Set Reconciliation and File Synchronization Using Invertible Bloom Lookup Tables

Set Reconciliation and File Synchronization Using Invertible Bloom Lookup Tables

A Thesis presented

by

Marco Gentili

to

the Department of Computer Science
in partial fulfillment of the requirements

for the degree of

Bachelor of Arts in Computer Science and Physics

at

Harvard University

Cambridge, Massachusetts

April 2015

Abstract

As more and more data migrate to the cloud, and the same files become accessible from multiple different machines, finding effective ways to ensure data consistency is becoming increasingly important. In this thesis, we cover current methods for efficiently maintaining sets of objects without the use of logs or other prior context, which is better known as the set reconciliation problem. We also discuss the state of the art for file synchronization, including methods that use set reconciliation techniques as an intermediate step. We explain the design and implementation of a novel file synchronization protocol tailored to minimize transmission complexity and targeted for files with relatively few changes. We also propose an extension of our file synchronization protocol for more general file directory synchronization. We describe `IBLTsync`[1], our implementation of the aforementioned file synchronization protocol, and benchmark it against a naïve file transmission protocol and `rsync`, a popular file synchronization library. We find that for files with relatively few changes, `IBLTsync` transmits significantly less data than the naïve protocol, and moderately less data than `rsync`. In addition, we provide the first (to our knowledge) implementation of multi-party set reconciliation using Invertible Bloom Lookup Tables, a hash based data structure, and evaluate its performance for message propagation in large networks.

[1] all relevant code available at `https://github.com/mgentili/SetReconciliation`

Acknowledgments

I am infinitely grateful for the guidance of Professor Michael Mitzenmacher, who proposed this work's topic, provided me with numerous suggestions, and offered me constant and insightful feedback. CS 124 turned me into a Computer Science concentrator, and for that, I'm forever indebted.

I am also extremely thankful for the mentorship of Professor Eddie Kohler, who first introduced me to the exciting domains that lie at the intersection of systems and algorithms, and spent countless hours helping me in my quest to implement a concurrent and dynamically resizable cuckoo hashtable. I bow down to your sagely wisdom.

I would like to thank Professor Jelani Nelson for being a willing thesis reader, and teaching me all those cool algorithms in CS 224. I don't know when I'll next use y-fast tries, but man, they're awesome.

I'd also like to thank my friends for making my four years here at Harvard an amazing experience. Two people in particular, David Ding and Alisa Nguyen, deserve special thanks for taking the time to read through (at least most!) of my thesis.

Lastly, I would like to thank my mom, dad, and brother for always being there for me – you're the best.

Contents

Chapter 1

Introduction

1.1 Motivation

For decades, computer scientists have explored methods for establishing consistency among data from different sources [15, 28, 29]. One of the earlier such applications involved Personal Digital Assistants (PDAs), first invented in 1984 and popularized in the late 1990s. PDAs allowed users to synchronize data between their mobile devices (PDAs) and their home computers. Initial data synchronization methods used by many such devices involved wholesale transfer of data from one device to the other, a bandwidth-expensive process. As that step became more and more of a bottleneck in performance, computer scientists began researching methods that could take advantage of data similarities between source and destination to decrease the amount of data transferred [37].

Fast-forwarding to the present, where cloud file storage systems like Dropbox [1] and distributed databases like Cassandra [2] have become a part of our everyday lives, finding efficient methods for data synchronization is becoming increasingly relevant. In the case of Dropbox, files accessed from multiple different machines, each of which may keep local copies, and may periodically be disconnected from the Internet, must be kept in synchrony. In the case of Cassandra, updates made at one node of a distributed database, with datacenters potentially located across the globe, must eventually get replicated across all datacenters.

1.2 Model

We can pose the problems faced above, of establishing consistency among sets of objects across remote machines, in the more general framework of *set reconciliation*, which we describe below.

Say that we have two sets, S_A and S_B, on different machines, and we want to determine the *set difference*, that is, the elements that are exclusively in S_A and the elements that are exclusively in S_B (we denote these as $S_A \setminus S_B$ and $S_B \setminus S_A$ respectively). Set reconciliation is the problem of determining those sets while minimizing transmission complexity. In the extension to more than just two sets, we want to determine elements from each set that are not in the intersection of all the sets. That is, we have sets $S_1, S_2, \ldots, S_m$, and letting I denote $\cap_{i \in 1 \ldots m} S_i$, we want to determine

$$(S_1 \setminus I), (S_2 \setminus I), \ldots, (S_m \setminus I)$$

In this thesis, we focus on set reconciliation without prior context such as log files. While set reconciliation can be done efficiently with logs in some situations[1], using logs is not always possible or necessarily the best solution. For instance, if we only need to perform set reconciliation operations infrequently, or if certain data items are written particularly often, then the overhead for every update to the log may not be worth it. Also, if multiple parties need to maintain consistent state, logs could result in redundant communication when parties receive the same update from many other parties. Lastly, logs usually require stable storage and mostly synchronized time, which are not always available on routers and other such networking devices.

Mapping the problem Dropbox faces to this framework, we can think of Dropbox-enabled devices as the different machines, and all of the files within those devices's Dropbox directories as the sets to be reconciled. For Cassandra, we can think of different database servers as the machines, and the series of database operations applied at each server as the sets to be reconciled (we would need some method to preserve ordering, for instance, by appending a unique identifier to each operation).

[1] To use a log, both parties must start with the same data set and thereafter log all changes

1.3 Proposed Solution

We consider a particular data structure called an Invertible Bloom Lookup Table (IBLT) [21] that is able to take two sets of fixed-length keys and find the set difference using space within a constant factor of optimal[2]. We will explain IBLTs in more depth in Chapter 2, but we cover the basics here.

Structure-wise, we can think of an IBLT as similar to a hash table[3] except with three main differences. First, unlike in a hash table, each key in an IBLT must be of fixed length. Second, instead of each bucket holding a unique key, each bucket holds a sum of keys. Third, instead of inserting each key exactly once to the hash table, each key in an IBLT is inserted multiple times at different indices. Since one bucket can hold the sum of multiple keys, we cannot always directly retrieve a key from an IBLT. Using a clever trick involving the repeated removal of keys from buckets that only have one key, we can in fact retrieve all the keys with high probability, given that the number of keys inserted is below a certain threshold. We can use IBLTs for set reconciliation by noting that removal of keys can proceed in the same manner as insertion, except subtracting the key instead of adding it. After inserting one set of keys and removing the other set of keys, we will only retrieve keys in the set difference.

1.4 Applications

Invertible Bloom Lookup Tables have many applications outside of just pure set reconciliation.

1.4.1 Individual File and File Directory Synchronization

In this thesis we demonstrate how IBLTs can be used for individual file synchronization, which involves exchanging similar strings held by different hosts, ideally with

[2] the minimum transmission complexity is proportional to the size of the set difference multiplied by the object size

[3] A basic hash table is an array-based key-value store that uses a hash function (which takes a key as input) to compute an index into an array. That index in the array will contain the specific key-value pair.

minimal transmission complexity.

We also show how to combine set reconciliation and individual file synchronization techniques for file directory synchronization. Instead of just trying to ensure that one particular file is the same on two machines, we now want to maintain synchrony between directories of files on local and remote machines. To do so, we can think of all the files in one machine's directory as a set of files. Using set reconciliation, we can determine the files that are only on one machine or the other. Individual files that differ can then be reconciled using individual file synchronization techniques.

1.4.2 Gossip Protocols

Invertible Bloom Lookup Tables have also been proposed as building blocks for gossip protocols, which are communication protocols that allow for robust message propagation in large networks [31]. Message propagation protocols in such settings need to be 1) very simple, since each node has limited computational power, 2) distributed, as there is no notion of centralized control, 3) robust, as the network may be unstable, and 4) efficient, as bandwidth is limited. In this thesis, we describe a protocol satisfying those constraints that uses multi-party IBLTs as the transmitted message.

1.4.3 Other Applications

Invertible Bloom Lookup Tables are also useful in networking applications such as link-state database[4] synchronization or deduplication [18, 25], which involves identifying keys in the intersection of multiple sets of keys so that duplicate data can be replaced with pointers. Deduplication is commonly used to improve the efficiency of backups [41].

IBLTs can also be used for synchronizing results generated from multiple actors acting in parallel on similar tasks [18]. As an example, a search engine might have two independent crawlers that use different techniques to search URLs. The resultant sets

[4] essentially a computerized representation of the topology of a system – in such a diagram, routers are nodes, and connections are lines. In essence, the link-state is the graph representing the network

of URLs are probably quite similar, so IBLTs could be used to efficiently determine the ones unique to each.

In opportunistic ad hoc networks, which are networks characterized by low bandwidth and intermittent connectivity to peers, such as in military settings, IBLTs could be used to synchronize data when connectivity is up.

IBLTs have even been proposed for use in privacy-enhanced methods of comparing compressed DNA sequences [17]. More recently, Gavin Andersen, the chief scientist of the Bitcoin Foundation, proposed using Invertible Bloom Lookup Tables for more efficient block propagation [3].

1.5 Outline

Our goals in this thesis are four-fold:

1. Introduce the set reconciliation problem and provide an overview of various proposed methods to solve it

2. Explain in depth how an Invertible Bloom Lookup Table, one such set reconciliation method, works

3. Show how IBLTs can additionally be used for file synchronization and as a data structure for gossip protocols

4. Provide an empirical evaluation of IBLTs in the two contexts mentioned above

The organization of the following chapters reflects those goals. Chapters 2 and 3 provide background on set reconciliation and file synchronization, respectively. Chapter 4 explains our novel file synchronization protocol using IBLTs, `IBLTsync`, and proposes an extension for directory synchronization. Chapter 5 provides an experimental evaluation of `IBLTsync`. Chapter 6 introduces gossip algorithms and provides an evaluation of IBLTs as a building block for such algorithms.

In more detail, the organization of the thesis is as follows:

In Chapter 2, we begin by more formally posing the problem of set reconciliation. We categorize approaches into methods that are exact (always returning the correct set difference) and methods that succeed with high probability (which have a small chance of returning the incorrect result), and describe a few methods from each. In particular, we provide an in-depth explanation of two techniques, one using characteristic polynomials and one using Invertible Bloom Lookup Tables (IBLTs), that only need to transmit messages proportional to the size of the set difference. We also consider an extension of IBLTs, aptly called multi-party IBLTs, that are able to simultaneously reconcile sets from $n > 2$ parties. Many set reconciliation techniques rely on having a tight upper bound on the size of the set difference. We discuss various approaches to this problem, and explain a bit of the mathematical basis behind each.

In Chapter 3, we consider the problem of file synchronization, and explain our assumptions and setup. We discuss current techniques, which can be divided into two categories, single-round and multi-round protocols, and discuss the trade-off in computation and transmission complexity between both categories of approaches. We then explain a commonly used single-round protocol, `rsync`, in depth. After, we discuss approaches that use set reconciliation as a part of the file synchronization protocol, going over the appropriate mathematical tools (such as de Bruijn digraphs and context-dependent file partitioning) when necessary.

In Chapter 4, we explain how to combine some of the above techniques into a new protocol for file synchronization, which we also implement in C++. We then go through back-of-the-envelope calculations to optimize various parameters of our protocol assuming different models of file differences, and explain where our protocol could be improved and optimized for specific workloads. We also propose an adaptation and extension of our protocol for more general file directory synchronization.

In Chapter 5, we discuss the experimental setup for the testing of `IBLTsync`, our implementation of the file synchronization protocol discussed previously, and provide results comparing it to a naïve file synchronization approach and the popular file synchronization library `rsync`. We test three different workloads, one where each character in a file is independently altered with some probability (the random error model), one where a fixed number of short blocks are altered (the block changes model), and one based on actual source code repository changes (the practical workload).

In Chapter 6, we provide background information on gossip algorithms and explain a few of the basic protocols. We then explain how multi-party IBLTs can be used with such protocols for message spreading in large, unstable networks. We finish this chapter by providing experimental results verifying the theoretical performance bounds from [31] and showing that multi-IBLTs can be useful in practice.

In Chapter 7, we conclude and discuss directions for future work.

Chapter 2

Set Reconciliation

As explained in the introduction, the set reconciliation problem involves efficiently establishing consistency among sets of objects. In this chapter, we explore the problem in more depth, as set reconciliation is an important step in our file synchronization algorithm developed in Chapter 4. We being by restating the set reconciliation problem in the case of two sets here.

We have sets S_A and S_B, on different machines, and want to determine the elements exclusively in S_A and exclusively in S_B (denoted $S_A \setminus S_B$ and $S_B \setminus S_A$ respectively), which is known as the set difference. An information-theoretic lower bound on the transmission required is the size of the set difference multiplied by the element size (which we assume to be some constant c), $O(c \cdot (|S_A \setminus S_B| + |S_B \setminus S_A|))$. Note that in this thesis we assume that our sets contain fixed-size bitstring keys.

In the following sections, we consider various methods that tackle the set reconciliation problem. In general, we can partition those methods along two axes – first, between those whose transmission cost is proportional to the size of the set difference (we call these near-optimal), and those that are not, and second, between those that are exact (always returning the correct set difference), and those that succeed with high probability (sometimes missing elements in the set difference).

Section 2.1 covers two exact methods, a naïve method, and a characteristic polynomial method, the latter of which is near-optimal.

Section 2.2 covers three methods that succeed with high probability, Bloom filters, Approximate Reconciliation Trees, and Invertible Bloom Lookup Tables (IBLTs), the last of which is near-optimal. We also explain an extension to IBLTs called multiparty IBLTs that can handle set reconciliation with more than two parties. Since we choose to use IBLTs in our file synchronization protocol described in Chapter 4, we explain IBLTs in depth.

Section 2.3 covers methods to obtain a tight upper bound on the size of the set difference, which is an important step for both the characteristic polynomial and IBLT approaches (the two near-optimal methods).

2.1 Exact Methods

2.1.1 Naïve Approach

Perhaps the simplest method to determine the set difference is for each party to send an encoding of their entire set. This requires messages with total size $O(|S_A| + |S_B|)$ and takes computation time $O(|S_A| \cdot |S_B|)$, assuming the completely naïve approach of each party scanning the other party's entire list to remove shared elements. The computation time could be decreased to $O(|S_A| + |S_B|)$ if each party sent the lists in sorted order (the pre-processing would then take $O(|S_A| \log |S_A| + |S_B| \log |S_B|)$ time. We could also achieve $O(|S_A| + |S_B|)$ computation time without any pre-processing by inserting all of the set elements into a hash table.

2.1.2 Characteristic Polynomials

Minsky, Trachtenberg, and Zippel [30] discovered a method to use characteristic polynomials for set reconciliation. A key part of their approach involves the representation of sets by their characteristic polynomials, an idea first proposed by Lipton in 1990 [26].

Let our set $S = \{x_1, x_2, \ldots, x_n\}$. Then the characteristic polynomial for S is

$$\chi_S(Z) = (Z - x_1)(Z - x_2)(Z - x_3) \ldots (Z - x_n)$$

The zeros of $\chi_S(Z)$ are exactly the elements of S, meaning that we can retrieve the elements of S if we can factor $\chi_S(Z)$. However, since a set's characteristic polynomial contains all the information in that set, transmitting the full characteristic polynomial is no cheaper than transmitting the set itself. If we know beforehand that two sets share many elements though, then we do not actually need to send the full characteristic polynomial to be able to retrieve elements in the set difference, as we will soon see.

Consider two parties A and B with sets S_A and S_B, and corresponding characteristic polynomials $\chi_{S_A}(Z)$ and $\chi_{S_B}(Z)$. Parties A and B wish to efficiently find the elements in their set difference. If we consider the quotient of their characteristic polynomials, we find something surprising:

$$\frac{\chi_{S_A}(Z)}{\chi_{S_B}(Z)} = \frac{\chi_{S_A \cap S_B}(Z)\chi_{S_A \setminus S_B}(Z)}{\chi_{S_A \cap S_B}(Z)\chi_{S_B \setminus S_A}(Z)} = \frac{\chi_{S_A \setminus S_B}(Z)}{\chi_{S_B \setminus S_A}(Z)}$$

All the terms in $S_A \cap S_B$ cancel out! If Parties A and B were able to efficiently reconstruct $\frac{\chi_{S_A}(Z)}{\chi_{S_B}(Z)}$, the ratio of characteristic polynomials, then they would be able to find the set difference. Minsky et al.'s insight was to evaluate the polynomials at a collection of points and use those to reconstruct $\frac{\chi_{S_A}(Z)}{\chi_{S_B}(Z)}$, something known as *rational function interpolation.*

More formally, the rational interpolation problem is as follows: Given bound d_1 and d_2 on the degrees of the numerator and denominator of a rational function[1] $R(Z) = P(Z)/Q(Z)$, where $P(Z) = \sum_i p_i Z^i$ and $Q(Z) = \sum_i q_i Z^i$, and a *support set* T consisting of $d_1 + d_2 + 1$ pairs of points $(x_i, y_i) \in \mathbb{F}^2$, there is a unique rational function f such that $f(x_i) = y_i$ for each $(x_i, y_i) \in T$. Though we will not go into the details of this proof, the idea is that each pair $(x_i, y_i) \in T$ implies a linear constraint:

$$p_{d_1} x_i^{d_1} + p_{d_1 - 1} x_i^{d_1 - 1} + \ldots + p_0 = y_i \cdot (q_{d_2} x_i^{d_2} + q_{d_2 - 1} x_i^{d_2 - 1} + \ldots + q_0)$$

Solving the $d_1 + d_2 + 1$ simultaneous linear equations gives us the desired polynomial.

[1] A rational function is any function such that the numerator and denominator are both polynomials

If we know the size of the set difference, $|S_A \setminus S_B| + |S_B \setminus S_A| = d$, then we have a bound on the sum of the degrees of the numerator and denominator ($d = d_1 + d_2$), but not on the individual terms d_1 and d_2. Since $d_1, d_2 \geq 0$ though, we have the loose bounds $d_1, d_2 \leq d$.

Thus, given that Parties A and B know the size of the set difference d, if they evaluate $\chi_{S_A}(Z)$ and $\chi_{S_B}(Z)$ at $2d+1$ points (that do not happen to be a root of the polynomial), then they can reconcile their sets. Minsky et al. [30] develop a protocol based on this technique and show its communication complexity is within a small factor of optimal.

The computational complexity of their technique has two dominating terms, one from the cost of evaluating the characteristic polynomials at the chosen points, and one from the rational function reconstruction. The former takes $O(|S|d)$ time, and the latter takes $O(d^3)$ time using Gaussian elimination to solve the system of linear equations. As discussed in their paper, faster methods for solving systems of linear equations are available in theory, though they do not seem to be more efficient in practice, and are significantly more complex to implement.

2.2 Methods That Succeed with High Probability

While each of the methods described above guaranteed that all elements of the set difference were found, in this section we describe methods that have a small (and tunable) chance of retrieving an incomplete set.

2.2.1 Bloom Filter

A Bloom filter [9] is a data structure that supports insert and lookup queries, and is used to store a set S of n items, $\{x_1, x_2, \ldots, x_n\}$, from a universe[2] U of size u. It consists of an array of m bits all initially set to 0, and uses k random and independent hash functions $h_1, h_2, \ldots, h_k$, each of which maps the keys to a range $\{0, 1, \ldots, m-1\}$ (thus giving us an index into one of the positions in the Bloom filter).

[2] A universe consisting of all l-bit bitstrings would have size 2^l, for instance

To insert an element x_i into a Bloom filter, we apply each of the k hash functions in turn to determine k positions in the bit array, and then set each of those k bits to 1. Looking if an element is in the Bloom filter proceeds in a similar fashion, except that we check the k corresponding bits rather than setting them. If at least one of the bits is 0, then the key is not in the set, so we return false. If all the bits are set, then we return true, though there is some false positive rate (i.e., the key is not actually in the set but we say that it is). For instance, say we have a Bloom filter consisting of 6 cells all initially set to 0 (we represent this as 000000), and have $k = 2$ hash functions. Say that key A sets bits 0 and 2, key B sets bits 1 and 3, and key C sets bits 2 and 3. After inserting A and B to our Bloom filter, the Bloom filter would look like this: 111100. If we now looked up key C (corresponded to looking up bits 2 and 3), then we would return true even though C was not in the Bloom filter.

The false positive rate ϵ depends on the number of bits used per item and the number of hash functions according to the equation:

$$\epsilon = (1 - e^{-kn/m})^k$$

Minimizing the size of the Bloom filter over k for a fixed value of ϵ, a space-optimized Bloom filter uses $k = \log_2(1/\epsilon)$ hash functions, and with those settings, can store each item using $1.44 \log_2(1/\epsilon)$ bits.

