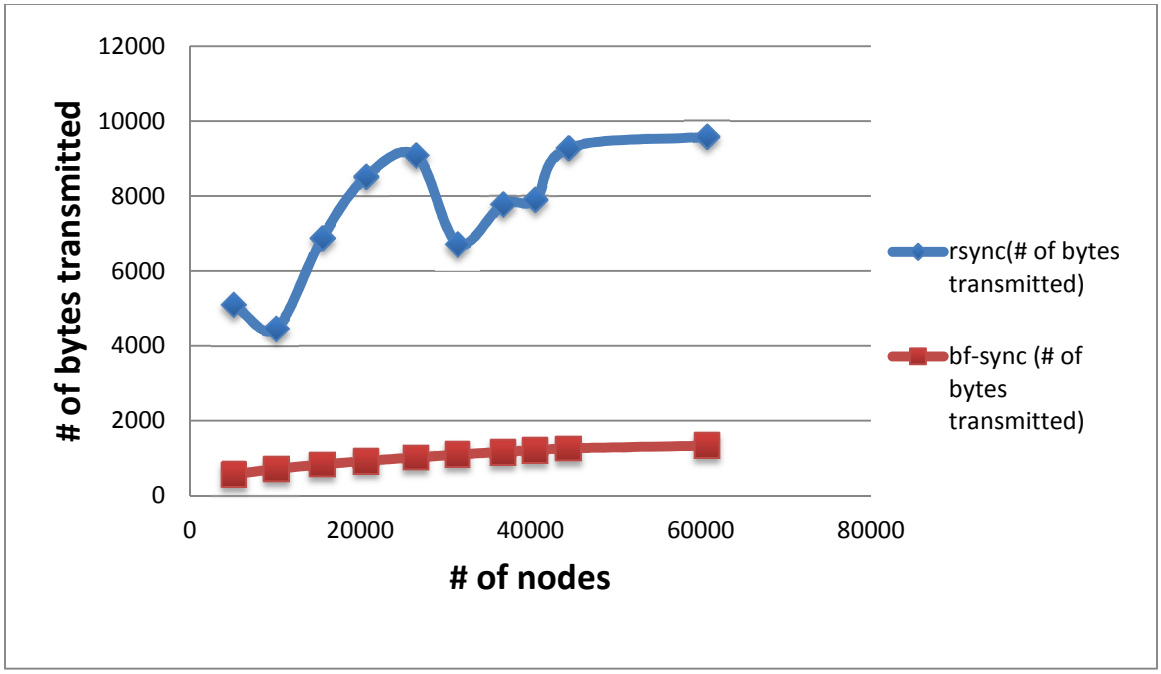

Figure 4.4 The number of bytes transmitted as the number of nodes varying, rsync versus bf-sync

