Fast Approximate Reconciliation of Set Differences

John Byers* Jeffrey Considine* Michael Mitzenmacher†

Abstract

We present new, simple, efficient data structures for *approximate reconciliation of set differences*, a useful standalone primitive for peer-to-peer networks and a natural subroutine in methods for exact reconciliation. In the approximate reconciliation problem, peers A and B respectively have subsets of elements S_A and S_B of a large universe U. Peer A wishes to send a short message M to peer B with the goal that B should use M to determine as many elements in the set $S_B - S_A$ as possible. To avoid the expense of round trip communication times, we focus on the situation where a single message M is sent.

We motivate the performance tradeoffs between message size, accuracy and computation time for this problem with a straightforward approach using Bloom filters. We then introduce *approximation reconciliation trees*, a more computationally efficient solution that combines techniques from Patricia tries, Merkle trees, and Bloom filters. We present an analysis of approximation reconciliation trees and provide experimental results comparing the various methods proposed for approximate reconciliation.

1 Introduction

The problem of finding set differences is a fundamental problem in distributed computing, arising in protocols for gossiping, synchronization, and replication. Recent work such as [8, 10, 9] solves the problem of finding exact set differences between two parties with near optimal communication complexity and presents various tradeoffs between the number of rounds and the computational complexity. In contrast to previous work, we consider finding *approximate* set differences, where the goal is to determine a large fraction of the set difference between the parties. We obtain solutions that allow greatly reduced computation over exact methods and utilize only a single round of communication.

*Supported in part by NSF grants ANI-0093296 and ANI-9986397. E-mail: {byers,jconsidi}@cs.bu.edu.

†Supported in part by NSF grants CCR-9983832, CCR-0118701, CCR-0121154, and an Alfred P. Sloan Research Fellowship. E-mail: michaelm@eecs.harvard.edu.

Approximate reconciliation is motivated by the potential expense (in computation or communication) of exact reconciliation methods. Fast approximate reconciliation can also be used as a prelude to a more expensive exact reconciliation scheme, for applications such as database synchronization. Here, the substantial reduction in the size of the set difference afforded by fast approximate reconciliation speeds up subsequent exact reconciliation of a small set difference.

The approximate reconciliation problem is also strongly motivated by peer-to-peer applications. For example, if two peers with large song libraries wish to trade music, it may suffice for them to learn a large fraction of the set difference in order to begin useful trading. As another example, in [2], approximate reconciliation is used as a subroutine for informed content delivery of large data files protected with erasure-resilient codes.[1] In this context, approximate reconciliation allows peers downloading a piece of popular content to quickly identify and send useful encoding packets to other peers downloading that same content in parallel. With content protected by an erasure-resilient encoding, a peer need only receive a sufficiently large subset of all encoding packets to recover the content in its entirety, so approximate reconciliation is all that is needed for this application. Moreover, the reconciliation must be performed in real-time, and in parallel with a number of other processes, thus the speed advantage of one-round approximate reconciliation is compelling.

Previous to our efforts, work on set reconciliation generally focused on the exact case. Reconciliation can clearly be done by simple enumeration of set elements or, to save space, hashes of set elements. Exact reconciliation with near optimal communication complexity was considered in [8, 10, 9]. Their general approach involves each peer evaluating the characteristic polynomial of their set at several points and using this information to compute the ratio of the two characteristic polynomials and hence their set differences. This necessitates interpolation and factorization of rational polynomials, which are computationally expensive tasks. Furthermore, their approach either requires knowing an upper bound on the number of differences $D = |S_A - S_B| + |S_B - S_A|$, or requires multiple rounds of communication. These additional communication rounds are used either to improve the estimated upper bound or to apply divide and conquer techniques. Their best protocol to date runs in expected linear time but requires $O(\log D)$ rounds of communication. Some of the basic data structures we use (Patricia tries and Merkle trees) were also employed in different ways in some of the protocols by these authors.

In contrast to previous work, our work focuses on one round protocols reconciling differences between very large sets. For example, in the informed content delivery application, sets of encoding symbols for 1GB files may have sizes in the millions, and peers may have sets with hundreds of thousands of differences. In such a scenario, it is essential to reconcile quickly to allow the more important transfer of content to begin. On the other hand, minimizing the communication complexity in bits may not be necessary as long as the cost for reconciliation is a tiny fraction of the

[1] We note that we first provided a brief overview of the data structures we develop in this paper in [2]. This paper provides a full description, including all relevant theoretical details, multiple variations, and detailed experimental results.

bandwidth used for content transfer.