We can use Bloom filters for set reconciliation by having Party A insert all keys in its set S_A into a Bloom filter F_A, and then have Party A send F_A to Party B. Party B then looks in F_A for each of the keys in S_B. If the key is found, then Party B assumes that Party A already has the key, so it does not send the key to Party A. However, as Bloom filters can have false positives, there is a chance that Party A does not actually have the key that Party B found, which is why this method is not exact. Since looking up a key that has previously been inserted into a Bloom filter will never return false, the Bloom filter never causes Party B to mistakenly send a key Party A already has.

As shown above, for a fixed false positive rate, the space used by a Bloom filter is still proportional to the set size, not the set difference, so Bloom filters do not have transmission complexity within a constant factor of optimal.

2.2.2 Approximate Reconciliation Trees

Approximate Reconciliation Trees [12, 13] (ARTs) were developed as an extension to the Bloom filter approach, and provide another method for set reconciliation. We describe the process of creating an ART below:

Party A creates a binary tree of depth $\log u$, where u is the number of elements in the universe U. The root corresponds to the full set S_A. We will explain exactly how to correspond a set to a binary tree node in more detail later. The left and right children correspond to the subsets of S_A in each half of U, respectively. That is, the left child corresponds to $S_A \cap [0, u/2 - 1]$ and the right child corresponds to $S_A \cap [u/2, u)$. Deeper nodes are constructed in the same fashion by dividing each parent node's interval in half.

To reconcile sets, Party B creates a similar binary tree but with the keys in S_B. If the root of Party A's binary tree matches that of Party B, then we know that the sets are identical. Otherwise, Party B recursively traverses the children nodes of the roots of both trees. If an element $x \in S_B \setminus S_A$, then Party B will find that the leaf node corresponding to x in its tree is not present in Party A's tree. Thus, we can find all of the keys in $S_B \setminus S_A$.

Without any optimizations, the binary trees have $\Theta(u)$ nodes and depth $\Theta(\log u)$. However, most of the nodes in a tree represent the exact same sets. Since there are only $|S_A|$ elements in set S_A, there are actually only $O(|S_A|)$ nodes representing distinct sets, so the tree can be compressed to $O(|S_A|)$ nodes by removing edges connecting nodes that correspond to the same set.

As described so far, ARTs would provide an exact method for set reconciliation. To optimize traversal time and space complexity, we make two modifications that lead ARTs to have a small false positive rate. Our first modification seeks to solve the problem of comparing nodes (which represent sets containing $O(|S_A|)$ elements) efficiently. Our second modification seeks to decrease the depth of the tree, which currently is potentially $O(|S_A|)$. To solve the first problem, we let each set of elements correspond to a value using hashing. The hash associated with each internal node of the tree is the exclusive-or of the values of its children. This introduces a small chance of false positives, as now two different sets can have the same hash. To solve the second problem, we do not insert the keys themselves into the ART, but instead insert hashes of the keys. If the range of the hash function is at least $\text{poly}(|S_A|)$, then with high probability, the depth of the compressed tree is $O(\log |S_A|)$. Using a hash instead of the key itself again introduces a chance of false positives.

In summary, given S_A and an Approximate Reconciliation Tree for S_B, Party A is able to compute $S_A \setminus S_B$. In this situation, an ART requires $O(d \log(|S_B|))$ computation time and $O(|S_B|)$ space, where d is the size of the set difference.

2.2.3 Basic Invertible Bloom Lookup Tables

An Invertible Bloom Lookup Table (IBLT) [21] is a randomized data structure that stores a set of key-value pairs. IBLTs have the property that, under certain conditions, they can be inverted to retrieve the keys that were previously inserted. This is in contrast to Bloom filters or Approximate Reconciliation Trees, which enabled lookup but not retrieval of keys. As we use IBLTs in our file synchronization protocol in Chapter 3, we describe them in detail here.

An IBLT consists of a table T of m buckets and uses k random and independent hash functions $h_1, h_2, \ldots, h_k$, each of which maps the keys to a range $\{0, 1, \ldots, m-1\}$ (giving us an index into one of the positions of the IBLT). The number of buckets is chosen with respect to some threshold t so that inverting the table succeeds with high probability when there are fewer than t key-value pairs in the structure.

Each bucket has a `keysum` field that is the exclusive-or of all the keys that have been added to that bucket and a `count` field that is the number of keys that have hashed to that bucket. For simplicity, we only consider keys and not key-value pairs – we can easily extend the analysis to accommodate the latter by including a `valuesum` field.

Operations

An Invertible Bloom Lookup Table supports three main operations, `insert`, `delete`, and `listing`. `insert` adds a key to the IBLT and `delete` removes a key from the IBLT. For now, we assume that we only can remove keys that are already in the IBLT. Later, we will show how we can allow for removal of keys never inserted to the IBLT. `listing` is our inversion function, and lists all the keys in the structure, succeeding with high probability as long as the number of keys is below a certain threshold.

To `insert` a key x, we compute $h_1(x), h_2(x), \ldots, h_k(x)$ to determine which buckets the key should be added to. For each of those buckets, we compute the exclusive-or of the key with the existing `keysum` and increment `count`. Method 1 shows pseudocode for `insert`.

Method 1 Insert a new key to the IBLT

```
function INSERT(key)
    for i in GetKeyIndices(key) do
        $T$[i].keysum = $T$[i].keysum $\oplus$ key
        $T$[i].count = $T$[i].count + 1
```

To `delete`, we perform the exact same operations, except that we decrement `count`.

To perform a `listing` operation, we go through each of the buckets in turn, looking for buckets that have `count` $= 1$ and adding those buckets to a queue. If a bucket's `count` is not equal to 1, we skip the bucket, as in that case it is not possible that there is only one key in that bucket. We then pop a bucket off of the queue. Let the `keysum` of that bucket be y_i. Since that bucket's `count` is 1, y_i corresponds exactly to one key x, so we add x to our retrieved set. We then perform a `delete` operation with key x to remove it from the IBLT. During the process of deleting x (looking through buckets corresponding to $h_1(x), \ldots, h_k(x)$), we can see if any of those buckets now have a count of 1. If a bucket does, then we add it to the queue. We repeat this process of popping a bucket off the queue and removing the corresponding key from the IBLT until we either have listed all keys (in which case every field of every bucket has value 0), or we cannot proceed (which will happen if every bucket i

has `count` > 1). Pseudocode for `listing` is provided in Method 2.

Method 2 List all the keys in the IBLT

```
function LISTING($S$)
    queue = emptyQueue()
    for i = 0 to $m - 1$ do
        if $T[i]$.count = 1 then
            Add $i$ to queue
    while queue not empty do
        $i$ = queue.pop()
        if $T[i]$.count != 1 then
            continue
        $key = T[i]$.keysum
        add $key$ to $S$
        for $i$ in GetKeyIndices($key$) do
            $T[i]$.keysum = $T[i]$.keysum $\oplus$ $key$
            $T[i]$.hashsum = $T[i]$.hashsum $\oplus$ $H(key)$
            $T[i]$.count = $T[i]$.count - 1
            if $T[i]$.count = 1 then
                Add $i$ to queue
    for $i = 0$ to $m - 1$ do
        if $T[i]$ is not empty then
            return Failure
    return Success
```

Intuition

The rationale behind using exclusive-ors for both `insert` and `delete` is as follows: Imagine that key x has been inserted once and deleted once from the same IBLT. Then for every bucket corresponding to $h_1(x), \ldots h_k(x)$, the resultant `keysum`, which we denote $\texttt{keysum}_f$ is exactly equivalent to the `keysum` before first inserting k, which we denote $\texttt{keysum}_i$. This is the case as $\texttt{keysum}_f = (\texttt{keysum}_i \oplus k) \oplus k = \texttt{keysum}_i$.

Listing Example

We will now work through a simple example to see how the `listing` process works in practice. Let us consider an IBLT T with $k = 2$ hash functions, $m = 4$ buckets, and bitstring keys of length 4. Initially, T looks like the IBLT in Table 2.1 (the 1,2,3,4 in the first row are indices of the bucket).

Table 2.1: Initial IBLT

IBLT T	1	2	3	4
keysum	0000	0000	0000	0000
count	0	0	0	0

We consider inserting three keys, $x = 1011, y = 1001, z = 0111$ (we assume these are the bitstring representations of the keys), into the IBLT. We first insert $x = 1011$, which hashes to buckets 1 and 2 (we have not specified our hash functions, so this choice is arbitrary). Table 2.2 shows what T now looks like.

Table 2.2: IBLT after inserting key x

IBLT T	1	2	3	4
keysum	1011	1011	0000	0000
count	1	1	0	0

We insert $y = 1001$, which hashes to buckets 2 and 3. Table 2.3 shows T after inserting this additional key.

Table 2.3: IBLT after inserting keys x, y

IBLT T	1	2	3	4
keysum	1011	0010	1001	0000
count	1	2	1	0

And we finally insert $z = 0111$, which hashes to buckets 3 and 4, leaving the IBLT as in Table 2.4.

Table 2.4: IBLT after inserting keys x, y, z

IBLT T	1	2	3	4
keysum	1011	0010	1110	0111
count	1	2	2	1

Note that after this last step we cannot find key y in the IBLT, as it is only in buckets with `count` > 1. We begin our `listing` process by searching for a bucket with `count`$= 1$. Buckets 1 and 4 satisfy this property, and are added to our queue. We pop the first bucket off the queue (bucket 1), and remove the key x from each location it hashed to (buckets 1 and 2). This leaves us with the IBLT as shown in Table 2.5.

Table 2.5: IBLT after removing key x

IBLT T	1	2	3	4
keysum	0000	1001	1110	0111
count	0	1	2	1

When removing x from bucket 2, we find that bucket 2's resultant `count` is 1, so we add bucket 2 to the queue. We then pop the next bucket off the queue (bucket 4), and remove the key y from each location it hashed to (buckets 3 and 4). During the process of removing y from bucket 3, we find that bucket 3's `count` is 1, so we add bucket 3 to the queue. This leaves us with the IBLT as in Table 2.6.

Table 2.6: IBLT after removing keys x, z

IBLT T	1	2	3	4
keysum	0000	1001	1001	0000
count	0	1	1	0

We pop the next bucket off the queue (bucket 2), and remove key z from buckets 2 and 3, leaving us with the IBLT as in Table 2.7.

Table 2.7: Final IBLT after `listing`

IBLT T	1	2	3	4
keysum	0000	0000	0000	0000
count	0	0	0	0

We then pop the last bucket off the queue (bucket 3), but its `count` is already 0, so we skip it. There are no more buckets in the queue, so we check to see that all buckets are empty. Indeed, that is the case, so we are done!

Success Probability

Goodrich and Mitzenmacher [21] proved bounds on the success rate of a `listing` operation.

Theorem 1. *If m is the number of buckets in the IBLT, and t is a chosen threshold, then as long as m is chosen so that $m > (c_k + \epsilon)t$ for some $\epsilon > 0$, `listing` fails with probability $O(t^{-k+2})$ whenever n, the number of keys needing to be peeled, is $\leq t$.*

We list some values for c_k in Table 2.8.

Table 2.8: Threshold for successful listing

k	3	4	5	6	7
c_k	1.222	1.295	1.425	1.57	1.721

Supporting Removal of Keys Not Present in IBLT

We will now consider extending IBLTs to support removal of keys not present in the IBLT. To do so, let us first consider what happens when we delete a key k from an empty IBLT. We compute the exclusive-or of x with the existing `keysum` and subtract 1 from the count. So now we have a bucket with a count of -1. Note also that since `insert` and `delete` both use exclusive-ors, the `keysum` is the same whether we inserted or deleted the key. In this situation, even though the key was never inserted before being deleted, `keysum` exactly corresponds to the key itself, and furthermore, we can tell that the key was never inserted since the `count` is -1.

Before, we could only retrieve a key from a bucket if its `count` was 1. Furthermore, having a `count` of 1 was sufficient to guarantee that exactly one key was in the bucket. Now, we can retrieve a key from a bucket if its `count` is 1 or -1. However, having a `count` of 1 or -1 is no longer sufficient to guarantee that there is only one key in the bucket. To see this, consider three distinct keys x_1, x_2, x_3 that all happen to hash to the same bucket i. If we first insert x_1 and x_2, and then delete x_3, then the resultant `count` of bucket i will be 1 even though there are three keys in that bucket.

To ensure that we only retrieve a key when there is exactly one key in the bucket, we introduce a new hash function H that maps keys to values in $\{0, 1, \ldots, 2^l - 1\}$. $H(x)$ for some key x is thus an l-bit hash for the key. We also add a new field to each bucket called the `hashsum` that is the exclusive-or of all the l-bit hash values of the keys added to that bucket. insert and delete now additionally need to compute the exclusive-or of $H(x)$ with the hashsum. We provide updated pseudocode in Method 3.

Method 3 Insert a new key to the IBLT

```
function INSERT(key)
    for i in GetKeyIndices(key) do
        $T$[i].keysum = $T$[i].keysum $\oplus$ key
        $T$[i].hashsum = $T$[i].hashsum $\oplus H$(key)
        $T$[i].count = $T$[i].count + 1
```

For listing, in addition to checking if count is 1 or -1, we further check that H(keysum) = hashsum. For exactly one key to be in the bucket, H(keysum) must equal hashsum. If H(keysum) = hashsum, we still have a small probability of retrieving an incorrect key (i.e., when the keysum in a bucket corresponds to an exclusive-or of multiple keys, but H(keysum) still happens to match the hashsum at the time). Assuming a completely random hash function, we will only retrieve an erroneous key with probability 2^{-l}. Our modified listing protocol that accounts for negative values of count is shown in Method 4.

Set Reconciliation

For set reconciliation, we introduce a new operation called subtract that takes another IBLT (of the same size) as a parameter and "removes" all the keys in that other IBLT from the current IBLT. More specifically, to perform a subtract operation, we go through each of the buckets in turn, compute the exclusive-or of the keysums from both IBLTs, compute the exclusive-or of the hashsums, and subtract the two counts. As shorthand, for IBLTS T_1, T_2, T_3, when we write $T_3 = T_1 - T_2$, we mean that T_3 is the result of subtracting T_2 from T_1. Pseudocode for subtract is provided in Method 5.

Let us consider inserting a set of keys S_A into IBLT T_A and inserting a set of keys S_B into IBLT T_B. We will determine which keys from S_A and S_B we will be able to retrieve during a listing operation from the IBLT $T = T_A - T_B$.

If a key k is in both S_A and S_B, then k's contribution to the keysum and hashsum

Method 4 List all the keys in the IBLT. S_A will contain all the keys that were added to the IBLT, and S_R will contain all the keys that were removed from the IBLT without having been added previously

```
function Listing($S_A, S_R$)
    queue = emptyQueue()
    for i = 0 to $m - 1$ do
        if $T[i]$.count = 1 or -1 and $H(T[i]$.keysum$) = T[i]$.hashsum then
            Add $i$ to queue
    while queue not empty do
        $i$ = queue.pop()
        $c = T[i]$.count
        if $c$ != 1 or -1 then
            continue
        $key = T[i]$.keysum
        if $c$ is 1 then
            add $key$ to $S_A$
        if $c$ is -1 then
            add $key$ to $S_R$
        for $i$ in GetKeyIndices($key$) do
            $T[i]$.keysum = $T[i]$.keysum $\oplus key$
            $T[i]$.hashsum = $T[i]$.hashsum $\oplus H(key)$
            $T[i]$.count = $T[i]$.count - $c$
            if $T[i]$.count = 1 or -1 and $H(T[i]$.keysum$) = T[i]$.hashsum then
                Add $i$ to queue
    for $i = 0$ to $m - 1$ do
        if $T[i]$ is not empty then
            return Failure
    return Success
```

Method 5 Subtract T_2 from T_1. T_3 will contain the resulting IBLT

```
function Subtract($T_1, T_2, T_3$)
    for i = 0 to $m - 1$ do
        $T_3$[i].keysum = $T_1$[i].keysum $\oplus T_2[i]$.keysum
        $T_3$[i].hashsum = $T_1$[i].hashsum $\oplus T_2[i]$.hashsum
        $T_3$[i].count = $T_1$[i].count - $T_2$[i].count
```

in each relevant bucket of T will be exclusive-ored twice (once for k being in T_A, and once for k being in T_B), and k's contribution to `count` will be 0 (+1 for it being in T_A, and -1 for it being in T_B), and thus T will have no idea that k was inserted. Thus, if we perform a `listing` operation on $T = T_A - T_B$, we will not find any key in $S_A \cap S_B$.

Any key in $S_A \setminus S_B$ will be exclusive-ored once in `keysum` and `hashsum` for each corresponding bucket in T (since it was added to T_A but not to T_B), and will contribute +1 to `count`. Any key in $S_B \setminus S_A$ will be exclusive-ored once in `keysum` and `hashsum` for each corresponding bucket in T (since it was added to T_B but not to T_A), and will contribute -1 to `count`. Thus, the keys in the IBLT after `subtract` will correspond exactly to the keys in the set difference, and we can just use a `listing` operation to retrieve them.

For set reconciliation, all we need to do is have Party A create an IBLT T_A containing all the keys from S_A and have him send T_A to Party B. Party B then creates an IBLT T_B containing all the keys from S_B and subtracts T_A from T_B to get a final IBLT T_C. Assuming the size of the set difference is less than the threshold t, Party B will be able to successfully perform a `listing` operation on T_C, and can thus determine all the keys in $S_A \setminus S_B$ and $S_B \setminus S_A$.

Example Set Reconciliation Using IBLTs

We now provide an example of two-party set reconciliation using IBLTs. As in our `listing` example above, we once again have keys $x = 1011, y = 1001, z = 0111$ that hash to buckets (1,2),(2,3), and (3,4) respectively. We have an additional key $a = 0001$ that hashes to buckets 2 and 4. Party A has a set of keys $S_A = \{x, y, z\}$, and Party B has a set of keys $S_B = \{y, z, a\}$, so $S_A \setminus S_B = \{x\}$ and $S_B \setminus S_A = \{a\}$. After inserting S_A and S_B into their respective IBLTs T_A and T_B, T_A and T_B look as in Table 2.9 (we omit the `hashsum` field for simplicity).

Note that a `listing` operation on T_B would fail since there is no `count` equal to 1. We then compute $T = T_A - T_B$, which gives us the resultant IBLT T as shown in Table 2.10.

Table 2.9: IBLTs T_A and T_B after inserting keys $\{x, y, z\}$ and $\{y, z, a\}$ respectively

IBLT T_A	1	2	3	4
keysum	1011	0010	1110	0111
count	1	2	2	1

IBLT T_B	1	2	3	4
keysum	0000	1000	1110	0110
count	0	2	2	2

Table 2.10: $T = T_A - T_B$

IBLT T	1	2	3	4
keysum	1011	1010	0000	0001
count	1	0	0	-1

Note that bucket 2's `count` is 0 but its `keysum` is non-zero. This is the case since it has one key from S_A and one key from S_B. Now, we search through each bucket for `count`s that are 1 or -1. In this case, we add buckets 1 and 4 to the queue. We pop the first bucket from the queue (bucket 1), removing the key x from both locations it hashed to (buckets 1 and 2). Since bucket 1's `count` was 1, we know x came from Party A's set. T with x removed is shown in Table 2.11.

Table 2.11: T after removing key x

IBLT T	1	2	3	4
keysum	0000	0001	0000	0001
count	0	-1	0	-1

Note that during the process of removing x, bucket 2's `count` became -1, at which time we added bucket 2 to the queue. We now pop the next bucket from the queue (bucket 4), removing the key z from both locations it hashed to (buckets 2 and 4). Since bucket 2's `count` was -1, we know z came from Party B's set. This also means that we add 1 to `count` of each location z hashed to rather than subtracting 1. We then pop the last bucket from the queue (bucket 2), but since its `count` is already 0, we skip it. We are then left with an empty IBLT, and our `listing` process finishes. We indeed obtained that $S_A \setminus S_B = \{x\}$ and $S_B \setminus S_A = \{a\}$.

2.2.4 Multi-party Invertible Bloom Lookup Tables

We now consider the extended set reconciliation problem, which involves multiple parties that all want to reconcile their sets. As stated in the introduction, we want to determine elements from each set that are not in the intersection of all the sets. That is, we have sets $S_1, S_2, \ldots, S_m$, and letting I denote $\cap_{i \in 1 \ldots m} S_i$, we want to determine

$$(S_1 \setminus I), (S_2 \setminus I), \ldots, (S_m \setminus I)$$

An extension of IBLTs was recently proposed [31] to handle this situation. In the next section, we explain the intuition behind that extension, and show how it works in practice.