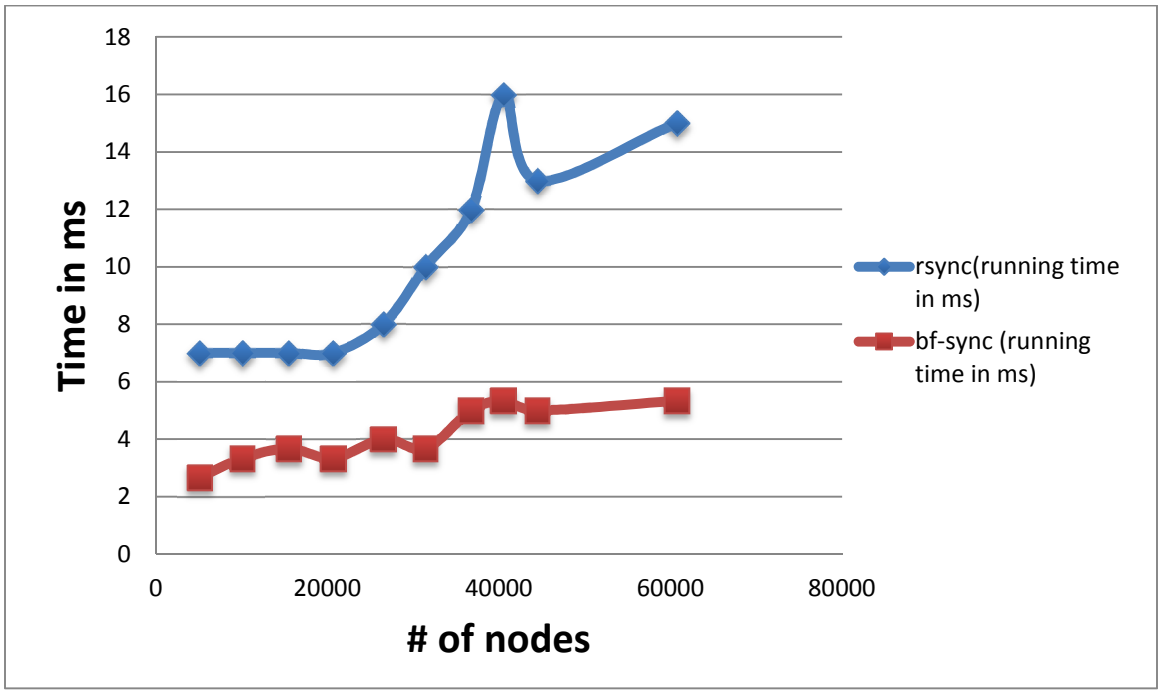

Figure 4.5 Runtime in ms as the number of nodes varying, rsync versus bf-sync

To figure out whether the number of differences inserted in the tree will influence the performance of the BF-Tree data synchronization. We did the experiment of changing the number of differences and observed how bf-sync performed. The data trees in our experiments are all big ones, the tree sizes are around five thousands to fifty thousands nodes. However, the locations of the differences will have effect on the performance. For instance, if all the differences are located densely, then the number of visited nodes won't change dramatically as the number of differences increase. If the differences are randomly located sparsely, the number of visited nodes will increase which results in the overhead of other measurements. We varied the differences size from 25 to 150 (which weighs from 0.05% to 0.3%) in the tree size of fifty thousands and measured three metrics. We insert the differences both sparsely and densely, as shown in Table 4.3.

Table 4.4

# of differences weighs	# of nodes visited	# of bytes transmitted	runtime in ms
0.05%	111	2588	3.333
0.1%	111	2629	3.667
0.15%	162	3913	4.667
0.20%	162	3976	5.000
0.25%	216	5387	5.750
0.30%	216	5477	6.500

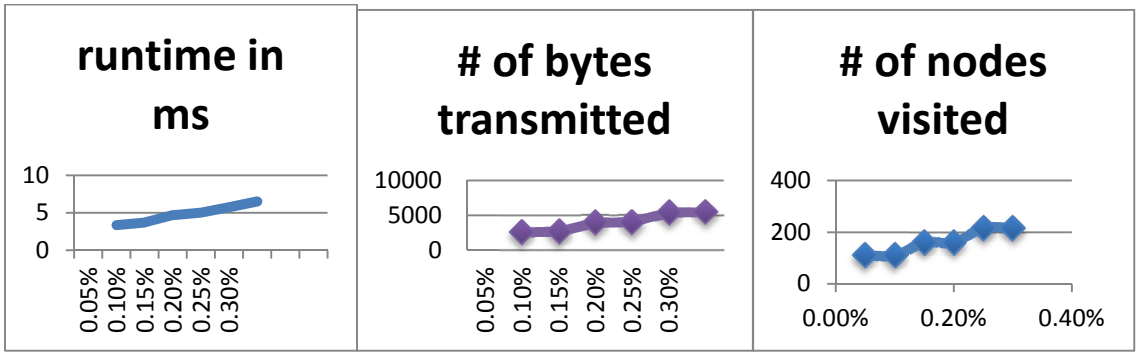

Figure 4.6 The performances of bf-sync in three aspects when the number of inserted differences varying from 0.05% to 0.3%

The reason why we choose the difference size varied from 25 to 150 is that in real life we are considering the data synchronization happens frequently or the two data files are quite similar. Thus the number of differences weights from 0.05% to 0.30% of all leaf nodes of such a big size tree is reasonable. The line graphs in Fig. 4.6 show the number of nodes visited and the number of bytes transmitted is linearly increasing with small degrees of slope as the number of differences increases, which indicates the robustness of our method for trees of varying number of differences.