Another important aspect of our work is to carefully separate preprocessing costs from the costs of reconciliation, since the latter must always be incurred in real-time. This makes sense in a peer-to-peer setting, where a peer B may be requested to determine set differences with several other peers, and hence the time to construct the set representation can be amortized over several pairwise communications. Therefore, while we account for all costs, we focus on minimizing the work done once peer A has sent its message to B.

Our work draws upon several elegant, classical data structures, including Merkle trees [7], Bloom filters [1], and Patricia tries [6]. We describe each of these data structures below, beginning with Bloom filters, which by themselves provide a reasonable starting point for approximate reconciliation.

2 Bloom Filters for Approximate Reconciliation

One simple mechanism for one-round approximate reconciliation is to use a Bloom filter [1]. This solution is surprisingly effective, particularly when the number of differences is a large fraction of the set size. Our goal in designing subsequent data structures is to improve upon this solution for other, more challenging cases.

We review the Bloom filter data structure, following the framework of [4]. A Bloom filter is used to represent a set $S = \{s_1, s_2, \ldots, s_n\}$ of n elements from a universe U of size u, and consists of an array of m bits, initially all set to 0. A Bloom filter uses k independent random hash functions $h_1, \ldots, h_k$ with range $\{0, \ldots, m-1\}$. For each element $s \in S$, the bits $h_i(s)$ are set to 1 for $1 \leq i \leq k$. To check if an element x is in S, we check whether all $h_i(x)$ are set to 1. If not, then clearly x is not a member of S. If all $h_i(x)$ are set to 1, we assume that x is in S, although we are wrong with some probability. Hence a Bloom filter may yield a *false positive*, when it suggests that an element x is in S even though it is not. The probability of a false positive f depends on the number of bits used per item m/n and the number of hash functions k according to the following equation:

$$f \approx (1 - \mathrm{e}^{-km/n})^k.$$

(We note that this is a highly accurate approximation and we treat it as an equality henceforth.) This false positive rate is minimized by picking $k = (\ln 2)(m/n)$ which results in $f = (1/2)^{(\ln 2)(m/n)}$.

For an approximate reconciliation solution, peer A sends a Bloom filter F_A of S_A; peer B then simply checks for each element of S_B in F_A. When a false positive occurs, peer B will assume that peer A has an element that it does not have, and so peer B fails to find this element of $S_B - S_A$. However, the Bloom filter does not cause peer B to ever mistakenly declare an element to be in $S_B - S_A$ when it is not.

To evaluate this data structure, we now specify the relevant performance criteria for approximate reconciliation.

- **Number of communication rounds.**

- **Message size:** the number of bits sent by A to B.
- **Construction time:** for A, the construction time is the time for A to compute its message; for B, the construction time is the time to produce an appropriate representation of S_B.
- **Reconciliation time:** the time for B to compute the approximation to $S_B - S_A$ given the message from A and an appropriate representation of S_B.
- **Accuracy:** the probability that a given element in $S_B - S_A$ is identified by B. Of secondary concern is the correlation between correct identification of elements in $S_B - S_A$.

With Bloom filters, the message size can be kept small while still achieving high accuracy. For example, using just four bits per element of S_A, i.e. a message of length $4|S_A|$, and three hash functions yields an accuracy of 85.3%; using eight bits per element and five hash functions yields an accuracy of 97.8%. Further improvements can be had by using the recently introduced compressed Bloom filter, which reduces the number of bits transmitted between peers at the cost of using more bits to store the Bloom filter at the end-systems and requiring compression and decompression at the peers [11]. With a constant number of hash functions and assuming hashes and array accesses are constant time operations, A's construction time is $O(|S_A|)$ to set up the Bloom filter, and B's reconciliation time is $O(|S_B|)$ to find the set difference. B has no construction time, although B could precompute hashes of its set elements to shift reconciliation time to construction time.

This may be summarized with the following (obvious) theorem.