Changing the Key Representation for Multiple Parties

In the 2-party case, we considered keys as bitstrings, or equivalently, numbers base 2. We can think of the exclusive-or we performed for `keysum` before as taking the bit-wise sum of two numbers base 2. For instance, if we have $x = 011_2$ and $y = 101_2$, then $x \oplus y = 110_2$ (counting from the right leftwards, bit 0 in $x \oplus y$ results from adding 1 and 1 base 2, which is 0, bit 1 results from adding 0 and 1 base 2, which is 1, and bit 2 results from adding 0 and 1 base 2, which is also 1). A bitstring key exclusive-ored with itself always equals 0. We can see this since each bit is either 0 or 1. If it is 0, then when computing the exclusive-or, we find $0 + 0 \mod 2 = 0$, and if the bit is 1, we compute $1 + 1 \mod 2 = 0$, meaning that the result is always 0.

We want to find something analogous for $n > 2$ parties. More specifically, we want a key representation and "exclusive-or" such that if we add the same key to the same bucket n times, then it cancels out. Based on the discussion in the previous paragraph, we can imagine that instead of representing each key as a number base 2 (for two parties), we can represent each key as a number base n (for n parties), which we call the key's *nit*-wise representation (so for three parties, it would be its $3it$-wise representation). Our "exclusive-or" operation then takes the *nit*-wise sum of the two keys, computing the sum of each *nit* individually, and then reducing each *nit* modulo n.

As an example, let us consider a key x with a binary representation of 10001_2. x's *nit*-wise representation would then be 10001_n. We denote x's binary representations as x_2, and more generally denote the *nit*-wise representation of a key x as x_n. We overload $\oplus$ for *nit*-wise addition, and $\ominus$ for *nit*-wise subtraction. As an example of these operations, let us consider $n = 3$ (corresponding to 3 parties) and the same key x defined previously. Then $x_3 \oplus x_3 = 20002_3$, and $x_3 \oplus x_3 \oplus x_3 = 00000_3$. Thus, exactly as desired, if the same key is added n times, then it cancels out of the *nit*-wise representation. Note that if a key is added i times for $i < n$, then the result's *nit*-wise representation will consist only of i's and 0's. As another example, if we have a different key y such that $y_2 = 10111$, then $x_3 \oplus y_3 = 20112_3$, and $x_3 \ominus y_3 = 00220_3$.

Changing the Protocol for Multiple Parties

With this modification to the key representation, we can consider how to change the protocol to handle multiple parties. Each bucket of the multi-party IBLT will once again have `keysum`, `hashsum`, and `count` fields, except the former two now store *nit*-wise representations of the numbers. `count` is now a counter modulo n – we explain why this is the case in a bit.

`insert` and `delete` proceed analogously to the two-party case, except using *nit*-wise $\oplus$ and $\ominus$ operations.

For `listing`, we need to be a bit more careful. As shown previously, if a key is added n times to an n-party IBLT, it will cancel out of all the buckets it was added to. Let us consider what happens if there is a single key x that is in a bucket, but that key is added i times for $0 < i < n$. As we explained before, we expect that the `keysum` and `hashsum` will have *nit*-wise representations consisting solely of i's and 0's. Similarly, since the key is added i times, we expect `count` to be i. Thus, to check if a key x is in a bucket b, we first look at bucket b's `count`. Say that it has a value j. We then see if bucket b's `keysum` and `hashsum` *nit*-wise representations consist solely of j's and 0's. If so, then let the key (that when added j times corresponds to `keysum`'s *nit*-wise representation) be y, and let the hash (that when added j times corresponds to `hashsum`'s *nit*-wise representation) be z. As an example, if we see that `keysum` is 4044_5 and `count` is 4, then the key y is 1011. However, if we see that `keysum` is 4034_5 and `count` is 4, we know that there is more than one distinct key in the bucket. We then check that $H(y) = z$ to confirm that the retrieved key does indeed correspond to exactly one key. If it does, then we can remove j copies of key y from the IBLT, and continue the `listing` process as before.

To use this for set reconciliation, say we have n parties with IBLTs $T_1, T_2, \ldots, T_n$ respectively. Each party adds its corresponding keys to its IBLT. We define an `add` operation for IBLTs that is analogous to `subtract` defined for two-party IBLTs, except that we add the corresponding `count`s rather than subtracting them. We also adopt the shorthand of $T = T_1 + T_2$ meaning that IBLT T is the result of performing `add` on T_1 and T_2. Now consider the IBLT $T = T_1 + T_2 + \ldots + T_n$. Any key that appears in all n sets will be canceled out of `keysum`, `hashsum`, and `count`, so the keys in T are exactly the keys from $(\cup_i S_i) \setminus (\cap_i S_i)$. We then perform a `listing` operation to retrieve all those keys.

Determining Which Key Belongs to Which Party

As described so far, we are only able to determine the set of keys in $(\cup_i S_i) \setminus (\cap_i S_i)$, but not which party had each key to begin with. To resolve this problem, we can add a bit vector `bv` of length n to each bucket. When Party i adds a key to bucket j in its IBLT, it toggles the ith bit of `bv` in bucket j. When summing IBLTs, we now additionally sum the corresponding `bv`s. When we find a bucket with exactly one key, we check `bv` to see which bits are set. All the bits that are set correspond to the parties that originally had the key. We then remove this `bv` from the `bv` of all the buckets the key hashed to.

As stated, this method of determining which parties had a certain key is not entirely correct, and even after further modification, actually only works when the number of parties is odd. Say that Parties A, B, C all have sets with one key x. Then in the resultant IBLT $I = I_A + I_B + I_C$, for each bucket that x hashes to, the `keysum`, `hashsum`, and `count` will all be 0, but `bv` will be 111. Now say that Party A has an additional key y that hashes to that same bucket. In $I = I_A + I_B + I_C$, the `keysum` of that bucket will be y, the `hashsum` will just be $H(y)$, and the `count` will be 1, but `bv` will be 011. Using our approach as stated so far, we would say that Parties B

and C both had key y even though only Party A had it. To correct this, we need to check that the `count` matches the number of bits set in `bv`. If it does not, then we flip all the bits in `bv` before checking that `count` matches the number of bits set in `bv`. This approach does not necessarily work if n is even, since if $n/2$ bits of `bv` are set, flipping all the bits will not change the number of set bits. In that situation, we will be unable to determine if the parties corresponding to the original `bv` or the final `bv` are the ones that originally had the key.

Example Set Reconciliation Using Multi-party IBLTs

We extend our two-party set reconciliation example to three-party set reconciliation by introducing an additional Party C. As before, we have keys $x = 1011, y = 1001, z = 0111, a = 0001$ that hash to buckets (1,2),(2,3),(3,4),(2,4) respectively. Parties A and B once again have sets of keys $\{x, y, z\}$ and $\{y, z, a\}$ respectively. Party C has keys $\{z, a, x\}$. Thus, all three parties share key z, but keys x, y, a are each in two out of the three sets. We once again omit the `hashsum` field, but now include the `bv` field. We index `bv` such that position 0 (leftmost bit) corresponds to Party A, position 1 to Party B, and position 2 to Party C. We use the subscript $_3$ in `keysum` to note that these are $3it$-wise representations. The initial IBLTs T_A, T_B, T_C are shown in Table 2.12.

Table 2.12: IBLTs T_A, T_B, T_C after inserting keys $\{x, y, z\}, \{y, z, a\}, \{z, a, x\}$

IBLT T_A	1	2	3	4
keysum	1011_3	2012_3	1112_3	0111_3
count	1	2	2	1
bv	100	000	000	100

IBLT T_B	1	2	3	4
keysum	0000_3	1002_3	1112_3	0112_3
count	0	2	2	2
bv	000	000	000	000

IBLT T_C	1	2	3	4
keysum	1011_3	1012_3	0111_3	0112_3
count	1	2	1	2
bv	001	000	001	000

We then compute $T = T_A + T_B + T_C$. The resultant T is shown in Table 2.13.

Table 2.13: $T = T_A + T_B + T_C$

IBLT T	1	2	3	4
keysum	2022_3	1020_3	2002_3	0002_3
count	2	0	2	2
bv	101	000	001	100

We now go through T, looking for any bucket with `count` $i > 0$ and `keysum` consisting solely of i's and 0's. Buckets 1, 3, and 4 satisfy this property, so we add them in that order to our queue. We see that bucket 1's `keysum` is 2022_3 and `count` is 2, meaning that our key x is 1011, and that x appears in two of the original sets. Using `bv` (which agrees that there are two parties holding the key, since the number of set bits is 2), we can tell that Parties A and C originally had x. We now remove two copies of x from T. The IBLT with two copies of x removed from T is shown in Table 2.14.

Table 2.14: T with two copies of key x removed

IBLT T	1	2	3	4
keysum	0000_3	2001_3	2002_3	0002_3
count	0	1	2	2
bv	000	101	001	100

We now pop the next bucket off the queue, which is bucket 3. We see that bucket 3's `keysum` is 2002_3 and `count` is 2, meaning that our key y is 1001, and that y appears in two of the original sets. However, if we look at bucket 3's `bv`, we see that the number of set bits is actually 1. Thus (as explained in the previous section), we flip all the bits in `bv`, and can tell that Parties A and B originally had key y. We now remove two copies of y from T. The new IBLT is shown in Table 2.15.

Table 2.15: T with two copies of key x and two copies of key y removed

IBLT T	1	2	3	4
keysum	0000_3	0002_3	0000_3	0002_3
count	0	2	0	2
bv	000	100	000	100

We pop the next bucket off the queue, which is bucket 4. We see that bucket 4's `keysum` is 0002_3 and `count` is 2, meaning that our key a is 0001, and that a appears in two of the original sets. However, if we look at bucket 4's `bv`, we see that the number of set bits is actually 1, so we flip all the bits in `bv`, and can tell that Parties B and C originally had key a. We now remove two copies of a from T, leaving us with an empty IBLT, and our `listing` finishes. Again, we are able to retrieve all the keys for each party that are not already contained by all parties.

2.3 Estimating the Size of the Set Difference

For both of our near-optimal methods (recall that we defined "near-optimal" as having transmission complexity within a constant factor of the information-theoretic limit), we needed a good estimate of the size of the set difference, which we denote d. In the case of the characteristic polynomial method, the degree of the rational function formed from the quotient of both parties' characteristic polynomials depended on d. In the case of IBLTs, the choice of the size of the table depended on d.

Several methods have been proposed to estimate the size of the set difference, including using a hierarchy of samples [14], using random projections [22], and using a more generalized sketching algorithm involving limited-independence random variables [19].

In the following sections we describe three approaches, the Strata Estimator, Min-wise sketches, and repeated doubling. We adopt the Strata Estimator in our file synchronization protocol.

2.3.1 Strata Estimator

Eppstein, Goodrich, Uyeda, and Varghese [18] proposed a data structure using multiple smaller IBLTs to estimate the set difference. Their approach works especially well when the set difference is small. If the set difference is large, then other approaches, including random sampling [22] and min-wise hashing [10, 11] (which we describe in the Section 2.3.2), work better.

Their approach is based on the methods developed by Flajolet and Martin [20] to estimate set sizes. We assume that we have a set S such that all elements of S come from a universe of size u. We also assume that we have a random hash function h that maps the elements of the universe to the integers in $[0, u)$. In Flajolet and Martin's approach, we create a $\log(u)$ bit estimator, with each bit i in the estimator set to 1 if there is at least one element $e \in S$ such that the ith bit of $h(e)$ is the first one that is set, counting from the right. Since we assume that the hash function is random, for a given element, the probability that the ith bit is the first bit that is set is $1/2^{i+1}$. The estimator then returns 2^I as the set size, where I is the highest index such that bit I is set in the estimator.

Adapting this for set differences, we separate the universe U into $L + 1 = \log(u)$ partitions, $P_0, P_1, \ldots, P_L$ such that the ith partition covers a fraction $1/2^{i+1}$ of U. This can be done quite easily by mapping each element e in the universe to partition i, where i is the highest power of 2 that divides e (equivalently, the number of trailing zeros in e's binary representation). We correspond to each partition an IBLT of some fixed size C (Eppstein et al. choose $C = 80$). We call the series of $L + 1$ IBLTs a "Strata Estimator". For a given Strata Estimator E, $E[i]$ corresponds to the ith IBLT. To create a Strata Estimator E for a set S, for each element $e \in S$, we compute $h(e)$, count the number of trailing zeros of $h(e)$, which we denote z, and insert $h(e)$ into $E[z]$.

To estimate the size of the set difference, Parties A and B create Strata Estimators E_A and E_B for their respective sets S_A and S_B. Party A then transmits E_A to Party B. For $i \in \{L, L-1, \ldots, 0\}$ in descending order, Party B computes $E = E_B[i] - E_A[i]$ and then performs a `listing` operation on E. Each time `listing` succeeds, we add the number of recovered keys to a counter. If `listing` is unsuccessful, then we estimate the size of the set difference as $2^{i+1}c$, where c is the value of the counter (which corresponds to the number of recovered keys), and i is the index at which our `listing` failed.

As shown in [18], the Strata Estimator technique performs quite well:

Theorem 2. *Let ϵ and δ be constants in the interval (0,1), and let S and T be two sets* *whose set difference has size d. If we encode the two sets with our Strata Estimator, in which each IBLT in the estimator has C cells using k hash functions, where C and k are constants depending only on ϵ and δ, then with probability at least $1 - \epsilon$, it is possible to estimate the size of the the set difference within a factor of $1 \pm \delta$ of d.*

This approach takes only one round and is able to estimate the size of the set difference using $O(\log(u))$ space. However, the constant factors hidden are quite large, so it works best when the sets of keys compared are quite large. Eppstein et al. also proposed a hybrid approach using both IBLTs and min-wise sketches to improve performance for a fixed size estimator.

2.3.2 Min-wise Sketches

Min-wise sketches [10, 11] are another approach to estimate the set similarity. In particular, they are used to estimate the resemblance r of two sets S_A and S_B, defined as $r = \dfrac{S_A \cap S_B}{S_A \cup S_B}$. This quantity is also known as the Jaccard Similarity coefficient.

To create a min-wise sketch, we first select k random hash functions, $h_1, h_2, \ldots, h_k$ that each permute elements within the universe U of keys. As an example, if we imagine our universe of keys to be $\{0, 1, 2, 3\}$, then one such hash function h_A satisfying the permutation property would be defined as follows: $h_A(0) = 2, h_A(1) = 1, h_A(2) = 3, h_A(3) = 0$.

We then apply each of the hash functions $h_1, h_2, \ldots, h_k$ to all of the elements of S. Let $\min(h_i(S))$ be the smallest value produced by h_i over all the elements of our chosen set S. Using the hash function h_A we defined before, if our set S were $\{0, 2\}$, then $\min(h_A(S))$ would be 2, since $h_A(0) = 2$ and $h_A(2) = 3$. The min-wise sketch is defined as the k values $\min(h_1(S)), \min(h_2(S)), \ldots, \min(h_k(S))$. Letting the min-wise sketches for Parties A and B be W_A and W_B respectively, the set similarity is estimated by the fraction of hashes from W_A and W_B that return the same minimum value.

Expanding on our example, say that we have two more hash functions, h_B and

h_C defined as follows:

$$h_B(0) = 0, h_B(1) = 1, h_B(2) = 3, h_B(3) = 2$$

$$h_C(0) = 3, h_C(1) = 2, h_C(2) = 1, h_C(3) = 0$$

Let us say that our two sets are $S_1 = \{1, 3\}$ and $S_2 = \{2, 3\}$. Then the min-wise sketch using the three hash functions h_A, h_B, h_C for S_1 would be $\{0, 1, 0\}$ and the min-wise sketch for S_2 would be $\{0, 2, 0\}$. Our estimate of the set similarity would then be $2/3$, since both min-hashes corresponding to h_A are 0, and both min-hashes corresponding to h_C are 0.

If S_A and S_B have resemblance r, then we expect that the number of matching hashes m in W_A and W_B to be $m = rk$. This is the case since the hash functions are random permutations, so each element in $S_A \cup S_B$ is equally likely to be the one that hashes to the minimum. For $\min(h_i(S_A))$ to equal $\min(h_i(S_B))$, we need the element mapping to that minimum to be in both sets (in $S_A \cap S_B$). This happens with probability $(S_A \cap S_B)/(S_A \cup S_B)$, which is precisely the resemblance.

Using the fact that $|S_A \cap S_B| = |S_A| + |S_B| - |S_A \cup S_B|$ and $|S_A \cup S_B| = |S_A| + |S_B| - |S_A \cap S_B|$, we can rewrite the resemblance equation $r = \dfrac{S_A \cap S_B}{S_A \cup S_B}$ as

$$r = \frac{|S_A| + |S_B| - |S_A \cup S_B|}{|S_A| + |S_B| - |S_A \cap S_B|}$$

Rearranging, we find that

$$|S_A \cup S_B| - |S_A \cap S_B| = \frac{1-r}{1+r}(|S_A| + |S_B|)$$

Note that the left hand size is exactly the size of the set difference d.

2.3.3 Repeated Doubling and Using Number of Listed Keys

The Strata Estimator and Min-wise Sketches methods both seek to first determine the size of the set difference and then use that information to determine the appropriate data structure to transmit. Let us consider an alternative approach, where we just guess an initial size for the set difference, transmit the data structure according to that guess, and retry until we succeed. In the case of Invertible Bloom Lookup Tables, we start with Party A transmitting an IBLT of some fixed size c to Party B. Party B then subtracts its IBLT from Party A's and performs a `listing` operation. If `listing` succeeds, then we are done. Otherwise, we double the size of the IBLT and retry. As this is an exponential backoff solution, we will transmit messages with a total size proportional to the set difference. However, this method potentially requires many rounds of communication, which is a significant downside.

In the following section, we see if we can gain any more information from the number of keys listed during a failed `listing` operation. More concretely, say that we have an IBLT with 80 buckets, and when we tried to perform a `listing` operation, we managed to list only 2 keys before every bucket's `count` was greater than 1. Does this mean we have approximately 100 keys in the table? 1000? 10000? If we can determine a good estimate, then we can choose an appropriate expansion factor that is potentially much greater than 2, which will reduce the number of rounds of communication needed to successfully list. Unfortunately, as we will show, if the number of keys inserted to the IBLT is significantly more ($>$4 times) the number of IBLT buckets, then for IBLTs of reasonable size, we will most likely not be able to list any keys.

Listing Analysis

Say we have an IBLT with m buckets, we are using k independent and random hash functions, and we insert n keys to the IBLT. Determining the number of keys that hash to each bucket is effectively the same as the well-studied balls-and-bins problem, where we have a fixed number of bins, and we randomly add balls to those bins. Our buckets are the bins, and our keys are the balls. In our situation, we actually have kn balls since each individual key is inserted into k slots.

As in typical balls-and-bins analyses, we use the Poisson approximation to the Binomial [32]. That is, we assume that the number of balls that fall into a certain bin, X, follows a Poisson Distribution:

$$P(X = x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

with $\lambda = \dfrac{kn}{m}$.

A key can be retrieved from an IBLT bucket if it is the only key in that bucket, and thus we want to find the number of buckets with exactly 1 key. For a Poisson random variable,

$$P(X = 1) = \frac{\lambda e^{-\lambda}}{1!}$$

If we let Y_i be an indicator random variable for bucket i having exactly one key, we have that the expected value of Y_i is just equal to the probability that bucket i has exactly one key. Plugging in to the previous expression, we get the following:

$$E[Y_i] = \frac{kn}{m} e^{-kn/m}$$

If we let $Y = Y_1 + Y_2 + \ldots + Y_m$ be a random variable for the total number of buckets having exactly one key, by linearity of expectation, we have

$$E[Y] = E\left[\sum_i Y_i\right] = \sum_i E[Y_i] = kne^{-kn/m}$$

So for a fixed number of buckets m, the number of retrievable keys falls of exponentially with the number of inserted keys. Note that this is only for the first sweep through the IBLT for `listing`. With more nuanced "peeling" analysis (see [23] for more information), which takes into consideration the fact that when we remove the keys found during the first sweep through the IBLT, more buckets will now have only one key in them, we would be able to show a more precise result. For our purposes, this simpler analysis is sufficient.

Assuming $k = 4$, an IBLT of 1000 buckets, and 4000 inserted keys, we find that the expected number of buckets with exactly one key is ≈ 0.002, which is quite small.

If there is not a single bucket with exactly 1 key, that we cannot even begin the `listing` process, in which case `listing` will always fail with 0 keys retrieved. This means that the number of keys successfully listed only is useful when the number of keys inserted is within a rather small (<4) multiplicative constant above the IBLT table size, as above that number we will almost always retrieve 0 keys.