4.4 Results and Performance Analysis

In this chapter, we propose a BF-Sync algorithm allowing the data synchronization between two tree-structured files. The BF-Sync algorithm is based on the BF-Tree structure introduced in Chapter 3. After constructing the co-existing BF-Trees, the synchronization process starts from comparing the two roots' bloom filters. If two roots' bloom filters are identical, then we are with highly certainty that there's no difference between two files and stop further comparisons. Otherwise, we continue exploring the children at the first level of each root. Here, the comparison only happens to the nodes at the same location from each end. The bloom filter of the leftmost child of the root from one end only can be compared with the bloom filter of the leftmost child of the other end. If it yields positive, we prune out this node rooted subtrees in each end tree. If it yields negative, we proceed to keep checking in that subtree top-down recursively, in the same fashion. During the data synchronization, bloom filter is playing dual roles, not only an efficient data structure but also the message to be compared in order to locate the differences. Thus, we optimize the transmission size by calculating the 160-bit SHA-1 values of bloom filters for comparing and transferring.

For the data tree, we randomly generated data elements for the leaf nodes and the interior nodes of the tree. Even though we are using integers as the experiment data elements and haven't yet implemented BF-Sync in the context of a full working system for an application, the required size of filter size *m* won't be influenced. Bloom filter is considered as constant space because the size does not depend on the data size of each element in the whole set. The property of using *k* independent hash functions simultaneously leads the small storage cost. We also implement the classic *rsync* scheme for remote data synchronization, for comparison and evaluation of our method. We experimented with two variants of data, one with append-only operations and the other with randomly located differences between the two trees. In each case, the bloom filter tree needs to be updated when the data items are added or changed. For case 'addition', the new data items are appended, thus the bloom filters updating is just adding new elements to their parent bloom filter which is a simply bitwise OR operation. For case 'change', the change can be data item is deleted or modified. In this case, the parent bloom filter must be re-calculated for its entire children leaf set, however, the bloom filters connecting this node in a chain up to the root node need only be re-calculated by re-doing the bitwise OR at each node, so log N bloom filters changes (only one of which is recomputed bloom filter from scratch, all the others are simple bitwise OR operations). Three metrics are used to measure the performance: number of nodes visited in the tree, number of bytes transmitted between the data sites, and the runtime. In all three metrics, our method has better performance for both variants of data.

5

Conclusion and Future Work

5.1 Conclusion

Commonly used method for data synchronization is *rsync*. It identifies parts of the source file which are identical to some part of the destination file, and only verbatimly sends those parts which can not be matched. The method computes the differences without having both files on the same machine by firstly calculating a weak rolling checksum and a strong checksum, and then using the properties of rolling checksum and the associated search mechanism to compute the differences.

Data synchronization between files is essential in a lot of scenarios in the industry and the tree structure files are also quite common. The method we provide locates the differences based on the construction of Bloom filter trees. We firstly propose an algorithm to build one co-existing bloom filter tree for the existing tree structure data file. The construction of a bloom filter tree not only maintains all of the properties of bloom filters but also applies to the specific tree structure. With the bloom filter tree, we have proposed a method to efficiently search for data items and record the searching path in data sets that are organized into hierarchical tree structures. Using the bloom filters assigned to interior nodes of the tree, the search through the tree can prune out subtrees, thereby greatly reducing search time. Another method we propose using the bloom filter trees called BF-sync allowing synchronizing data between two tree structure files. The algorithm efficiently computes which parts of the leaf nodes of a source data tree don't match some part of an existing destination data tree by comparing the bloom filters of interior nodes level by level top-down and gradually narrow down to the comparisons of two or three subtrees. Only the leaf nodes whose parent nodes' bloom filters are not matched need to be set verbatim through the

communication link. The parts need not be sent across the link have been pruned out by only transmitting its root nodes' bloom filters.

The performance of the search method is thoroughly investigated compared with the naïve method of tree traversal. Our method is clearly significantly more efficient by orders of magnitudes. What's more, we also implement the search method onto an android application. In this search application, we selectively choose hundreds of articles from a large XML file called DBLP dataset. Digital Bibliography and Library Project or DBLP has bibliographic information on published journal, articles, and conferences proceedings in the field of computer and information science [38]. DBLP dataset records have eight different elements of publications and for each main element there are sub elements which provide information regards to any publication such as authors, year, title, journal, pages, volume … etc. To better display the efficiency of our search method, we enrich the XML file with one more element tag, in which we put the abstract of each article as the actual element text. Since all the records are stored in XML format, we need to parse the XML file to build a tree for further construction of Bloom filter tree. We use the SAX parser to parse the XML file and manipulate the contents under the abstract element tag as the leaf nodes, and the rest element tags as the interior nodes or the path. With the hundreds of articles recorded in the XML file, the number of total nodes in the whole tree is approximately 35,000. 35,000 nodes for a mobile device is not small, and it still turns out a good performance of the search method.