Theorem 1 *There exists a one-round protocol with message size $O(|S_A|)$, construction time $O(|S_A|)$ for A, reconciliation time $O(|S_B|)$, and constant accuracy.*

The requirement for $O(|S_A|)$ construction time and $O(|S_A|)$ message size may seem excessive for large $|S_A|$. There are several possibilities for scaling this approach up to larger set sizes. For example, for large $|S_A|$ or $|S_B|$, peer A can create a Bloom filter only for elements of S_A that are equal to β modulo γ for some appropriate β and γ. Peer B can then only use the filter to determine elements in $S_B - S_A$ equal to β modulo γ (still a relatively large set of elements). The Bloom filter approach can then be pipelined by incrementally providing additional filters for differing values of β as needed. This pipelining approach can similarly be used in many other schemes.

Another unavoidable drawback of Bloom filters is the large reconciliation time: even when the set difference $S_B - S_A$ is small, every element of S_B must be tested against the filter F_A. As described in the introduction, minimizing the reconciliation time can be crucial in applications where reconciliation must be performed in real-time. Our new data structures avoid this problem.

3 Approximate Reconciliation Trees

In this section, we develop a data structure which we call an approximate reconciliation tree that can reconcile set differences in time sub-linear in $|S_B|$, at the expense of extra logarithmic terms in the construction times. In practice, the extra expenses appear relatively minor, as we detail in Section 4. The key idea is for A to send a searchable structure as a message, so that not every element of S_B needs to be tested for membership in $S_B - S_A$. Specifically, the protocol using the final version of this data structure we develop will the following theorem.

Theorem 2 *There exists a one-round protocol with message size $O(|S_A|)$, construction time $O(|S_A| \log |S_A|)$ for A, construction time $O(|S_B| \log |S_B|)$ for B, reconciliation time $O(|S_B - S_A| \log |S_B|)$ (with high probability), and constant accuracy.*

(This theorem assumes that operations for handling $O(\log |S_A|)$ or $O(\log |S_B|)$ (contiguous) bits can be done in constant time, e.g. for comparing hashes of $O(\log |S_A|)$ bits.)

We describe the data structures upon which approximate reconciliation trees are based (in addition to Bloom filters), in Section 3.1. We provide and analyze the basic construction of approximate reconciliation trees in Section 3.2. Improvements over this basic construction leading to asymptotically better performance are discussed in Section 3.3 and the final construction is analyzed in Section 3.4.

3.1 Prerequisite Data Structures

In addition to Bloom filters, approximate reconciliation trees are based upon Patricia tries [6] and Merkle trees [7]. Patricia tries are used to provide structured searching based upon comparable subsets while Merkle trees are used to make comparisons of these subsets practical. Finally, Bloom filters are used to provide compact representations and avoid some complications. We describe Patricia tries and Merkle trees briefly below.

3.1.1 Patricia Tries

Each peer represents its set as a *Patricia trie* [6]. We describe the construction of a binary Patricia trie for an arbitrary subset S of a universe $U = \{0, \ldots, u-1\}$. The root (with depth 0) corresponds to the entire subset S. The children correspond to the subsets of S in each half of U; that is, the left child is $S \cap [0, u/2 - 1]$ and the right child is $S \cap [u/2, u - 1]$. The rest of the trie is similar; the jth child at depth k corresponds to the set $S \cap [(j-1) \cdot u/2^k, j \cdot u/2^k - 1]$. As described, the trie has $\Theta(u)$ nodes and depth $\Theta(\log u)$, which is unsuitable when the universe is large. However, almost all the nodes in the trie correspond to the same sets. In fact there are only $2|S| - 1$ non-trivial nodes. The trie can be collapsed by removing edges between nodes that correspond to the same set, leaving only $2|S| - 1$ nodes. The resulting trie

is referred to as a Patricia trie (or Patricia tree) in the search literature. (Patricia tries do not need to be binary, as we discuss further in Section 3.3.)