Chapter 3

File Synchronization

As stated in the introduction, the file synchronization problem involves efficiently keeping versions of a file from multiple different hosts up to date. This problem arises in many situations, including website mirroring, file system backup and replication. In those sorts of situations, successive versions of each file generally are very similar. We would like to take advantage of that fact to decrease the amount of data transmitted, which is especially relevant in systems where bandwidth is a bottleneck. Ideally, the amount of data transmitted should be roughly proportional to the "difference" between the old and new file.

In general, file synchronization techniques fall into one of two categories, single-round protocols and multi-round protocols. Single-round protocols, as their name suggests, involve one round of communication between local and remote host. Multi-round protocols generally involve a divide and conquer approach, where one party breaks the file down into a few large blocks, sends hashes corresponding to those blocks, and then recursively divides any blocks the remote host was not able to match until a matching block is found.

Single-round protocols are more effective when transmitting small files or when the network has high latencies, as the overhead of sending multiple rounds of messages might not be worth the bandwidth savings. Multi-round protocols can be more effective when transmitting large files or when the network is bandwidth limited – as experiments have shown, multi-round protocols can allow significant transmission savings over single-round protocols on real data sets [38].

In Section 3.1, we describe rsync, a popular single-round protocol. Since our file synchronization protocol involves a (small) constant number of rounds of messages, we do not describe multi-round protocols in more detail.

In Section 3.2, we describe a few techniques that use set reconciliation for file synchronization. We focus on one such method, called string reconciliation using puzzles, since we use an optimization based on that technique in our file synchronization protocol discussed in Chapter 4.

In Section 3.3, we discuss content-dependent file partitioning, which uses local features of a file to partition that file into blocks. This is beneficial since two parties can independently partition their files (using the same content-dependent file partitioning technique), and if the two files are similar, end up with relatively similar blocks. We adopt this approach to partitioning the file in our file synchronization protocol.

We have some assumptions in our setup for file synchronization. First, we assume that files can be modified in arbitrary ways (insertion, deletion, etc). Second, we assume that there is no system that can log all file changes and then transmit them to the remote machine. Third, we assume that the synchronization is between two parties. Methods have been proposed to handle multiple parties [34], though we do not discuss them here.

3.1 rsync

rsync [39] is a popular single-round protocol for file synchronization. The basic idea behind rsync is to split a file into blocks and use hash functions to compute short hashes of those blocks. The hashes from the local file are then compared with the hashes computed for the remote file. If a hash does not match, then we need to send the actual block contents that correspond to the unmatched hash. Otherwise, we can just send an acknowledgement that we already have the corresponding block.

In particular, rsync takes advantage of rolling hash functions [24] to skip file blocks that match on both the remote and local machines. However, it is unable to use more nuanced information from the outdated file to reduce transmission overhead. For instance, `rsync` is unable to detect that the file block is present on the local machine but in a different file). We describe the rsync protocol in more detail below.

3.1.1 Protocol

Say Party A wants to get an updated version of a file from Party B.

1. Parties A and B agree on some block size b. Let f_A and f_B denote A's and B's file respectively, and let $f_A(i, i+b-1)$ denote the b-gram (the b bytes) starting at position i in f_A.

2. Party A splits its file into disjoint blocks B_i, each of size b (starting at file positions $0, b, 2b, \ldots$), computes a hash $h(B_i)$ for each i, and transmits $h(B_i)$ for each i to Party B.

3. Party B receives those hashes and inserts $(h(B_i), i)$ into a map structure (an efficient key-value store).

4. For each index i in Party B's file, Party B computes $h(f_B(i, i+b-1))$ and checks the map for a matching hash. If a match is found, then Party B sends an acknowledgement of that match back to Party A and increments i by b. If a match is not found, then Party B sends $f_B(i, i)$ (one character) to Party A and then increments i by 1.

5. Party A recreates the file by adding the block corresponding to index i if a match was found or adding the character sent over if a match was not found.

6. To ensure that the recreated file is correct, Party B sends a longer hash corresponding to the whole file f_B (for instance a 128-bit MD5 hash) and transmits it to Party A. Party A computes the same hash on the newly recreated file and ensures that the hashes match. If they do not, then Party A requests Party B to send f_B over in its entirety.

Note that it is necessary for Party B to compute the hash at every offset of its file – if Party B just computed hashes for disjoint blocks of length b, then adding a single character at the beginning of the file would cause all blocks to be misaligned.

Optimizations

The actual `rsync` protocol includes various optimizations for efficiency. To decrease the number of bits sent over the wire, all indices and characters sent are compressed. The hashing in step 4 is done using a rolling hash function, such as the one used for Karp-Rabin String Matching [24], which is able to efficiently compute a new hash based on the last hash computed.

Design Decisions

One significant decision that needs to be made for `rsync` (and many other file synchronization protocols, including ours), is an appropriate block size b. This depends both on the number of changes between the local and remote files and the length of each of those changes. We can understand this more fully by considering a few examples.

Imagine that the changes are all spread out, such that one character is altered in each block. Since `rsync` will compute a different hash for the block with and without the altered character, it will most likely be unable to match a single block, and will have to transfer the full file over with additional overhead. If instead all the changes are clustered in one area (which is more likely in practical scenarios), then most blocks will be unchanged, so only a small section of the file will need to be transmitted. `rsync` uses a default block size of 700 bytes, but for large files, uses the square root of the file size instead (this is just a heuristic, but it seems to work well in practice).

Another parameter that can be tweaked is the number of bits x for the block hashes. `rsync` defaults to 48 bits each. In the following section, we will provide a brief analysis for the approximate number of bits necessary to achieve a certain false positive rate.

Approximate Number of Bits Necessary for Block Hash

Say we have two files, f_A and f_B, each of length n, and we use the rsync protocol as described above. Each hash in f_A ($\frac{n}{b}$ in total) is potentially compared to $n-b+1$ hashes from f_B (corresponding to the $n-b+1$ different blocks of size b created at every offset of the file). Assuming a perfectly random hash function, for hashes of length x, the probability of a specific pair of hashes conflicting is $1/2^x$. For rsync, we are comparing $\frac{n}{b}(n-b+1)$ pairs of hashes in total ($\frac{n}{b}$ from f_A and $(n-b+1)$ from f_B). Using the Poisson Approximation to the Binomial [32], we have that the distribution of the number of conflicting hashes follows

$$X = Pois\left(\lambda = \frac{n}{b}(n-b+1)\cdot\frac{1}{2^x}\right)$$

So the probability that we have no conflicting hashes is:

$$P(X=0) = \exp\left(-\frac{n}{b}(n-b+1)\cdot\frac{1}{2^x}\right)$$

Setting this equal to 2^{-d}, our chosen false positive rate, and solving for x, we find that we need $\approx \lg(n/b) + (\lg n) + d$ bits to have a false positive rate of 2^{-d}.

3.2 Set Reconciliation for File Synchronization

A few methods have been proposed that use set reconciliation as part of a file synchronization protocol, including one by Agarwal, Chauhan, and Trachtenberg [7] that splits files into overlapping blocks (covered in more depth in Section 3.2.1) that are then reconciled, and one by Yan, Irmak, and Suel [40] that uses Characteristic Polynomial Interpolation (see Section 2.1.2) and content-dependent file partitioning (which we will go over in Section 3.3). However, both of the above methods assume prior knowledge of an upper bound on the size of the set difference. We use the idea of splitting the file into overlapping blocks in our file synchronization protocol discussed in Chapter 4.

3.2.1 String Reconciliation Using Multisets and de Bruijn Digraphs

Agarwal et al. [7] proposed a bandwidth efficient method to reconcile strings (equivalently, files) using set reconciliation techniques. Their method scales linearly with the edit distance[1] between the two strings. One limitation of using the edit distance as a metric can be seen in the following example. If the two halves of a file are swapped, then the resultant file will be significantly different from the original file according to the edit distance metric (i.e., it will be proportional to the length of the file itself), even though the file contents are largely the same.

At a high level, their protocol works as follows. Each party converts its string into a multiset[2] of overlapping parts (which they call "puzzle pieces") by computing the length l substring at every offset of the string, where l is some predetermined fixed length. For instance, if the string were "01110011" and l were 3, then the multiset would be $\{011, 111, 110, 100, 001, 011\}$.

After converting their strings into multisets, and further converting those multisets into regular sets (we will go over how they do this later on), the two parties then reconcile their sets using any set reconciliation algorithm. Each party can then determine which pieces of the string it is missing and ask the other party for those pieces. They can then reconstruct the string by choosing the right index in an enumeration of all Eulerian cycles [3] in a modified de Bruijn digraph corresponding to the multiset of puzzle pieces. We will now parse the jargon-laden last sentence.

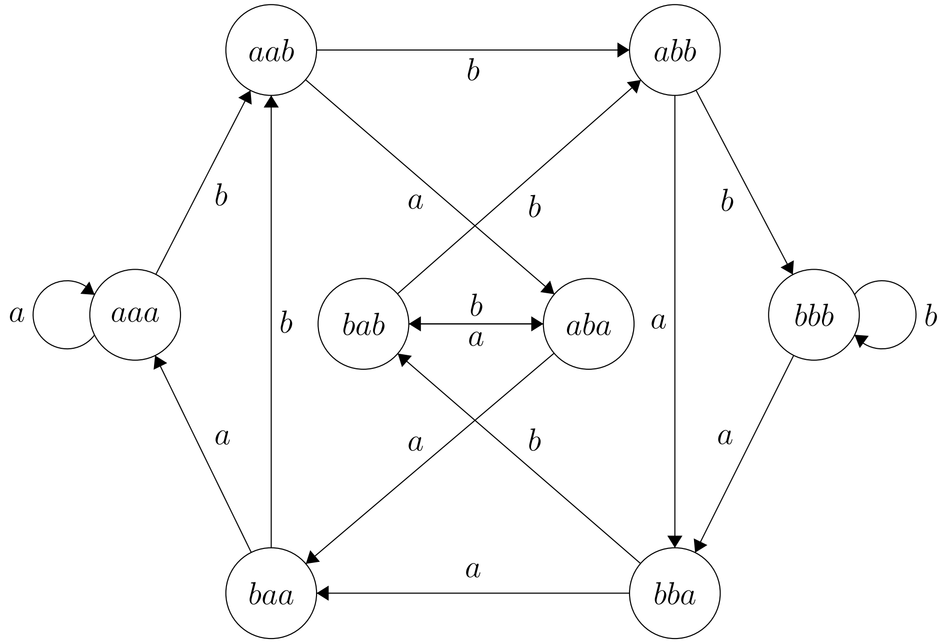
Before we explain what a *modified de Bruijn digraph* is, we will go over a regular *de Bruijn digraph.* The *de Bruijn digraph* $G_l(\Sigma)$ over an alphabet Σ and a chosen length l is defined to contain $|\Sigma|^l$ vertices. Each vertex v's label $L(v)$ corresponds to a distinct string of length l over the alphabet. An edge from vertex v_i to v_j exists with label L_{ij} if the last $l-1$ characters of $L(v_i)$ match up with the first $l-1$ characters of

[1] the edit distance between two strings is the minimum number of inserts, deletes, and replacements of single characters needed to convert one string into the other

[2] a multiset is a set that allows for the same element to appear multiple times

[3] An Eulerian cycle is a path in a graph that traverses every edge exactly once and starts and ends at the same edge

Figure 3-1: de Bruijn digraph for an alphabet $\Sigma = \{a, b\}$ and length $l = 3$

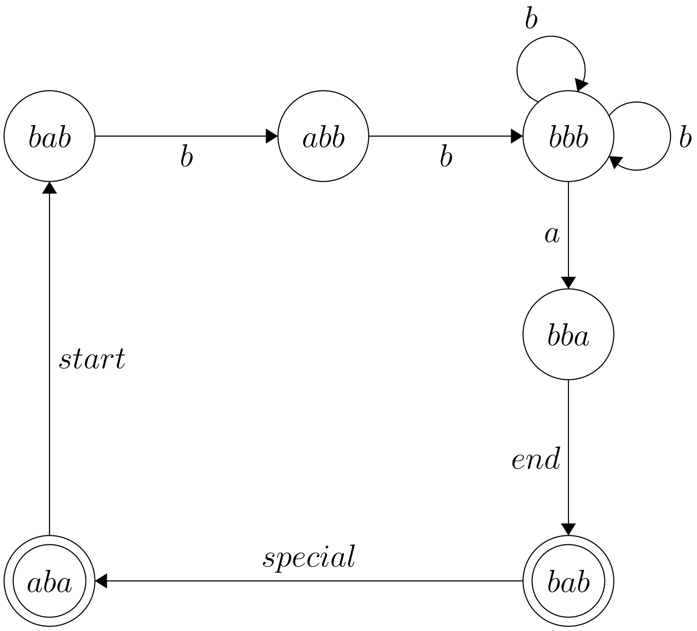


$L(v_j)$, and the last character of $L(v_j)$ is L_{ij}. Effectively, the de Bruijn digraph shows how we can move from any length l string to any other by shifting the characters over by 1 (in the process truncating the leftmost one) and then appending a new character to the end. Figure 3-1 shows the de Bruijn digraph for an alphabet $\Sigma = \{a, b\}$ and length $l = 3$.

A *modified de Bruijn digraph* is a slightly modified version of the de Bruijn digraph to handle the multisets of puzzle pieces created from our full string. First, we add parallel edges between the appropriate de Bruijn digraph vertices for each occurrence of such a transition within our multiset. Second, we delete edges that represent transitions not in our multiset, and delete vertices with degree 0. Third, we add two special vertices and edges corresponding to the first and last pieces of the full string. We denote the special start and end vertices as v_s and v_t, and the special start and end edges as e_s and e_t respectively. Fourth, we add an artificial edge from v_t to v_s, so that the two new vertices have in-degree equal to their out-degree. We denote this edge as e^*.

As an example, the modified de Bruijn digraph for a string *ababbbbbab* (correspond-

Figure 3-2: Modified de Bruijn digraph for string *ababbbbbab*



ing to multiset $\{aba, bab, abb, bbb, bbb, bbb, bba, bab\}$ using the same alphabet $\Sigma = \{a, b\}$ and $l = 3$ is shown in Figure 3-2. Note that node *bbb* has two self-edges, as the substring *bbbbb* corresponds to the two transitions $bbb \to bbb, bbb \to bbb$. Note also that we have two vertices with label *bab*. This is the case since one of those vertices corresponds to v_t, our special end vertex. Note also the special edge connecting v_t to v_s.

Assuming a model of strings as random sequences of bits that each independently have probability p of being 1, Agarwal et al. [7] show that if l is chosen to be $O(\log n)$, where n is the length of the string, then the probability that the maximum out-degree in the modified de Bruijn digraph is one (meaning that there is exactly one Eulerian path), is $\leq 1/2$. If the maximum out-degree is 1, then the full string is uniquely determined by the modified de Bruijn digraph, as we can just start from the special vertex corresponding to the first piece of the full string, traverse every edge once (there is only choice at each step), and return back along e^* from v_t to v_s.

In more detail, the protocol works as follows.

Protocol

1. Parties A and B convert their strings s_A and s_B into multisets M_A, M_B using a chosen length l, and each constructs a modified de Bruijn digraph from the puzzle pieces.

2. Parties A and B determine the indices i_A, i_B of their string encodings in the enumeration of all Eulerian cycles in their modified de Bruijn digraphs. Note that both parties need to predetermine some method for ordering all the Eulerian cycles.

3. Parties A and B convert their multisets into ordinary sets by concatenating each element in the multiset with the number of times it appears, and hashes each of the results. Sets S_A and S_B hold the resulting hashes.

4. Parties A and B reconcile S_A and S_B using characteristic polynomial interpolation (described in Section 2.1.2). At this point, both parties know S_A and S_B. Each then sends the other the index of their string encoding i_A, i_B (computed in step 2) and the puzzle pieces that correspond to the hashes unique to their sets ($S_A \setminus S_B$ and $S_B \setminus S_A$).

5. Parties A and B can now recreate both multisets since they have all the relevant puzzle pieces. They then use the indices i_A, i_B to recreate the strings s_A and s_B.

The idea of using overlapping puzzle pieces to decrease the number of possible Eulerian paths is instructive – we use a similar technique when deciding how to partition our files in our file synchronization protocol.

3.3 Content-Dependent File Partitioning

So far, we have considered splitting files into blocks of the same length b by just partitioning the file at locations $0, b, 2b, \ldots$, and then hashing the resultant blocks.

Content-dependent file partitioning provides a different approach to separating a file into blocks – it determines blocks based on local features of a file. In this way, sender and recipient can independently separate their files into blocks while still ensuring some degree of similarity between the chosen blocks, as long as the files are similar. After determining the locations to split the file into blocks, we can then hash each block individually and use the set of resultant hashes as the representation of the file.

There are several different content-dependent file partitioning techniques. We focus on two in particular. In Section 3.3.1 we go over point-filter chunking, and in Section 3.3.2 we go over winnowing. We adopt the latter in our file synchronization protocol, as it has stronger performance guarantees than the former and is reasonably simple to implement.

3.3.1 Point Filter Chunking

One content-dependent file partitioning approach [27] is to compute the hash of every b-gram (length b block) at every offset of the file, but only select a location to start a new block when the hash value is some constant $c \mod p$, for a fixed and predetermined integer p. Assuming a totally random hash function and a random file, this method has a *density*, defined to be the expected fraction of locations selected from among all the locations considered, of approximately $1/p$. One of the drawbacks of this method is that the maximum gap between two chosen locations is unbounded. That is, if the b-gram starting at position i happens to have a hash $= c \mod p$, we do not have any guarantees on the first integer $j > i$ such that the b-gram starting at position j is also chosen. This is the case since we could have that every b-gram starting after position i hashes to $d \mod p$ for $d \neq c$. Also, the minimum gap between two chosen locations can be one, which corresponds to a block of one character. Since we use the chosen locations to define block boundaries, and we then hash each resultant block, if locations i and $i + 1$ are both chosen, we end up creating a hash to represent just one character.

The Low Bandwidth File System [33] adopts a modified approach to point-filter chunking that ensures every block length is greater than a certain minimum length

m. It works as follows:

1. Skip the first m positions in the file since those would result in a block of length $< m$

2. For each following position, if that position satisfies the equivalence relation for basic point filter chunking, select it, and then skip the next m positions because of the same reason as in Step 1. Otherwise, increment the position and repeat this step until the end of the file.

3.3.2 Winnowing

Winnowing [35] provides another solution to the problem basic point filter chunking has of potentially unbounded block length. We define a window of size w to be a series of w consecutive hashes of b-grams in the file. Winnowing guarantees that at least one hash (and hence location) from every window is chosen. Thus, if two files F_1 and F_2 share a substring s of length at least $2w + b - 1$, then since winnowing is guaranteed to pick two block locations within that substring, it will produce file partitionings such that one block from each of F_1's and F_2's partitionings correspond to the same substring of s.

The algorithm behind winnowing is simple: in each window, select the location corresponding to the minimum hash value. If there are multiple such hash values in a window, then choose the hash corresponding to the rightmost element in the window. The *density*, which we defined in the previous section, of such an algorithm is $\frac{2}{w+1}$. A more recent method, called the local maximum method [8], shares many similarities to winnowing, except that it also enforces a lower bound on the block length.

We use winnowing in our file synchronization protocol to determine a partitioning of the file, as it is quite straightforward to implement and has reasonable performance guarantees.

Chapter 4

Our Algorithms

With our coverage of set reconciliation techniques in Chapter 2 and file synchronization protocols in Chapter 3, we now have (more than) sufficient background to fully explain all steps of our new algorithms.

In Section 4.1, we go over our file synchronization protocol, and in Section 4.2, we explain an extension of our file synchronization approach for directory synchronization.

4.1 File Synchronization

4.1.1 Motivation

Our algorithm for file synchronization is motivated by the methods of `rsync` (discussed in Section 3.1) and string reconciliation using puzzles (discussed in Section 3.2.1), and hence shares many similarities with them.

We want to take advantage of transmitting hashes in place of actual file blocks when possible, as file blocks are generally much longer than hashes. We want to compare fewer hashes than `rsync`, as that enables us to use shorter hashes for the same hash collision rate, so we use content-dependent file partitioning to split the file. To further save in transmission costs, we use set reconciliation to only send hashes corresponding to file blocks that are not already in both locations. And as in the de

Bruijn digraph approach of string reconciliation using puzzles, we can choose blocks from our file partitioning to have a few characters of overlap, so that only a few blocks can follow any given block.