We also did a lot of experiments to test the performance of BF-Sync algorithm. The experiments are performed comparing to the *rsync* algorithm during the whole data synchronization process in three aspects, the number of bytes transferred, the number of nodes visited and the runtime in ms. For all the experiments, the number of nodes are varying from around 5,000 to 60,000. And for each aspect, we consider two scenarios; one is the differences are randomly inserted in the middle

of the whole datasets; the other one is that the append-only differences, which means it only happens at the end of the whole set. The results shows our method is efficient and linearly stable as the number of nodes or the number of differences increase.

5.2 Future Work

In this research, we deploy an android search app with our search algorithm and the data model is an XML file, which includes hundreds of records selected from DBLP dataset. The SAX parser has been used to parse the XML file, and after parsing we just manipulate the data by putting them into the structure as wanted. This simple app works smoothly as the purpose is to implement the algorithm but for a long-lasting and sophisticated mobile application design, we should store the data with database after the SAX parsing. The SQLite database [39] is a small, compact, and self-contained database and already installed for Android devices. In the further or some more practical scenarios, the size of the data file could be very large, like millions of records. Using a database not only helps well organize the data but also saves a lot of space (e.g. the tag elements parts). Also, we will implement the BF-Sync algorithm into an app, which allows frequent data synchronizations between mobile device and the cloud or between two mobile devices. During the implementation, we should consider the problem of reducing the amount of overhead associated with computing and maintaining a full bloom filter tree using optimal placement in a sparse bloom filter tree as discussed in section 3.3.

References

[1] Shi, L., Niu, C., Zhou, M., & Gao, J. (2006, July). A dom tree alignment model for mining parallel data from the web. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics* (pp. 489-496). Association for Computational Linguistics.

[2] Tridgell, A., & Mackerras, P. (1996). ``*The rsync algorithm,''Australian National University*. TR-CS-96-05.

[3] Tarkoma, S., Rothenberg, C. E., & Lagerspetz, E. (2012). Theory and practice of bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, *14*(1), 131-155.

[4] Chen, M., Ebert, D., Hagen, H., Laramee, R. S., Van Liere, R., Ma, K. L., ... & Silver, D. (2009). Data, information, and knowledge in visualization. *Computer Graphics and Applications, IEEE*, *29*(1), 12-19.

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

[6] Cuenca-Acuna, F. M., Peery, C., Martin, R. P., & Nguyen, T. D. (2003, June). Planetp: Using gossiping to build content addressable peer-to-peer information sharing communities. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on* (pp. 236-246). IEEE.

[7] Fan, L., Cao, P., Almeida, J., & Broder, A. Z. (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, *8*(3), 281-293.

[8] Ledlie, J., Taylor, J. M., Serban, L., & Seltzer, M. (2002, July). Self-organization in peer-to-peer systems. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (pp. 125-132). ACM.

[9] Byers, J. W., Considine, J., Mitzenmacher, M., & Rost, S. (2004). Informed content delivery across adaptive overlay networks. *IEEE/ACM Transactions on Networking (TON)*, *12*(5), 767-780.

[10] Reynolds, P., & Vahdat, A. (2003, June). Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware* (pp. 21-40). Springer-Verlag New York, Inc.

[11] Chikhi, R., & Rizk, G. (2013). Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, *8*(22), 1.

[12] Liu, F., & Heijenk, G. (2007). Context discovery using attenuated Bloom filters in ad-hoc networks. *Journal of Internet Engineering*, *1*(1), 49-58.

[13] https://www.facebook.com/video/video.php?v=432864835468

[14] Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S., & Varghese, G. (2006). An improved construction for counting bloom filters. In *Algorithms–ESA 2006* (pp. 684-695). Springer Berlin Heidelberg.