4.1.2 Protocol

Our protocol involves a few main steps – partitioning the file and hashing the corresponding blocks, estimating the size of the difference between both sets of hashes, determining the hashes that differ, and then using that information to recreate the file. The exact protocol is explained below:

Say Party A wants to get an updated version of a file from Party B.

1. Party A use a content-dependent file partitioning technique (in our case, winnowing, discussed in Section 3.3.2) to partition its file f_A into distinct blocks. Each of those blocks is then hashed to form a set of hashes S_A. Party B does similarly with f_B to form a set S_B.

2. Parties A and B each create Strata Estimators (discussed in Section 2.3.1) E_A and E_B from their sets S_A and S_B. Party A transmits E_A to Party B, who then uses both Strata Estimators to get a tight upper bound on the size of the set difference d. Party B transmits the value d to Party A.

3. Parties A and B create IBLTs (discussed in Section 2.2.3) I_A, I_B with a number of buckets chosen to be able to successfully list d entries with high probability, and insert S_A and S_B into their respective IBLTs. Party A transmits I_A to Party B, who then computes $I = I_B - I_A$ and performs a `listing` operation on I to determine all the hashes in $S_A \setminus S_B$ and $S_B \setminus S_A$. Since Party B knows S_B, $S_B \setminus S_A$, and $S_A \setminus S_B$, it can then determine $S_A \cap S_B = (S_B \setminus (S_B \setminus S_A))$ and $S_A = (S_A \setminus S_B) \cup (S_A \cap S_B)$.

4. Party B then transmits the following to Party A:

   1) `blockExists`: A vector of booleans, one for each block b_i created for f_B, saying whether b_i was also a block created for f_A. We determine this by seeing

if the hash corresponding to b_i is in S_A. These booleans are ordered by the block's position in f_B.

2) `blockContents`: For each boolean that is false in `blockExists`, the corresponding block's contents (raw bytes).

3) `blockEncoding`: For each boolean that is true in `blockExists`, the index (in the sorted list of hashes for Party A) of the hash corresponding to that matched block. We can determine the sorted list of hashes for Party A since Party B can reconstruct all the hashes in S_A, as explained previously.

4) `checksum`: An MD5 hash of f_B to ensure that Party A correctly reconstructs the file.

5. Party A recreates the file as follows: For each boolean in `blockExists`, if it is false, then look at `blockContents` to get the appropriate block bytes. If it is true, look at `blockEncoding`, find the corresponding hash using the sorted list of Party A's hashes, and then get the corresponding bytes (by looking through f_A). Party A then computes an MD5 hash on the resultant file and ensure that it matches with `checksum`. If it does not, then we signal to Party B to transmit f_B in its entirety.

Computational Optimizations

To make this more computationally efficient, Parties A and B each create a data structure mapping their own block hashes to the corresponding block's offset within the file. In this way, each party can quickly retrieve the bytes corresponding to a certain block's hash value, as long as that hash is within that party's set of hash values. Each party also create a sorted list of its own block hashes so that it can quickly find the hash corresponding to a certain index from `blockEncoding`. To make this protocol more bandwidth efficient, all information sent over the wire is compressed using zlib.

Theoretical Transmission Costs

We have three main objects we send over the wire – the Strata Estimator, the IBLT, and the file reconstruction info. We consider the space cost of each below:

1. The Strata Estimator is of some fixed cost (it does not depend on the file size or the blocks), so for now we will just say that it is SE bytes. We will be more explicit about choosing its parameter settings later.

2. Let $|f_A|, |f_b|$ each be N bytes, let the hashes have length h bytes, let the average block size be B bytes, let the number of hashes unique to Party A or Party B be d, and let the IBLT we transfer have $c \cdot d$ buckets, where c is some constant. Each bucket of the IBLT has three fields, `keysum`, `hashsum`, and `count`. Our keys in this case are the block hashes, which have length h bytes. Assuming `hashsum` and `count` are also of the same size, the total size of our IBLT is $c \cdot d \cdot 3h$.

3. For `blockExists`, we need to send $\approx N/B$ booleans (one bit each). For `blockContents` (all the unmatched blocks, $\approx d$ of them), we need to send the block bytes ($\approx B$ bytes for each block). For `blockEncoding` (all the matched blocks, $\approx (N/B - d)$ of them), we need to send the indices of the hashes, which take $\approx \log(N/B - d)$ bits each. These are all approximations since content-dependent file partitioning does not produce blocks all of exactly the same size B.

In total, we need to send approximately

$$SE + c \cdot d \cdot 3h + \frac{N}{8B} + d \cdot B + \frac{N/B - d}{8} \log(N/B - d)$$

bytes. For our method to be beneficial, we need this to be less than N, the number of bytes in an individual file. Unfortunately, we cannot optimize directly for B based solely on this information, as B and d are correlated. In Section 4.1.3 we will assume a few models and see how the optimal B changes.

Note that in step 4 of our protocol, instead of sending the index of the corresponding hash for `blockEncoding`, we could just send the hash itself. However, with a small calculation as shown below, we see that it is nearly always optimal to send the index. Sending the index is better when

$$\frac{\log(N/B - d)}{8} < h$$

or

$$N/B - d < e^{8h}$$

With a four-byte hash, as long as we have fewer than $\approx 10^{13}$, or roughly 10 trillion, blocks, our method is better. If we consider a more realistic case, with perhaps 100,000 blocks, then our method uses $\approx$ 16.6 bits vs the 32 for a four-byte hash, which is nearly a 2x improvement.

Transmission Optimizations

By changing our file partitioning method to do something more like the de Bruijn digraph approach of [7], we can actually do better than just sending the index of the hash in step 4 of our protocol. In our current approach, the blocks we form are disjoint (they have no overlap). In this section, let us assume that we choose our blocks so that they have l bits of overlap. As an example, if our file were 01001011, our previous partitioning might be $\{010, 01, 011\}$. Using an overlap of 1 bit, then our new partitioning would become $\{0100, 010, 011\}$.

The benefit of using overlapping blocks can be seen as follows. Using disjoint blocks, if a boolean is true in `blockExists` (corresponding to a matched block), then for `blockEncoding`, we always have to send the index of the corresponding hash in the sorted list of hashes for Party A. Instead, if the blocks have overlap, then if the previous block was also matched (i.e. the previous boolean in `blockExists` was also true), then we only need to send the index of the hash within a smaller list of hashes corresponding to the blocks that could follow the previous block (i.e. the last l bits of the old block must be the same as the first l bits of the new block). For instance, if we knew that the previous matched block was 0101 and the set of shared blocks is $\{011, 100, 111, 110, 0101\}$, and $l = 2$, then only blocks 011 and 0101 could follow block 0101, so instead of having to choose an index from 0-4, which takes $\lg 5$ bits, we would only need to choose an index from 0-1, which takes $\lg 2$ bits. The drawback of this approach is that each time a hash does not match up, we have to send an extra l bits per item in `blockContents`.

To estimate when using overlapping blocks decreases transmission cost, we once again assume a file of length N bytes and d blocks that differ. Now, we have blocks of length $(B + l)$, but still N/B blocks because of the overlap. We also assume that the block contents are completely random, so that having l bits of overlap decreases the size of the set of blocks that can follow by a factor of 2^l (this is a pretty large assumption to make for real data sets, which are far from random, but we do so for simplicity). With those assumptions in mind, the size of `blockEncoding` is

$$N/(8B) + d \cdot (B + l/8) + \frac{(N/B - d)}{8} \log\left(\frac{N/B - d}{2^l}\right)$$

We want this to be less than the previous value of

$$N/(8B) + d \cdot B + \frac{(N/B - d)}{8} \log(N/B - d)$$

Subtracting, we find our new method is better when

$$dl/8 < (N/B - d)l/8$$

which is true as long as $(N/B - d) > d$, and should be the case given that the two files are very similar. Thus, we adopt blocks with overlap in our protocol.

4.1.3 Choosing Parameter Settings

As shown above, we have many parameters to tweak in this model. First, we have the content-dependent file partitioning and hashing step. We need to determine the average block size we want to create and the size of each hash. Second, we have the Strata Estimator. We need to determine the number of IBLTs in the Strata Estimator and the size of each IBLT. Third, we have the IBLT transferred from Party A to Party

B. We need to choose the number of hash functions and also the size of the IBLT relative to the size of the set difference.

Determining the Block Size

One of the most important problems in this protocol is determining an appropriate block size. If it is too small, then we will end up matching the majority of blocks, but Party B will end up sending long vectors for `blockExists` and `blockEncoding`. However, this is not terrible in our setting, as our transferred IBLT will only contain the unmatched keys. If the block size is too large, then many block hashes will not match, and we will end up sending large amounts of file data. We analyze two different block size estimation methods based on two different models of the file difference. Note that we do not include the transmission optimizations (block overlap) in these calculations.

Approach 1: Random Error Model

In this section, we will perform some back-of-the-envelope calculations to determine an appropriate block size for our model. We will assume a random error model, where Party B's file is a replica of Party A's, except each character has a probability $(1-p)$ of differing.

As in our analysis of theoretical transmission costs, we will assume that the file is of length N bytes, and our hashing technique allows us to split the file into blocks of B bytes, for a total of N/B blocks. The hashes will be of length h bytes.

The probability of one block being completely unchanged is p^B (each of the B characters cannot differ). Thus, the number of unchanged blocks is $\frac{N}{B} \cdot p^B$ and the number of differing blocks d is $\frac{N}{B}(1-p^B)$. Plugging in, we find that T, the expected total number of bytes transmitted is

$$T = SE + 3c \cdot h\frac{N}{B}(1-p^B) + \frac{N}{8B} + \frac{N}{B}(1-p^B) \cdot B + \frac{N}{8B} \cdot p^B \log\left(\frac{N}{B} \cdot p^B\right)$$

The first term comes from the Strata Estimator, the second from the transferred

IBLT, the third from `blockExists`, the fourth from `blockContents`, and the fifth from `blockEncoding`. The log term is annoying computationally since it does not scale linearly with N as is the case for the other terms. To simplify our calculation, we assume that we use the actual hash value rather than the index, and so replace $\frac{1}{8}\log\left(\frac{N}{B}\cdot p^B\right)$ with h. Our simplified version is:

$$T = SE + 3c \cdot h\frac{N}{B}(1-p^B) + \frac{N}{8B} + \frac{N}{B}(1-p^B)\cdot B + \frac{N}{B}\cdot p^B h$$

Taking the derivative of T with respect to 0 and setting it to 0 (and noting that the dependence on N cancels out), we find that to minimize transmission complexity, we need

$$\left(B^2\ln(p) + (3c-1)h\cdot B\cdot\ln(p) - (3c-1)h\right)p^B = -\frac{1}{8}(24ch+1)$$

If we set $c = 2$ (number of IBLT buckets = 2· estimated set difference) and $h = 4$ (4 bytes per hash), this becomes

$$\left(B^2\ln(p) + 20B\ln(p) - 20\right)p^B = -\frac{1}{8}(192+1)$$

As an example, if $p = 0.99$ (each character has a 1% chance of differing), we get that the optimal value B^* is ≈ 24 bytes. If instead $p = 0.999$, then $B^* \approx 67$ bytes.

If we send the index instead of the hashes (and assume that the file is of a reasonable length, say 10MB), then the optimal values actually differ quite a bit. If $p = 0.99$, $B^* \approx 15$, and if $p = 0.999$, $B^* \approx 43$. It makes sense that these values should be lower – the penalty for matching a hash is lower since the index is generally much smaller than the hash value itself.

Table 4.1 showing the optimal block size and percentage of data transferred (relative to transferring the whole file for a 10MB file) for a range of values of p. We do not include the fixed cost of the Strata Estimator. Below $p = 0.9$, our method does not save much – in those situations, it is too likely that a small block will contain a changed character.

Table 4.1: Random Error Model: Optimal Block Size and Data Transferred using IBLT method vs Full File Transfer

p	Optimal Block Size	% Data Transferred
70.000%	4.8	92%
80.000%	5.1	80%
90.000%	5.5	64%
95.000%	7.3	49%
99.000%	14.8	24%
99.500%	20.3	18%
99.900%	43.2	8%
99.950%	60.0	6%
99.990%	129.5	2%
99.995%	180.5	2%
99.999%	390.4	1%

Approach 2: Block Error Model

In this approach, we assume that the two files are the same except for d sequences of bytes, with each sequence smaller than the block size. This is perhaps a more reasonable model of changes to an actual file, as a user will generally make a small number of changes of larger length.

In this situation, our equation for the total number of bytes transferred is:

$$SE + c \cdot d \cdot 3h + \frac{N}{8B} + d \cdot B + \frac{1}{8}(N/B - d)\log(N/B - d)$$

Table 4.2 shows the results with $c = 2, h = 4$ and $N = 10^7$ bytes (10 megabytes), by varying the size of d. Note our assumption that each sequence is smaller than the block size – in reality d and B are not actually independent, as the smaller the block, the more likely it is that changed sequences of bytes will cross a block boundary.

Determining the Hash Size

We can use a similar analysis to the one we did for `rsync` in Section 3.1.1 to determine an appropriate hash size for a certain false positive rate. Here, each party creates $\approx N/B$ block hashes. Say the hashes have length x. Assuming a perfectly random

Table 4.2: Block Difference Model: Optimal Block Size and Data Transferred using IBLT method vs Full File Transfer

d	Optimal Block Size	% Data Transferred
1	3527	0.07%
10	1175	0.23%
100	390	0.77%
1000	129	2.70%
10000	42.3	10.40%
100000	13.8	49.00%

hash function, the probability of a given pair of hashes conflicting is $1/2^x$.

If any of the block hashes conflict, then we are in trouble, since the file recreation info assumes that each hash is uniquely mapped to a specific block of bytes. In total we have $2N/B$ blocks and corresponding hashes (N/B from each file), and we want to find the probability that no two blocks accidentally map to the same hash value. We have $\binom{2N/B}{2} \approx 2N^2/B^2$ pairs of hashes. Using the Poisson Approximation to the Binomial [32], we have that the distribution of the number of conflicting hashes follows

$$X = Pois\left(\lambda = 2\left(\frac{N}{B}\right)^2 \cdot \frac{1}{2^x}\right)$$

So the probability that we have no conflicting hashes:

$$P(X = 0) = \exp\left(-\left(\frac{N}{B}\right)^2 \cdot \frac{1}{2^{x-1}}\right)$$

Setting this equal to $1 - 2^{-d}$, where 2^{-d} is our false positive rate, and using the approximation $e^x \approx 1 + x$, which is valid when x is small, our equation becomes:

$$2^{-d} = \frac{(N/B)^2}{2^{x-1}}$$

so we need $\approx 2\lg_2(N/B) + d + 1$ bits to have a false positive rate of 2^{-d}. Note that we need fewer bits than `rsync` as we are creating fewer hashes.

As an example, if we have 1000 blocks and want a false positive rate of one in a million, we need approximately 41 bits. Assuming the same false positive rate of one in a million, with 32 bits we can have ≈ 23 hashes, and with 64 bits we can have $\approx$ 3 million hashes.

Determining the Strata Estimator and IBLT parameters

As described in Section 2.3.1, a Strata Estimator is a data structure consisting of multiple small Invertible Bloom Lookup Tables that can be used to estimate the size of a set difference, given that the size of the set difference is less than some number u. More specifically, Strata Estimator consists of $n = \log u$ IBLTs each of some fixed number of buckets b. Indexing the IBLTs as $I_{A1}, I_{A2}, \ldots, I_{An}$, Party A inserts $\approx 1/2^i$ of its keys to I_{Ai}. Party B does similarly with its own set of keys and its own Strata Estimator consisting of IBLTs $I_{B1}, I_{B2}, \ldots, I_{Bn}$. We can use the number of successfully peeled keys from $I_i = I_{Ai} - I_{Bi}$ for each i as a proxy for the number of keys in the set difference.

We choose a Strata Estimator with 32 IBLTs, which allows us to estimate the size of set differences up to $\approx 2^{32}$ (note that given how we partition files and subsequently hash the blocks, 32 IBLTs allows for set difference sizes much larger than any likely set difference). We choose each IBLT in the Strata Estimator to contain 80 buckets. The 80 buckets is based off of the results of Eppstein et al. [18], which showed that 80 buckets seemed to ensure good set difference estimates across a wide range of set difference sizes. As Eppstein et al. found, using four hash functions seemed to work best experimentally for a wide range of set difference sizes, so we default to four hash functions. There is some intuition for why four hash functions might be optimal: With a smaller number of hash functions, each time we find a key to list, we only get to remove the key from a small number of buckets, which limits the number of new buckets that will have a `count` of 1. If we have many hash functions though, the chance that we start out with no bucket with a `count` of one is higher.

Using a Strata Estimator with 80 buckets per IBLT, Eppstein et al. found that scaling the estimate returned by the Strata Estimator by ≈ 1.4 was enough to ensure that it was greater than the actual set difference 99% of the time. Per our results in Section 2.2.3, if t is the number of keys that we want to be able to list, then if we choose 4 hash functions for the IBLT, if our IBLT has $> 1.295t$ buckets, `listing` operations will succeed with high probability. Factoring in the multiplier from the Strata Estimator, we choose to create an IBLT with $2e$ buckets, where e is the estimate returned by the Strata Estimator.

Note that the overhead of the Strata Estimator is quite high – if the three fields in each IBLT bucket (`count`, `keysum`, and `hashsum`) are each 4 bytes, then the Strata Estimator is $32 \cdot 80 \cdot 4 \cdot 3 = 32720$ bytes. Unless the file sizes are quite large, the fixed cost of the Strata Estimator will exceed any benefit from transferring fewer blocks. Since the Strata Estimator's sole role is getting a tight upper bound on the size of the set difference, we could adopt other approaches. If the file is small, then we could adopt the approach from repeated doubling (see Section 2.3.3) and not even try to determine the set difference, but instead start by transmitting a small IBLT and doubling its size until a `listing` succeeds. If the set difference is sufficiently large, min-wise sketches (see Section 2.3.2) could provide more accurate set difference estimates using the same number of bytes as a Strata Estimator.

4.2 Extension: File Directory Synchronization

We can also use a similar method to our individual file synchronization protocol above for more efficient file directory synchronization. The main idea is that we can think of all the files in a directory as a set of (file name, file contents) pairs. Note that these pairs are all distinct if we consider the file name to be the file' s full path from the root directory. Thus, we can use set reconciliation techniques to determine which files have been changed, and only transmit the changed files. Note that here we do not consider permission bits or other metadata.

4.2.1 Protocol

The setup is pretty much the same as before. We have Parties A and B, with directories D_A and D_B, and Party A wants to update its directory to be the same as

Party B's.

1. Parties A and B go through each of the files in their directory, computing a hash of the file's full path, which we call the `pathHash`, and a hash of the file contents, which we call the `contentHash`. Each file's `pathHash` and `contentHash` are then concatenated to form a key representing that file. Those keys become the elements of the sets S_A and S_B.

2. Following the protocol described for individual file synchronization in Section 4.1.2 (using Strata Estimators and IBLTs), S_A and S_B are reconciled to determine $S_A \setminus S_B$ and $S_B \setminus S_A$.

3. Party B goes through all the keys in $S_B \backslash S_A$ and parses each back into (`pathHash`, `contentHash`) pairs, which we abbreviate as (ph, ch). It then sends a message containing three different components to Party A:

   (a) For each $(ph_i, ch_i) \in S_B \setminus S_A$, Party B looks through each of the (ph'_i, ch'_i) pairs in $S_A \setminus S_B$.

   `renameInfo`: If there is a ch'_i such that $ch'_i = ch_i$, then we know (with high probability, assuming no hash collisions in `contentHash`) that Parties A and B have files with the same contents but with different filenames. If `contentHash` is sufficiently small, we may want to transmit an additional checksum to ensure that both files' contents are indeed the same. If `contentHash` is a strong hash, say a 160-bit SHA-1 hash, then the chance of a hash collision is extremely small. Let fn_i be the file name corresponding to ph_i. Then Party B transmits (ph'_i, fn_i) to Party A.

   `newFileInfo`: If there is not a ch'_i such that $ch'_i = ch_i$, then (ph_i, ch_i) must correspond to a new file. Letting that file's name be fn_i and its contents be fc_i, Party B transmit the new file to Party A in the form (fn_i, fc_i).

   (b) `deleteInfo`: For each $(ph_i, ch_i) \in S_A \setminus S_B$ that was not matched in `renameInfo`, Party B transmits a delete message with ph_i. These correspond to files that are in S_A but not in S_B, and thus must be removed for Party A's directory to match Party B's.