[15] Ficara, D., Giordano, S., Procissi, G., & Vitucci, F. (2008, April). Multilayer compressed counting bloom filters. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE.

[16] Dharmapurikar, S., Krishnamurthy, P., Sproull, T., & Lockwood, J. (2003, August). Deep packet inspection using parallel bloom filters. In *High performance interconnects, 2003. proceedings. 11th symposium on* (pp. 44-51). IEEE.

[17] Almeida, P. S., Baquero, C., Preguiça, N., & Hutchison, D. (2007). Scalable bloom filters. *Information Processing Letters*, *101*(6), 255-261.

[18] Mitzenmacher, M. (2002). Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, *10*(5), 604-612.

[19] Shanmugasundaram, K., Brönnimann, H., & Memon, N. (2004, October). Payload attribution via hierarchical bloom filters. In *Proceedings of the 11th ACM conference on Computer and communications security* (pp. 31-41). ACM.

[20] Goh, E. J. (2003). Secure Indexes. *IACR Cryptology ePrint Archive*, *2003*, 216.

[21] Bellovin, S. M., & Cheswick, W. R. (2007). Privacy-enhanced searches using encrypted bloom filters.

[22] Tian, X., Zhang, D., Xie, K., Hu, C., Wang, M., & Deng, J. (2011, December). Exact set reconciliation based on bloom filters. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on* (Vol. 3, pp. 2001-2009). IEEE.

[23] Eppstein, D., Goodrich, M. T., Uyeda, F., & Varghese, G. (2011, August). What's the difference?: efficient set reconciliation without prior context. In *ACM SIGCOMM Computer Communication Review* (Vol. 41, No. 4, pp. 218-229). ACM.

[24] Goodrich, M. T., & Mitzenmacher, M. (2011, September). Invertible bloom lookup tables. In *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on* (pp. 792-799). IEEE.

[25] Bera, S. K., Dutta, S., Narang, A., & Bhattacherjee, S. (2012). Advanced Bloom Filter Based Algorithms for Efficient Approximate Data De-Duplication in Streams. *arXiv preprint arXiv:1212.3964*.

[26] Yang, J., Sun, X., Wang, B., Xiao, X., Wang, X., & Luo, D. (2010, May). Bloom filter-based data hiding algorithm in wireless sensor networks. In *Future Information Technology (FutureTech), 2010 5th International Conference on*(pp. 1-6). IEEE.

[27] Agarwal, S., Starobinski, D., & Trachtenberg, A. (2002). On the scalability of data synchronization protocols for PDAs and mobile devices. *Network, IEEE*,*16*(4), 22-28.

[28] Minsky, Y., Trachtenberg, A., & Zippel, R. (2003). Set reconciliation with nearly optimal communication complexity. *Information Theory, IEEE Transactions on*,*49*(9), 2213-2218.

[29] http://tutorials.jenkov.com/rsync/use-cases.html

[30] Broder, A., & Mitzenmacher, M. (2004). Network applications of bloom filters: A survey. *Internet mathematics*, *1*(4), 485-509.

[31] Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, *27*(1), 97-109.

[32] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, *1*(2), 146-160.

[33] Michael Ley, "DBLP — Some Lessons Learned", Universit ¨ at Trier, Informatik, Germany, VLDB '09, August 2428,2009, Lyon, France.

[34] Aiello, M., Platzer, C., Rosenberg, F., Tran, H., Vasko, M., & Dustdar, S. (2006, June). Web service indexing for efficient retrieval and composition. In *E-Commerce Technology, 2006. The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on* (pp. 63-63). IEEE.

[35] Xiaoji, L., Jianqiang, S., & Jinbao, C. (2009, December). VTD-XML-Based Design and Implementation of GML Parsing Project. In *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on* (pp. 1-5). IEEE.

[36] Tak Cheung Lam; Jianxun Jason Ding; Jyh-Charn Li, "XML Document Parsing: Operational and Performance Characteristics", IEEE Computer Society, Vol. 41, no. 9, pp. 30-37, 09 September 2008.

[37] https://en.wikipedia.org/wiki/SHA-1

[38]The DBLP Computer Science Bibliography, Universität Trier, http://www.informatik.uni-trier.de/~ley/db/

[39] https://docs.oracle.com/cd/E12095_01/doc.10303/e16214/soverview.htm