4. Party A reconstructs the directory as follows. For each (ph'_i, fn_i) in `renameInfo`, Party A finds its file f_i corresponding to the path hash ph'_i and renames it to fn_i. For each (fn_i, fc_i) in `newFileInfo`, Party A creates a file with name fn_i and contents fc_i, and adds it to its directory. For each ph_i in `deleteInfo`, Party A deletes the file corresponding to ph_i.

Step 4 can be a bit complicated if there is a series of renames that overlap (for instance, if we need to rename f_1 to f_2, f_2 to f_3, and f_3 to f_1). One method to solve this problem is to create temporary copies of all the files in the directory and then remove the original files. We perform the same operations as enumerated in Step 4, except using the temporary files. Temporary files that did not have any operations performed on them are then copied back to their original state. In this way, we will not run into any naming conflicts, and do not need to worry about the order in which we perform renames.

Another way to get around the problem of overlapping renames is to create a directed acyclic graph of the dependencies between files. We first go through all the `deleteInfo`, as those files could be preventing our renames. We then choose an arbitrary file f that needs to be renamed (say to f'). We then see if f' needs to be renamed (say to f''). If it does, then we check if f'' needs to be renamed, and so on. If we ever reach a file f^* such that $f^{*'}$ (the file it should be renamed to) has already been traversed, we move f^* to a temporary file (so now we have a linear dependency graph). We then work backwards to rename files that have no dependencies. Finally, we rename the temporary file to its appropriate name.

As an example, in our situation before, where we wanted to rename f_1 to f_2, f_2 to f_3, and f_3 to f_1, if we started with f_2, then we would form the path $f_2 \rightarrow f_3 \rightarrow f_1 \rightarrow f_2$. Since f_2 was already traversed, f_1 would be moved to $tmpf_1$, then we would move f_3 to f_1, f_2 to f_3, and then move $tmpf_1$ to f_2.

Chapter 5

File Synchronization Experiments

In this chapter, we provide an empirical evaluation of our file synchronization protocol described in Chapter 4.

5.1 Setup

We run three different sets of experiments, and for each compare our file synchronization algorithm, which we call `IBLTsync`, with `rsync`, described in Section 3.1 and `naïvesync`, a method that just sends over all the new file bytes in compressed form.

For a fair comparison, we use the same compression library for all three algorithms (zlib with the maximal level of compression). To serialize messages (Strata Estimator, IBLT, etc.) for `IBLTsync`, we use Google's Protocol Buffer [4]. We do this so that we can more easily put these structures into bitstring representations that can then be compressed.

We default to a Strata Estimator (see Section 2.3.1 and Section 4.1.3) with 32 IBLTs, each with 80 buckets. We choose four hash functions per IBLT, which Eppstein et al. [18] found to perform well empirically for set difference sizes ranging from 10 to 1 million. We choose `keysum` to be 64-bits (which ensures a collision probability of less than one in a million even with 3 million+ keys), choose `hashsum` to be 32-bits (so that the likelihood of an erroneous key being listed is $\approx 1/2^{32}$), and choose `count` to be 32-bits (so we can support more than a billion keys inserted to the IBLT). Note that we could tailor the sizes of each of these fields on a case-by-case basis (based on individual files), which would be more bandwidth-efficient, though we do not do so here.

We include our optimizations involving block overlap from Section 4.1.2. We choose the overlap l so that there is usually just one block that can follow a given block. For this to be the case, $2^l > N/B - d$, which upon rearranging gives us $l > \lg_2(N/B - d)$. Choosing a longer overlap does not add any benefit since we must minimally transmit one bit per following block.

Each graph in the sections below has three lines, *IBLT (no strata)*, *IBLT (with strata)*, and *rsync*. *IBLT (no strata)* corresponds to the transmission cost assuming that we already have a correctly sized IBLT, so that we do not need to use the Strata Estimator to estimate the size of the set difference. *IBLT (with strata)* corresponds to the transmission cost when we use a Strata Estimator to estimate the size of the set difference. We create these two different lines so that we can separate out the cost of Strata Estimation, as there are situations where we might not choose to use a Strata Estimator (see Section 2.3 and Section 4.1.3). Since our experiments below use quite large files (10MB), the lines corresponding to *IBLT (with strata)* and *IBLT (no strata)* are nearly the same line in many cases. *rsync* corresponds to the transmission cost using `rsync`.

We describe each set of experiments in more detail below.

5.2 Random Error Model

We test the random error model, where each character in a file is independently changed with probability $1-p$, with 10MB files and p values of $99\%, 99.9\%, 99.99\%$, and 99.999%. For each chosen value of p, we plot the transmission cost for a wide range of block sizes.

Tables 5.1 and 5.2 shows the performance of `IBLTsync` in comparison to `rsync` and `naïvesync` for the random error model. At an error probability of 1%, `IBLTsync` transfers roughly 12% more data than `naïvesync` and 60% more data than `rsync`.

Table 5.1: Random Error Model: Bytes Transferred for 10MB File

Error Probability	IBLT (w/o strata)	IBLT (w/ strata)	rsync	Naïve Transfer	Best block IBLT	Best block rsync
1.000%	8,447,528	8,461,422	5,327,823	7,547,299	140	30
0.100%	1,223,254	1,239,447	1,663,608	7,542,920	50	90
0.010%	179,051	194,798	477,522	7,542,174	60	320
0.001%	26,134	39,995	125,731	7,542,075	150	970

Table 5.2: Random Error Model: IBLT performance relative to rsync and naïve transfer

Error Probability	IBLT (as % of rsync) w/o strata	IBLT (as % of rsync) w/ strata	IBLT (as % of naïve transfer)
1.000%	158.6%	158.8%	112.1%
0.100%	73.5%	74.5%	16.4%
0.010%	37.5%	40.8%	2.6%
0.001%	20.8%	31.8%	0.5%

This is the case since most blocks will be unmatched, and the size of the set difference will be large. Consequently, the transferred IBLT structure will be large. Furthermore, content-dependent file partitioning does not ensure that blocks will all be of the same size. The variance in the block length likely contributes to the poor performance of IBLTsync.

At an error probability of 0.1%, IBLTsync transfers significantly less data than naivesync (83.6% less, when including the Strata Estimator) and marginally less data than rsync (25.5% less). As the error probability decreases, IBLTsync performs better and better relative to rsync. Including the Strata Estimator overhead, at an error probability of 0.01%, IBLTsync transfers 40.8% of rsync's data, and at an error probability of 0.001%, IBLTsync transfers only 31.8% of rsync's data. If we exclude the cost of the Strata Estimator, then at an error probability of 0.001%, IBLTsync only transfers 21% of rsync's data. Thus, our approach seems to work better relative to other approaches as the similarity between the two files increases, which is our desired outcome.

Figures 5-1, 5-2, 5-3, and 5-4 show the performance of IBLTsync in comparison

Figure 5-1: Random Error Model with P(err) = 1% and 10MB File

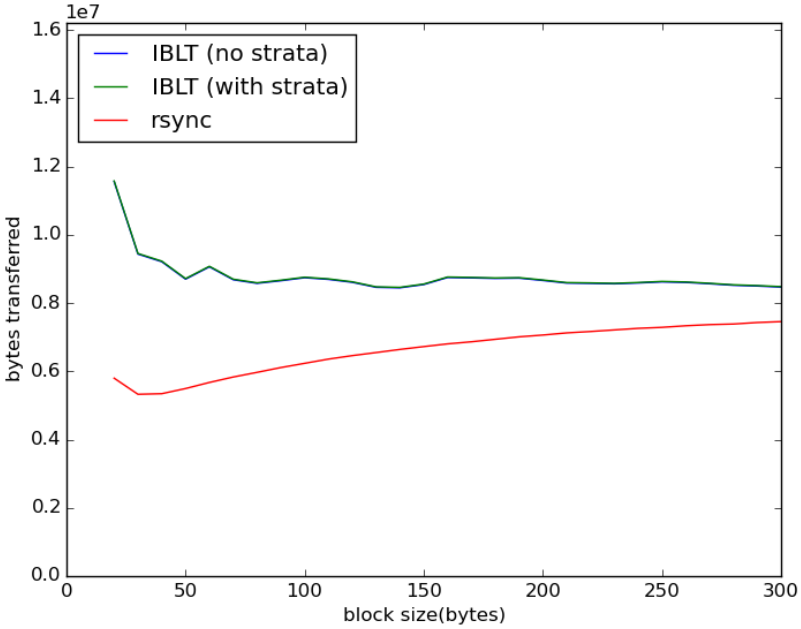
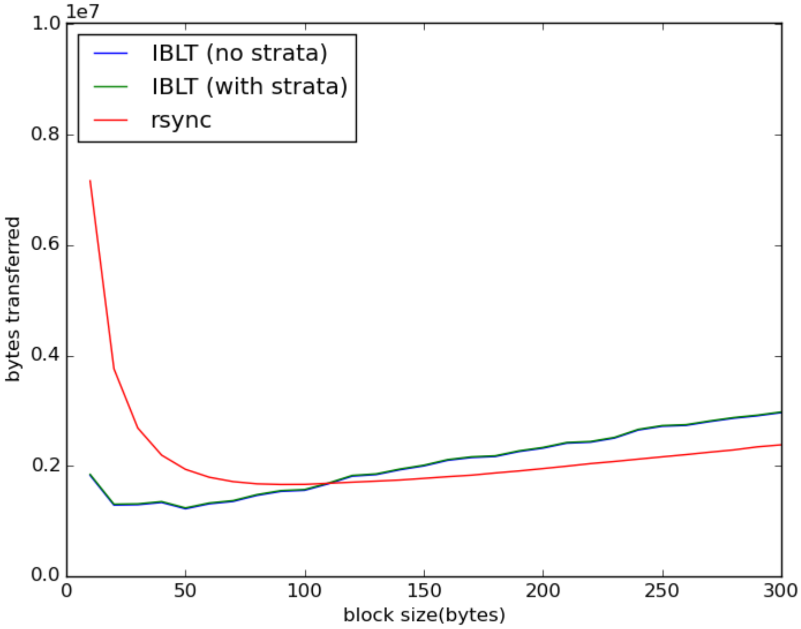


Figure 5-2: Random Error Model with P(err) = 0.1% and 10MB File



to `rsync` for the random error model and differing values of the block size. Since the files are 10MB, the overhead of the Strata Estimator is negligible, so we see that both `IBLTsync` lines closely track each other. As all the figures show, the optimal block size

Figure 5-3: Random Error Model with P(err) = 0.01% and 10MB File

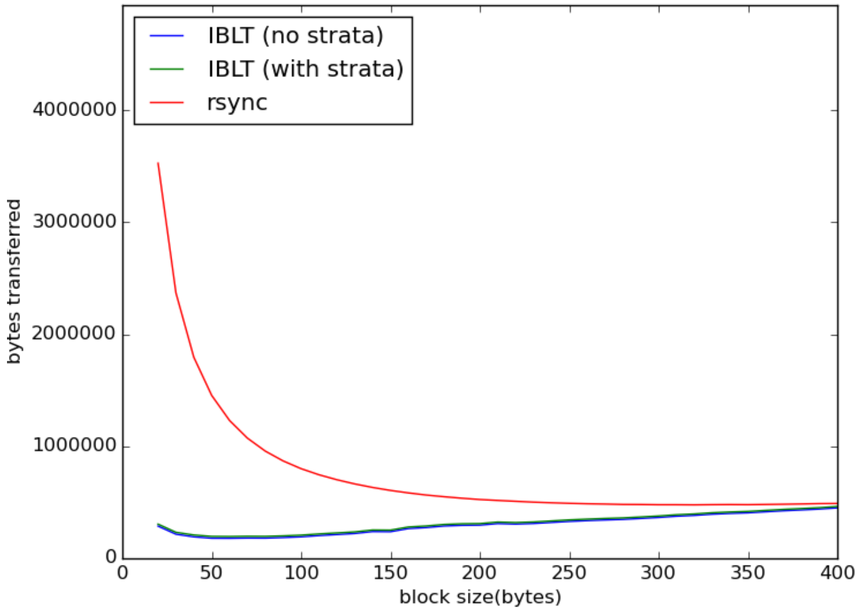
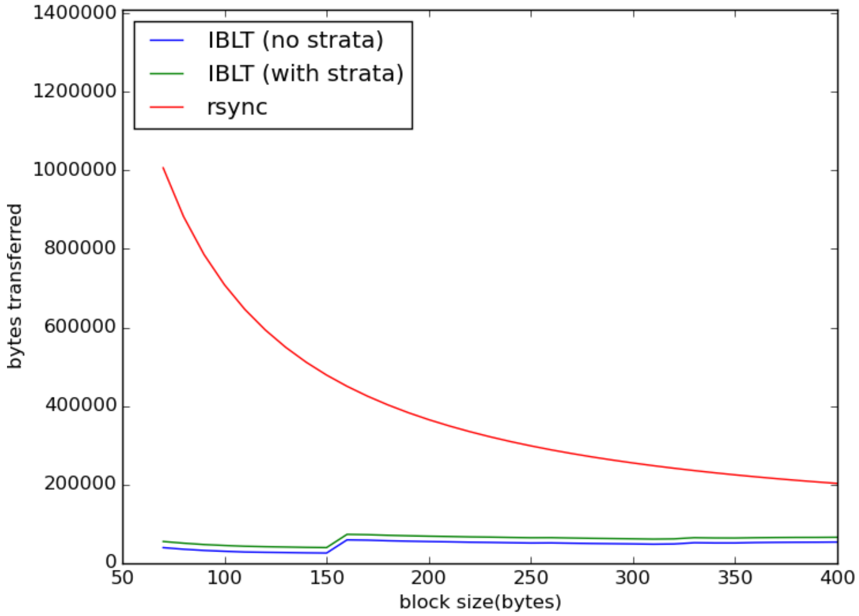


Figure 5-4: Random Error Model with P(err) = 0.001% and 10MB File



for `IBLTsync` tends to be much smaller than the corresponding size for `rsync`. This is the case because `rsync` has to transmit a full hash each time two blocks match. Instead in `IBLTsync`, when two blocks match, we just have to send an index, which tends to be much smaller. Since decreasing the block size increases the number of hashes transmitted for `rsync`, our method will perform better relative to `rsync` for smaller block sizes.

When the error probability is 1%, as in Figure 5-1, we can see that `rsync` outperforms `IBLTsync` at all block sizes. As we explained before, `IBLTsync` is tailored for very similar files, so it unsurprisingly gets beaten when the files are not that similar. When the error probability is 0.1%, as in Figure 5-2, `IBLTsync` is better for block sizes up to 120. With an error probability of 0.01%, `IBLTsync` is better for block sizes up to 400, and with an error probability of 0.001%, `IBLTsync` is better for all block sizes. Based on the results above, it seems that when files are sufficiently different, `rsync` is probably a better approach, but when they are sufficiently similar, `IBLTsync` can provide transmission benefits.

5.3 Block Error Model

Second, we test the block error model, where there is a fixed number of changes that happen in blocks. We tested the block error model with 10MB files, and 10, 100, 1000, 10000, and 100000 changes of length 5 bytes each. For each chosen number of changes, we plotted the transmission cost for a wide range of block sizes. Since the changes happen randomly throughout the file, we can think of this as having a block change approximately every 1000000, 100000, 10,000, 1000, and 100 bytes respectively.

Table 5.3: Block Error Model: Bytes Transferred for 10MB File

Number of Changed Blocks	IBLTsync (w/o strata)	IBLTsync (w/ strata)	rsync	naïvesync	Best block IBLTsync	Best block rsync
10	13,904	24,582	43,002	7,542,079	1,100	2,782
100	32,580	46,582	138,333	7,542,122	136	901
1000	183,681	199,745	492,344	7,542,536	55	325
10000	1,542,447	1,558,885	1,708,466	7,545,150	40	90
100000	7,933,327	7,944,065	5,558,934	7,559,802	973	37

Table 5.4: Block Error Model: Bytes Transferred for 10MB File

Number of Changed Blocks	IBLTsync (as % of rsync) w/o strata	IBLTsync (as % of rsync) w/ strata	IBLTsync (as % of naïvesync)
10	32.3%	57.2%	0.3%
100	23.6%	33.7%	0.6%
1000	37.3%	40.6%	2.6%
10000	90.3%	91.2%	20.7%
100000	142.7%	142.9%	105.1%

Tables 5.3 and 5.4 show the performance of `IBLTsync` in comparison to `rsync` and `naïvesync` for the block error model. With only 10 blocks changed, `IBLTsync` transfers roughly 0.3% of `naïvesync`'s data, and 57.2% of `rsync`'s data (including the Strata Estimator), and 32.3% of `rsync`'s data (when not including the Strata Estimator). `IBLTsync` actually improves relative to `rsync` for 100 blocks changed, transferring 23.6% of `rsync`'s data (without SE) and 33.7% of `rsync`'s data (with SE). At 1000 block changes, `IBLTsync`'s performance drops relative to `rsync`'s, transferring 37.3% of `rsync`'s data when excluding the Strata Estimator. Interestingly, when including the Strata Estimator, `IBLTsync` transfers a smaller proportion of `rsync`'s bytes at 1000 changed blocks as compared to 10 changed blocks (57.2% in the former case and 40.6% in the latter case). This happens since the total amount of data transferred for 10 blocks is so small that the size of the Strata Estimator is a sizeable chunk of the total amount of data transferred. At 10000 block changes, `IBLTsync` is slightly better than `rsync`, transferring approximately 90% of `rsync`'s data, both with and without the Strata Estimator. At 100000 block changes, changes are so frequent that `IBLTsync` performs extremely poorly, transferring approximately 40% more data than `rsync` and 5% more data than `naïvesync`.

Figures 5-5, 5-6, 5-7, 5-8, and 5-9 show the performance of `IBLTsync`, both with and without the Strata Estimator, in comparison to `rsync` for the block error model and differing values of the block size. Once again, since the files are 10MB, the overhead of the Strata Estimator is negligible, so we see both `IBLTsync` lines closely tracking each other. Also as before, the optimal block size for `IBLTsync` tends to be

Figure 5-5: Block Error Model with # errors = 10 and 10MB File

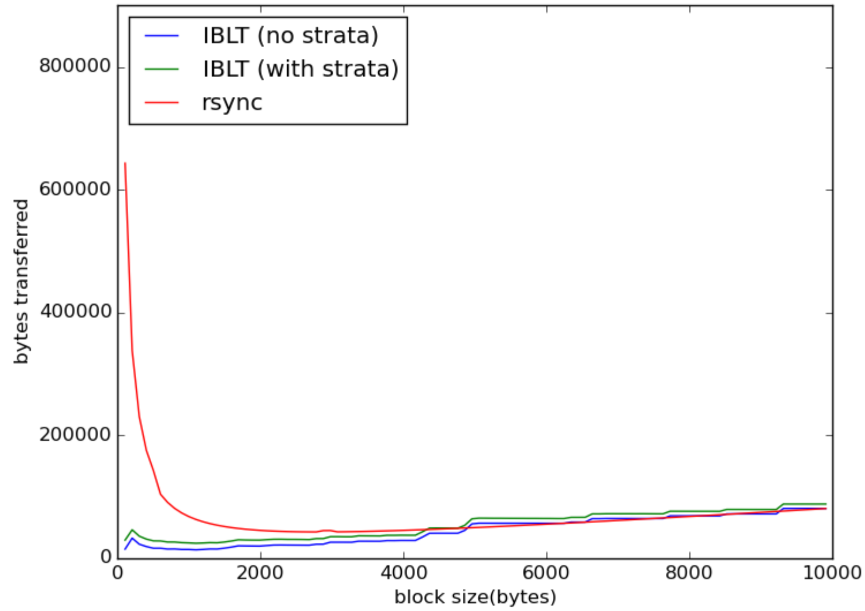
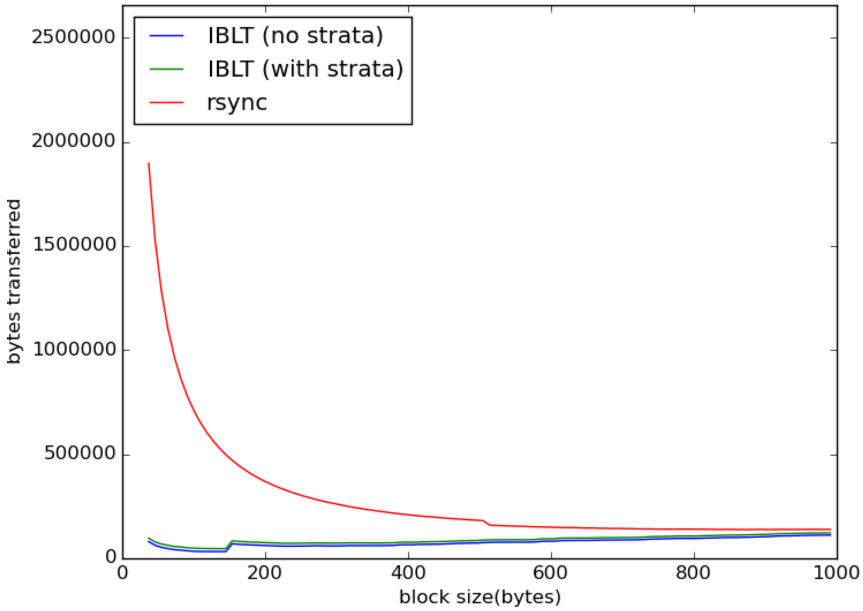


Figure 5-6: Block Error Model with # errors = 100 and 10MB File



much smaller than the corresponding size for `rsync`.

With only 10 block changes, `IBLTsync` outperforms `rsync` for block sizes up to approximately 5000. At that point, both `IBLTsync` and `rsync` closely track each

Figure 5-7: Block Error Model with # errors = 1000 and 10MB File

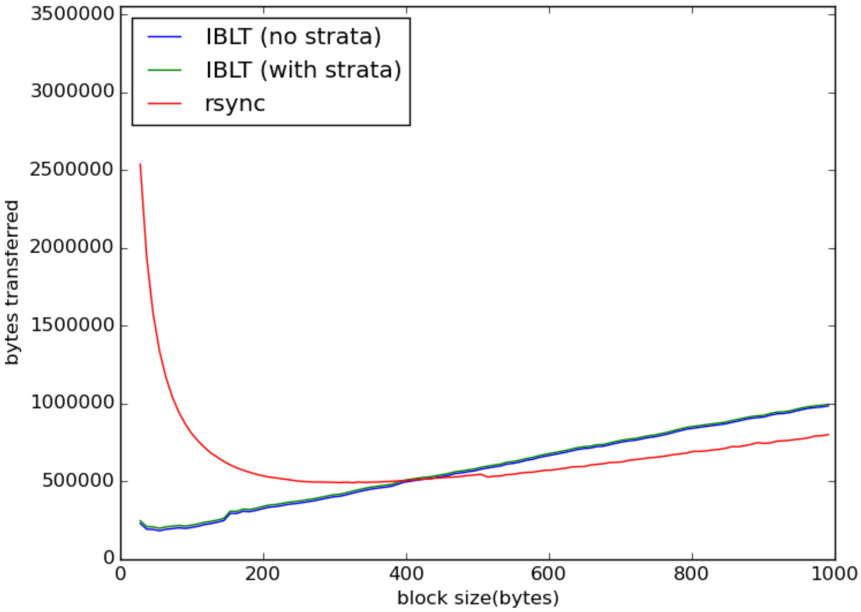
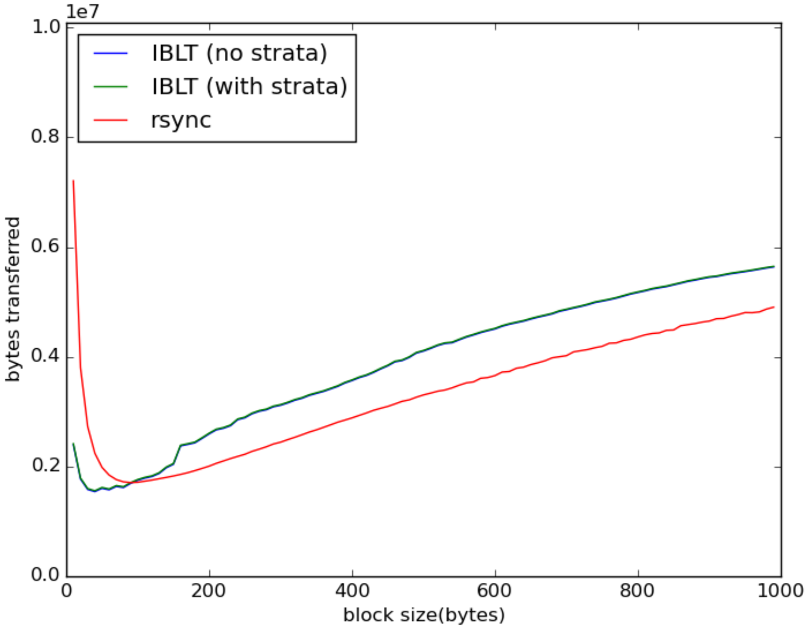
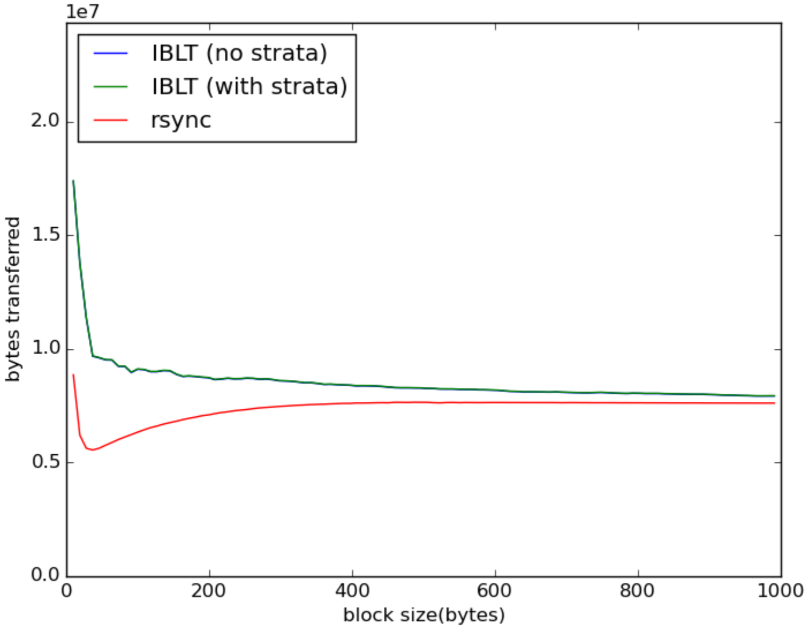


Figure 5-8: Block Error Model with # errors = 10000 and 10MB File



other. With 100 block changes, `IBLTsync` outperforms `rsync` for block sizes up to approximately 1000. With 1000 block changes, `IBLTsync` outperforms `rsync` for block sizes up to approximately 400. With 10000 block changes, `IBLTsync` outperforms

Figure 5-9: Block Error Model with # errors = 100000 and 10MB File



`rsync` for block sizes up to approximately 5000, and with 100000 changes, `IBLTsync` gets outperformed across the board.

5.4 Practical Workload

We also test a (perhaps more) realistic workload, which involves actual data sets based on different versions of source code directories we crawled from github. As our file synchronization protocol works best with larger files (because of the overhead of the Strata Estimator), we concatenate all the files in the source directory[1] and use those as individual files that we try to synchronize. We test our protocol on two source code directories, sharelatex [5] (≈ 100KB) and PredictionIO [6] (≈ 28 MB).

Table 5.5 shows the number of bytes transferred for `IBLTsync` as compared to `rsync` and `naivesync` for the aforementioned repositories. The file size difference is calculated as the difference in file size between the old repo and the new repo. For sharelatex, the file size difference is extremely small (only one line is changed), and thus `IBLTsync` transfers only 3.1KB. However here the cost of the Strata Estimator

[1] `find . -type f -print0 | sort -z | xargs -0 cat`

Table 5.5: Source Code Repositories: Bytes Transferred

Repository	IBLT (w/o strata)	IBLT (w/ strata)	rsync	Naïve Transfer	File size difference	Best block IBLT	Best block rsync
Sharelatex	3101	10308	6076	228216	80	400	900
PredictionIO	258036	269807	267485	24701960	1288633	1700	3574

is an extremely significant overhead (more than 2x the size of all the other structures combined). In situations like these, it might be better to take a repeated doubling approach – start with an IBLT of some fixed size c. If the `listing` operation fails, then double the IBLT size and repeat. For PredictionIO, the file size difference is quite significant, implying that there are most likely quite a few difference between the old and new versions. In this situation, the cost of the Strata Estimator is negligible. As we found in our previous experiments, we see that the best block size for `IBLTsync` is much smaller ($\approx$ 50%) than the best block size for `rsync`.

Table 5.6: Source Code Repos: IBLT performance relative to rsync and naïve transfer

Repository	IBLT (as % of rsync) w/o strata	IBLT (as % of rsync) w/ strata	IBLT (as % of naïve transfer)	IBLT (as % of file difference)
Sharelatex	51.0%	169.7%	4.5%	12885%
PredictionIO	96.5%	100.9%	1.1%	21%

Table 5.6 shows the relative performance of `IBLTsync` as compared to `rsync` and `naivesync`. For sharelatex, without including the Strata Estimator, we only transfer half the bytes of `rsync`. However, including the Strata Estimator causes us to transfer 1.5x the bytes of `rsync`. As explained in the previous paragraph, for sufficiently small files, some other method should be used to estimate the size of the set difference. For PredictionIO, `IBLTsync` is on par with `rsync` (96.5% of the bytes when not including the Strata Estimator, and 100.9% of the bytes when including). `IBLTsync` does not perform much better because the files are quite different – by the file size difference, they differ by at least $\approx$ 1.3MB. Interestingly, `IBLTsync` and `rsync` both transfer significantly fewer bytes than the file size difference. This is probably caused by a

mixture of compression and added blocks that are just repeats of preexisting blocks, for which we can just transfer the hash/index instead of the actual block.

Figure 5-10: Sharelatex: Cost of Updating Source Code from Version 0.1.0 to 0.1.1

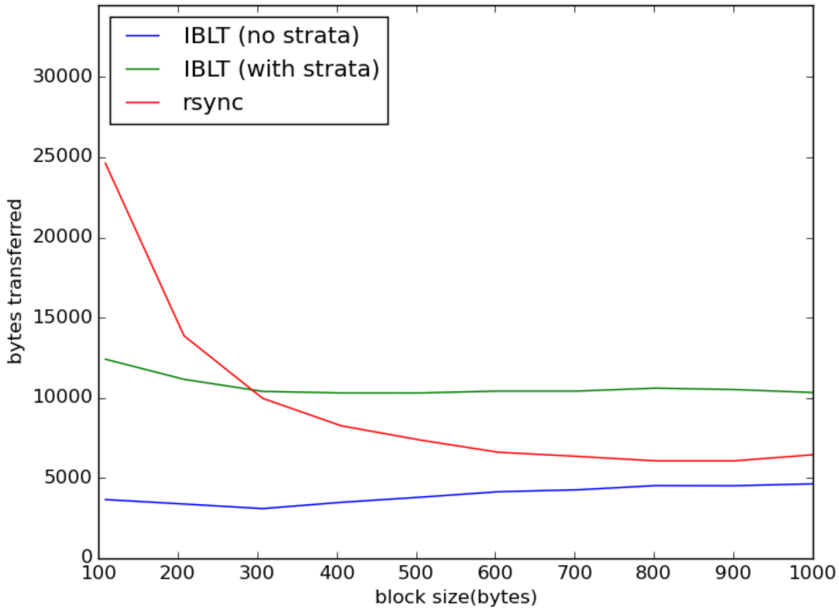
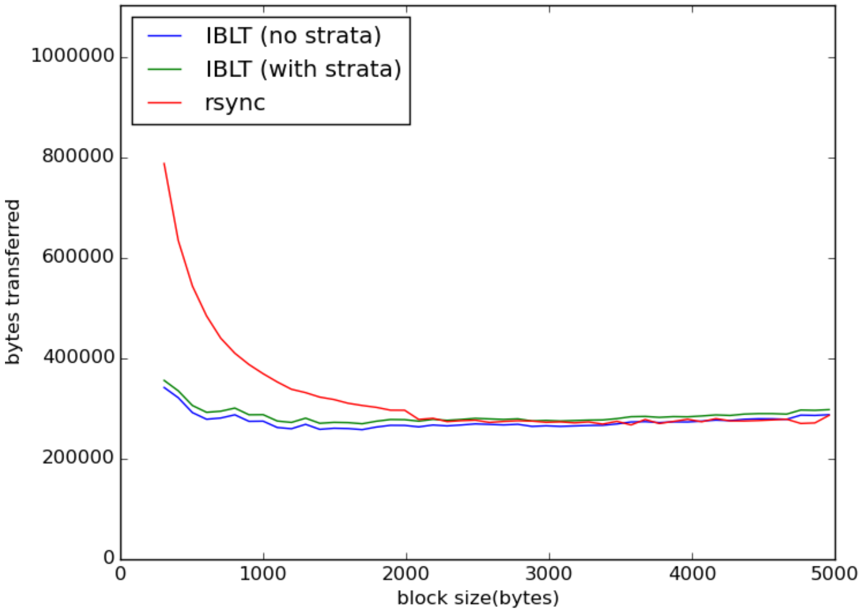


Figure 5-11: PredictionIO: Cost of Updating Source Code from Version 0.8.0 to 0.8.1



As we can see in Figure 5-10, the overhead of the Strata Estimator actually plays a significant role in the total transmission cost of IBLTsync when files are sufficiently small and sufficiently similar. While IBLTsync without the Strata Estimator always transfers fewer bytes than rsync, IBLTsync with the Strata Estimator only outperforms for blocks of size 300 bytes or less. Since the source code repo for PredictionIO is significantly larger, as seen in Figure 5-11, the overhead of the Strata Estimator is negligible. In this case, IBLTsync transfers fewer bytes than rsync for block sizes up to 120.

Chapter 6

Gossip Algorithms

Previous chapters focused on how Invertible Bloom Lookup Tables could be used for file synchronization. In this chapter, we consider an entirely different application of IBLTs, namely to gossip algorithms, which are message-spreading algorithms useful in large networks. As we will explain in more detail in Section 6.1, traditional gossip algorithms usually involve sending unaltered bits representing the messages to be transferred. We will show how multi-party IBLTs (described in more depth in Section 2.2.4) can be transmitted in place of those raw messages, transforming a traditional gossip protocol into something more similar to a set reconciliation protocol.

The benefit from using IBLTs arises when multiple messages, all from different parties, need to be propagated. While a traditional gossip protocol would require each party to keep track of multiple messages at one time and would need those parties to know which neighbors already received which messages, IBLTs allow multiple messages to be merged into one structure, obviating the need to keep track of the additional neighbor-message information. Furthermore, transferring an IBLT from one party to another and merging the two is a simple process, since IBLTs can be efficiently summed together.

In Section 6.1, we go over the definition of gossip algorithms and describe basic gossip algorithms used for message dissemination in large networks. In Section 6.2, we explain how a modification of the standard IBLT reconciliation algorithm can enable multi-party IBLTs to be adapted for gossip algorithms, and in Section 6.3, we provide experimental results verifying the theoretical performance bounds found by Mitzenmacher and Pagh in [31].

6.1 Background

Gossip algorithms are a class of networking protocols that are especially useful in settings where the network might be unstable, extremely large, or constantly changing, such as peer-to-peer networks, sensor networks, and social networks [36]. Algorithms in such settings need to be 1) very simple, so that each node, which has limited computing power, can perform the algorithm, 2) distributed, as there is no centralized control, 3) robust, as there may be network failures, and 4) efficient, as bandwidth is limited.

We consider a network of n nodes as a graph $G = (V, E)$, where V denotes the set of n vertices (we label them n_i, and call them nodes) and E denotes the set of undirected edges along which a pair of nodes can communicate. Thus, $e = (n_i, n_j) \in E$ if node n_i can communicate with node n_j. Denote d_i as the degree of node n_i. As explained in [36], the operation happening at any node n_i must satisfy the following properties to be part of a Gossip Algorithm:

1. The algorithm can only use information obtained from its neighbors (defined as the set of vertices V' such that if $n_j \in V'$, then $(n_i, n_j) \in E$).

2. The algorithm performs $O(d_i \log n)$ units of computation per time step.

3. The algorithm does not require synchronization between n_i and its neighbors.

4. The eventual outcome of the algorithm is resistant to reasonable changes (not involving complete network partitions, for instance) in the neighborhood of n_i.

6.1.1 Protocols

Depending on whether a single message or multiple messages need to be disseminated, the nature of the gossip protocol can be significantly different.

Single-Message Dissemination

In single-message dissemination, one node starts out with a message that it seeks to propagate to all other nodes as quickly as possible. We assume time happens in discrete steps. A common gossip algorithm used in this situation is the so-called randomized "rumor mongering" approach. At each time t, each node n_i contacts one of its neighbors at random. If either n_i or the neighbor it contacted had the message, then at the end of the time step, both will have the message. Note that each node can only contact one other node during each time step, but can be contacted by multiple nodes during that same time step. The protocol as described is a *push-pull* protocol. If at each time step each node that has a message sends its message to a random neighbor (but never asks for a message), it is called a *push* protocol. If instead, at each time step each node that does not have a message asks a random neighbor for its message (but never preemptively pushes a message), then it is called a *pull* protocol. The *push-pull* protocol is the natural combination of the two.

As shown in [36], for any $\epsilon > 0$, the message is propagated to all nodes in $O\left(\frac{\log(n) + \epsilon^{-1}}{\Phi}\right)$ rounds with probability $1 - \epsilon$, where Φ is the conductance of the graph, and n is the number of nodes in the graph.

The conductance of a graph is defined to be

$$\min_{S \subset V : |S| \leq n/2} \frac{\sum\limits_{n_i \in S, n_j \in S^c} a_{ij}}{a(S)}$$

where a_{ij} denotes the (i, j)th entry of the adjacency matrix of the graph (whether there is an edge from n_i to n_j), and $a(S)$ denotes the sum of the degrees of the vertices in S.

Multi-Message Dissemination

In multi-message dissemination, every node starts out with a unique message that it seeks to propagate to all other nodes as quickly as possible. At each time step, each node once again randomly contacts one of its neighbors. If we denote S_i as the set of messages node n_i has at some time t and S_j as the set of messages node n_j has at that same time, then n_i will send n_j a random message from $S_i \setminus S_j$ (if one such message exists) and n_j will send n_i a random message from $S_j \setminus S_i$. Note that this requires both nodes knowing the message sets of its neighbors at all times. Network coding, and in particular Random Linear Coding [16], provides a method to avoid this knowledge requirement by having a node send a linear combination of all the messages it holds at the time. In the section below, we consider a similar approach to Random Linear Coding that uses Invertible Bloom Lookup Tables.

6.2 Gossip Algorithms with Invertible Bloom Lookup Tables

In this section, we consider what happens if each party has multiple messages that it wants to disseminate to all other parties. We can consider this as a natural generalization of the multi-message dissemination case.

Instead of each party holding individual messages that it has received from neighbors, as in the previous section, we let each party hold an IBLT containing its set of messages. Instead of sending individual messages, each party sends an IBLT. More specifically, we have n parties, $A_1, \ldots, A_n$ with sets $S_1, \ldots, S_n$ and corresponding IBLTs $I_1, \ldots, I_n$. Each party seeks to learn all the new messages (corresponding to all the messages in $\cup_i S_i - \cap_i S_i$) as quickly as possible.

6.2.1 Motivation

As explained in Section 2.2.4, we can think of the `keysum` components of an individual multi-party IBLT as b *nit*s, where b is the length of the key and n is the number of parties. In the analysis below we omit consideration of the `hashsum` and `count` fields, as they are not essential for understanding our argument.

Let us consider what happens if we increase n to p for some prime $p > n$, so that the `keysum` now comprises b *pit*s. First, we will see why the method used in Section 2.2.4, where Party i was able to retrieve all keys in the set difference by computing $I = I_1 + I_2 + \ldots + I_n$, no longer works. If we are using *pit*-wise representations instead of *nit*-wise representations, then since each key in I is only added maximally $n < p$ times, none of the keys in $\cap_i S_i$ will be zeroed out in I. Since Party i knows its set of keys S_i, it also knows I_i. Thus, we can remedy this by adding I_i $(p-n)$ additional times to I, so that $I^* = (I + (p-n)I_i)$. $(p-n)I_i$ is shorthand for $(p-n)$ copies of I_i summed together. Note that we can compute this efficiently by just taking I_i and multiplying every field in every bucket by $p-n$ and then reducing mod p. In I^*, every key in $\cap_i S_i$ will be added p times, as desired.

The setup discussed in the previous paragraph corresponded to a complete graph on n nodes, where every party was connected to every other party and thus could get $I = I_1 + I_2 + \ldots + I_n$ (since it could ask each of the other parties individually for its IBLT). Let us now consider what happens in a connected but not necessarily complete graph, where each party can only communicate with a few other parties, and thus is not guaranteed to know $I = I_1 + I_2 + \ldots + I_n$. If a party knew some linear combination instead, say $I' = \alpha_1 I_1 + \alpha_2 I_2 + \ldots + \alpha_n I_n$, would it be able to retrieve the keys in the set difference?

Let us consider how many times a key k in $\cap_i S_i$ appears in I'. It appears once in each of $I_1, I_2, \ldots, I_n$, and hence $\alpha' = \alpha_1 + \alpha_2 + \ldots + \alpha_n$ times in I'. We want k to appear $0 \pmod{p}$ times, so that it cancels out in I'. As in the previous paragraph, if Party i were to add I_i $(p - \alpha')$ times to I' to create IBLT $I^* = (p - \alpha')I_i + I'$, then the multiplicity of k in I' would be $0 \pmod{p}$, as desired. Party i could then perform a `listing` operation on I' to retrieve all the keys in the set difference. Note that we would also have to have that $\alpha_1, \alpha_2, \ldots, \alpha_n$ are all non-zero, or else we would not have one or more of the IBLTs, meaning that we would not be able to retrieve all the keys. Based on this insight, we can create a gossip protocol with IBLTs.

6.2.2 Protocol

The gossip protocol with IBLTs is as follows. Each Party j stores I_j and one linear combination of IBLTs at any point in time. More specifically, after round l, Party j

stores

$$\sum_{i\in\{1\dots n\}} \alpha_{ijl} I_i$$

where α_{ijl} denotes the coefficient of IBLT I_i in the linear combination of IBLTs that Party j has during round l. Party j also stores the sum of the coefficients, $\alpha_{jl} = \sum_{i\in\{1\dots n\}} \alpha_{ijl}$. To send a message, Party j chooses a random coefficient $c_{jl} \neq 0 \pmod p$ and sends

$$c_{jl} \sum_{i\in\{1\dots n\}} \alpha_{ijl} I_i$$

along with the corresponding coefficient sum $c_{jl}\alpha_{jl} \pmod p$. Note that we need to transmit the coefficient sum so that we can compute $(p - \sum_{i\in\{1\dots n\}} \alpha_{ijL}) I_j$, as explained before. The receiving party just adds the received message to its current message and adds the received coefficient sum to its current coefficient sum. Using this approach, assuming that we choose p to be sufficiently large, the probability of ever accidentally zeroing out a coefficient α_{ijl} is negligible (for a formal analysis, see [31]).

Mitzenmacher and Pagh [31] prove bounds on the time to completion and failure probability of the above approach.

Theorem 3. *Set reconciliation on a graph of n vertices with n parties having sets to reconcile can be accomplished in $O(\log(n)/\Phi)$ time, where Φ is the conductance of the graph, using the randomized push-pull protocol with messages of size $O(|\cup_i S_i - \cap_i S_i|)$ with success probability $1 - O(t^{-k+2} + nt/q + ntL/p + n^2L/p + n^{-\beta})$ for any constant $\beta > 0$, where t is the* `listing` *success threshold, k is the number of hash functions in the IBLT, q is the number of possible* `hashsum` *values, and L is the number of rounds.*

As Theorem 3 states, the failure probability goes as $1/p$, so we can see the rationale for choosing a p significantly greater than n. In the next section, we see how this gossip protocol using multi-party IBLTs holds up in practice.

6.3 Experiments

In this section, we evaluate two main things. First, we measure the amount of time it takes for each party to receive some form of communication (direct or indirect) from every other party using the randomized protocol discussed in Section 6.2.2. This is explained more precisely in Section 6.3.1. Second, we measure the message retrieval success rate once that point is reached.

We use the Erdős-Rényi model for random graphs. In the $G(n, r)$ Erdős-Rényi model, there are n vertices, and each edge is independently present with probability r. It is well-known that if $r > (1+\epsilon)\frac{\ln n}{n}$ for $\epsilon > 0$, the graph is connected with high probability. For our experiments, we choose $\epsilon = 1$, so that the probability of any edge existing is $2\frac{\ln n}{n}$. We ensure that every graph generated this way is connected, as otherwise it is impossible for all parties to receive every message.

Each vertex in our graph is one party that contains its own unique message. We use the push-pull protocol as explained in Section 6.1.1. We choose each party's IBLT to have $2n$ buckets (there are n different messages that the IBLT must hold at the end), so that the probability of a failed `listing` operation is negligible.

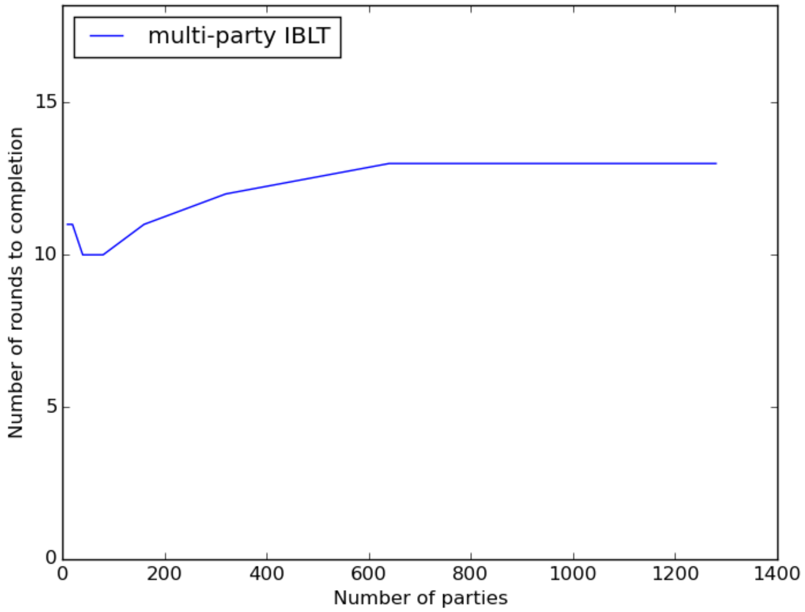
6.3.1 Dissemination Completion Time

We measure the number of rounds it takes for each party to receive some form of communication (whether direct or indirect) from all other parties using the randomized push-pull protocol. More precisely, with n parties, each party has an n-bit bit vector (we will call this r_i for Party i), with bit j of r_i set if Party i has received communication from Party j. Initially bit i is set for Party i, since it has "received" communication from itself. When Party i is contacted by Party j, Party i updates r_i so that $r_i = r_i \vee r_j$ (the OR of its bit vector and Party j's). We update like the above since after receiving communication from Party j, Party i has been transitively contacted by all the parties that have contacted Party j previously. We measure the round t such that r_i is all 1's for every Party i. At that point, it is theoretically possible for each party to retrieve all the messages from all the parties (we can imagine that at each communication, Party j transfers all of the messages it holds at that time to Party i, and Party i just takes the union of Party j's set of messages and its own).

According to Theorem 3, the number of rounds for that to happen is $O(\log(n)/\Phi)$ with high probability, where n is the number of parties and Φ is the conductance of the graph.

Figure 6-1 shows the 99.9th percentile completion time of message dissemination for graphs ranging from 10 to 1280 parties. We based this off of 1000 trials for each graph size. As we can see, the time to completion for message dissemination increases very slowly as the number of parties increases, as Theorem 3 suggests.

Figure 6-1: 99.9th Percentile for Number of Rounds Required for Each Party to Receive Linear Combination of All Messages



6.3.2 Listing Success Rate

In this section, we first run the randomized push-pull model until each party receives some form of communication from every other party, exactly as in the previous section. At that point, we measure the success rate of `listing` for each party. By waiting until each party receives some form of communication from all other parties, we ensure that `listing` will not fail because we have run the protocol for too few rounds. By our choice of the IBLT size, we also ensure that `listing` will only have negligible failure probability from having too few buckets. Thus, we have narrowed our failure probability to come from any failures resulting from the zeroing out of coefficients of the linear combination of IBLTs in the randomized push-pull protocol. This failure probability corresponds to the term $O(ntL/p + n^2L/p)$ from Theorem 3. Our L value (corresponding to the number of rounds we run the protocol) has already been fixed. Thus, the two parameters we can change are n, the number of parties, and p, the prime we choose to mod each coefficient by. We consider three settings of the parameter p, 14653, 860117, and 1000000007, and vary the number of parties from 10 to 1280 for each.

We expect that the error rate should decrease pretty much linearly with the size of the prime p (since we designed the experiment so that the failure probability should mainly come from the $O(ntL/p + n^2L/p)$ term in Theorem 3). Indeed, we find from our experiments that the probability of missing at least one message decreases approximately linearly with p. Increasing p from 14653 to 860117 (≈ 59× increase) decreases the error probability by ≈ 55×. Increasing p from 860117 to 1000000007 (≈ 1163× increase) decreases the error probability by ≈ 1900×.

Table 6.1: Success Rate of Listing Keys (Averaged over 1000 trials) with $p = 14653$

# Parties	% Retrieving All Msgs	% Missing 1 Msg	% Missing >1 Msg	# Retrieving All Msgs	# Missing 1 Msg	# Missing >1 Msg
10	99.93%	0.07%	0.00%	9993	7	0
20	99.86%	0.14%	0.00%	19972	28	0
40	99.79%	0.22%	0.00%	39914	86	0
80	99.50%	0.50%	0.00%	79601	398	1
160	98.92%	1.07%	0.01%	158278	1713	9
320	97.82%	2.16%	0.02%	313024	6904	72
640	95.77%	4.14%	0.09%	612920	26523	557
1280	91.68%	7.96%	0.35%	1173545	101921	4534

Table 6.1 shows the success rate of listing keys for a chosen p of 14653, averaged

over 1000 trials, for 10 to 1280 parties. The *% Retrieving All Messages* column shows the percentage of parties that are able to retrieve all of the messages from all the parties. The other two columns are defined analogously. Even for 10 parties, in 7 out of 1000 trials, one of the parties was unable to retrieve all the messages. However, for $n \leq 40$ parties, each party never missed more than one message. As the number of parties increased, the percentage of parties missing a message went up accordingly. With a graph of 1280 parties, the percentage of parties missing one message was $\approx 8\%$, and the percentage of parties missing more than one message was 0.35%. In the case of $n = 1280$, the chosen p value of 14653 is quite small (only a factor of 12 greater than the number of parties), so it makes sense that the missed message rate would be quite high. Also, note that the percentage of parties unable to retrieve all messages increases approximately linearly with the number of parties, as Theorem 3 also suggests (the ntL/p term in the error probability dominates here).

Table 6.2: Success Rate of Listing Keys (Averaged over 1000 trials) with $p = 860117$

# Parties	% Retrieving All Msgs	% Missing 1 Msg	% Missing >1 Msg	# Retrieving All Msgs	# Missing 1 Msg	# Missing >1 Msg
10	100.00%	0.00%	0.00%	10000	0	0
20	99.99%	0.01%	0.00%	19998	2	0
40	99.99%	0.01%	0.00%	39995	5	0
80	99.99%	0.01%	0.00%	79993	7	0
160	99.98%	0.02%	0.00%	159971	29	0
320	99.96%	0.04%	0.00%	319870	130	0
640	99.92%	0.08%	0.00%	639505	495	0
1280	99.85%	0.15%	0.00%	1278055	1942	3

Table 6.2 shows the success rate of listing keys for a chosen p of 860117, averaged over 1000 trials. With this p value, we are able to retrieve all messages for $n = 10$ in all of our trials. For $n = 80$, in 7 out of 1000 trials, one of the parties was unable to retrieve all the messages. Recall that for $p = 14653$, for that same error rate, we could only have 10 parties. For all of our tests up to and including $n = 640$, we never had a party that missed more than one message. For $n = 640$, 99.92% of the parties are able to retrieve all their messages, and for $n = 1280$, 99.85% are able to

do similarly. Once again, we see that the percentage of parties unable to retrieve all messages increases approximately linearly with the number of parties.

Table 6.3: Success Rate of Listing Keys (Averaged over 1000 trials) with $p = 1000000007$

# Parties	% Retrieving All Msgs	% Missing 1 Msg	% Missing >1 Msg	# Retrieving All Msgs	# Missing 1 Msg	# Missing >1 Msg
10	100.00%	0.00%	0.00%	10000	0	0
20	100.00%	0.00%	0.00%	20000	0	0
40	100.00%	0.00%	0.00%	40000	0	0
80	100.00%	0.00%	0.00%	80000	0	0
160	100.00%	0.00%	0.00%	160000	0	0
320	100.00%	0.00%	0.00%	320000	0	0
640	100.00%	0.00%	0.00%	640000	0	0
1280	100.00%	0.00%	0.00%	1279999	1	0

Table 6.3 shows the success rate of listing keys for a chosen p of 1000000007, averaged over 1000 trials. With this p value, we are able to retrieve all messages for n up to 640 in all of our trials. For $n = 1280$, we only miss one key from one party in one trial. Note that this value of p is not unreasonable. If we store each *pit* using a 32-bit integer, then we can have p values up to $2^{32} - 1$, which is approximately four times larger than the p chosen above.

As all three tables tables suggest, if we choose p to be sufficiently large, then the vast majority of messages are recovered. If p is small, then we still recover the majority of messages, but the rate of missed messages is much higher. The benefit of a smaller value of p is that each IBLT will be smaller. This is the case since each *pit* in `keysum` and `hashsum` needs to be maximally $p - 1$, which takes $\log p$ bits, so the size of each IBLT scales as $\log(p)$. In a more bandwidth constrained environment, we might be willing to accept a higher missed message rate so that the transmitted IBLTs are smaller. If maximally one message is missing per party, as is the case for larger values of p, then we could transmit an additional checksum that is the exclusive-or of all the messages, and use that to recover the missing message.

Chapter 7

Conclusion

In this thesis, we set out to better understand the many interesting properties of Invertible Bloom Lookup Tables and the practicality of their use for various applications. To understand Invertible Bloom Lookup Tables in the first place, we had to explain the set reconciliation problem, which involves efficiently synchronizing sets of objects. During that discussion, besides showing how Invertible Bloom Lookup Tables could be used for set reconciliation, we also explained how a few other methods, including Bloom filters, Approximate Reconciliation Trees, and Characteristic Polynomial Interpolation, could be used to solve the set reconciliation problem. Many of those techniques only work when considering set reconciliation with two parties – we also discussed multi-party IBLTs, an extension to Invertible Bloom Lookup Tables that is able to handle an arbitrary number of parties desiring to reconcile their sets.

Browsing the literature, we saw that file synchronization was one such application for set reconciliation. As we found file synchronization to be an interesting and relevant problem, we then sought to see how IBLTs could be harnessed for file synchronization. Reading up on various file synchronization techniques, including `rsync`, probably the most commonly used file synchronization protocol today, and string reconciliation using puzzles, an approach that involves set reconciliation for file synchronization, we were able to develop a new file synchronization protocol that uses IBLTs, which we called `IBLTsync`.

To evaluate whether `IBLTsync` had any practical relevance, we implemented our file synchronization protocol in C++ and tested it on a variety of workloads, comparing it to rsync and a simple file transfer protocol. We found that IBLTsync indeed had transmission benefits, transferring fewer and fewer bytes relative to rsync and the simple file transfer protocol as the two files become more and more similar.

As we explained, the above approach only works when trying to synchronize a single file between a local and remote machine. Although we did not provide an implementation, we also proposed an extension to our file synchronization protocol that allowed for synchronization between local and remote machine directories.

As another application for Invertible Bloom Lookup Tables, we considered gossip protocols, communication protocols that are especially effective in large and changing networks. Multi-party IBLTs fit particularly well into such a framework. In addition to explaining how multi-party IBLTs can easily be used for gossip protocols, thus converting gossip algorithms into reconciliation algorithms, we provided an implementation and empirical evaluation of IBLTs in such settings, corroborating many of the theoretical findings from [31], and showing that multi-party IBLTs can be useful in practice.

7.1 Future Work

As seen in our experiments from Chapter 5, the amount of data transmitted during file synchronization depends greatly on the chosen block size. It would be interesting to see if we could come up with efficient and accurate methods to estimate this optimal block size before initiating the file transfer, as that would significantly improve the performance of many file synchronization protocols.

In our evaluation of multi-party IBLTs for gossip protocols, we only considered the Erdős-Rényi model of random graphs. It might be interesting to consider other network topologies, such as the ring model, where every node is connected to a neighbor to its left and a neighbor to its right, or a social network model, where there are clusters of nodes that are highly connected intra-cluster but only sparsely connected inter-cluster.

Bibliography

[1] www.dropbox.com.

[2] cassandra.apache.org.

[3] gist.github.com/gavinandresen/e20c3b5a1d4b97f79ac2.

[4] developers.google.com/protocol-buffers.

[5] https://github.com/sharelatex/sharelatex.

[6] https://github.com/PredictionIO/PredictionIO.

[7] Sachin Agarwal, Vikas Chauhan, and Ari Trachtenberg. Bandwidth efficient string reconciliation using puzzles. *IEEE Transactions on Parallel and Distributed Systems*, 17, 2006.

[8] Nikolaj Bjorner, Andreas Blass, and Yuri Gurevich. Content-dependent chunking for differential compression, the local maximum approach. Technical Report MSR-TR-2009-74, Microsoft Research, August 2007.

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

[10] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[11] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 327–336, New York, NY, USA, 1998. ACM.

[12] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Fast approximate reconciliation of set differences. In *BU Computer Science TR*, pages 2002–19, 2002.

[13] John W. Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed content delivery across adaptive overlay networks. In *In Proceedings of ACM SIGCOMM*, pages 47–60, 2002.

[14] Graham Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. *IEEE/ACM Trans. Netw.*, 13(6):1219–1232, December 2005.

[15] Graham Cormode, Mike Paterson, Süleyman Cenk Sahinalp, and Uzi Vishkin. Communication complexity of document exchange. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, pages 197–206, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[16] Supratim Deb, Muriel Médard, and Clifford Choute. Algebraic gossip: A network coding approach to optimal multiple rumor mongering. *IEEE/ACM Trans. Netw.*, 14(SI):2486–2507, June 2006.

[17] David Eppstein, Michael T. Goodrich, and Pierre Baldi. Privacy-enhanced methods for comparing compressed DNA sequences. *CoRR*, abs/1107.3593, 2011.

[18] David Eppstein, Michael T. Goodrich, Frank Uyeda, and George Varghese. What's the difference?: Efficient set reconciliation without prior context. *SIGCOMM Comput. Commun. Rev.*, 41(4):218–229, August 2011.

[19] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 501–, Washington, DC, USA, 1999. IEEE Computer Society.

[20] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, September 1985.

[21] Michael T. Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. *CoRR*, abs/1101.2245, 2011.

[22] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[23] Jiayang Jiang, Michael Mitzenmacher, and Justin Thaler. Parallel peeling algorithms. *CoRR*, abs/1302.7014, 2013.

[24] Richard M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, March 1987.

[25] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[26] RichardJ. Lipton. Efficient checking of computations. In Christian Choffrut and Thomas Lengauer, editors, *STACS 90*, volume 415 of *Lecture Notes in Computer Science*, pages 207–215. Springer Berlin Heidelberg, 1990.

[27] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 2–2, Berkeley, CA, USA, 1994. USENIX Association.

[28] J. J. Metzner. A parity structure for large remotely located replicated data files. *IEEE Trans. Comput.*, 32(8):727–730, August 1983.

[29] John J. Metzner. Efficient replicated remote file comparison. *IEEE Trans. Comput.*, 40(5):651–660, May 1991.

[30] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. In *in International Symposium on Information Theory*, page 232, 2000.

[31] Michael Mitzenmacher and Rasmus Pagh. Simple multi-party set reconciliation. *CoRR*, abs/1311.2037, 2013.

[32] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, New York, NY, USA, 2005.

[33] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5):174–187, October 2001.

[34] Norman Ramsey and Elod Csirmaz. An algebraic approach to file synchronization. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 175–185, New York, NY, USA, 2001. ACM.

[35] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[36] Devavrat Shah. *Gossip Algorithms.* Foundations and Trends in Networking, Now Publishers Inc, 2009.

[37] David Starobinski, Ari Trachtenberg, and Sachin Agarwal. Efficient pda synchronization. *IEEE Transactions on Mobile Computing*, 2(1):40–51, January 2003.

[38] Torsten Suel, Patrick Noel, and Dimitre Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 153–, Washington, DC, USA, 2004. IEEE Computer Society.

[39] Andrew Tridgell. Efficient algorithms for sorting and synchronization, 1999.

[40] Hao Yan, Utku Irmak, and Torsten Suel. Algorithms for low-latency remote file synchronization. In *INFOCOM 2008. 27th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 13-18 April 2008, Phoenix, AZ, USA*, pages 156–160, 2008.

[41] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.