We now explain methods for identifying set discrepancies using trie representations. For simplicity, we describe a method for searching on uncollapsed tries of size $\Theta(u)$. Suppose peer A and peer B have constructed uncollapsed tries T_A and T_B of their respective subsets S_A and S_B, and peer A sends its trie to peer B. If the root of peer A matches the root of peer B, then there are no differences between the sets. Otherwise, there is a discrepancy. Peer B then recursively considers the children of the root. If $x \in S_B - S_A$, eventually peer B determines that the leaf corresponding to x in its tree is not in the tree for peer A. Hence peer B can find any $x \in S_B - S_A$. Assuming nodes in the trie can be compared in constant time, the total work for peer B to find all of $S_B - S_A$ is $O(|S_B - S_A| \log u)$, since each discrepancy may cause peer B to trace a path of depth $\log u$. Searching for discrepancies on collapsed Patricia tries is harder, because for a given node in T_B, it is no longer clear which node it maps to in T_A, unless annotations specifying the collapse operations are included with the trie. We present methods that avoid this problem subsequently.

Our search method operates on the Patricia trie T_B, and hence the running time is proportional to the depth of T_B. However, the worst-case depth of the Patricia trie may still be $\Omega(|S_B|)$. To avoid this issue with high probability, we hash each element initially before inserting it into the virtual tree, as shown in Figure 1(a,b). The range of the hash function should be at least $\text{poly}(|S_B|)$ to avoid collisions. We assume that this hash function appears random, so that for any set of values, the resulting hash values appear random. (Indeed, we make this assumption throughout this paper for all hash functions.) Hence we obtain a random Patricia trie, properties of which have been studied in the random search tree literature, in particular [5]. Specifically, the average depth of a leaf for a binary Patricia trie with n random leaf values over the interval is $\log n + O(1)$, and the variance of the depth of a leaf is constant. Moreover, the maximum depth is $\log n + \sqrt{2 \log n} + O(1)$ with high probability. Hence the distribution of the leaf depths is very closely concentrated around $\log n$.

An important consideration is that each node in the trie can represent a subset of elements, which makes exact comparison of nodes in constant time impossible. Probabilistic comparisons can be done by comparing hashes, however. This leads us to consider Merkle trees.

3.1.2 Merkle Trees

Merkle trees [7] provide a method for signing and comparing large databases while allowing fast updates and identifying differences. For our application, we can form a Merkle tree on top of our trie by associating a value with each node of the underlying tree. At the leaves, the value is obtained by applying a hash function to the element represented by the leaf. The values of internal nodes of a Merkle tree are then obtained by applying a hash function to the values of their children. Using this

construction, the hash of a node is dependent upon all of the elements in its subtree. In our case, it suffices to use a simple mixing function such as bit-wise exclusive-or to obtain the value of an internal node from the values of its children. (Also note that the hash function used at the leaves in the Merkle tree should be independent of the hash function used to organize elements in the Patricia trie to avoid correlation amongst elements in a subtree.) See Figure 2 for an example.

Merkle trees on top of uncollapsed tries give a natural set reconciliation algorithm if the tree and its hashes are sent. This algorithm is exactly that described above, however this time we compare the hash values associated with corresponding nodes in the trie. While this affords constant time comparisons, it now runs the risk of false positive matches due to hash collisions.

The problem in extending this method to the Patricia trie is the complexity associated with specifying the collapses so that corresponding nodes in the tries can be compared. We address this in the following section using Bloom filters.

3.2 Basic Construction

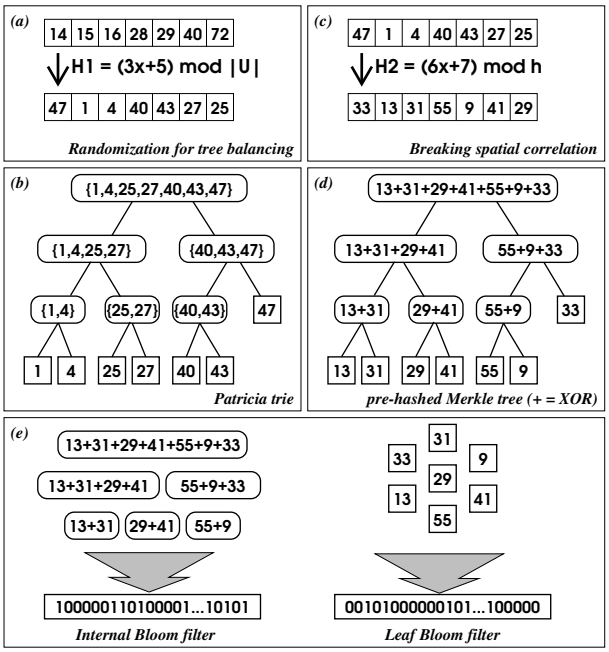
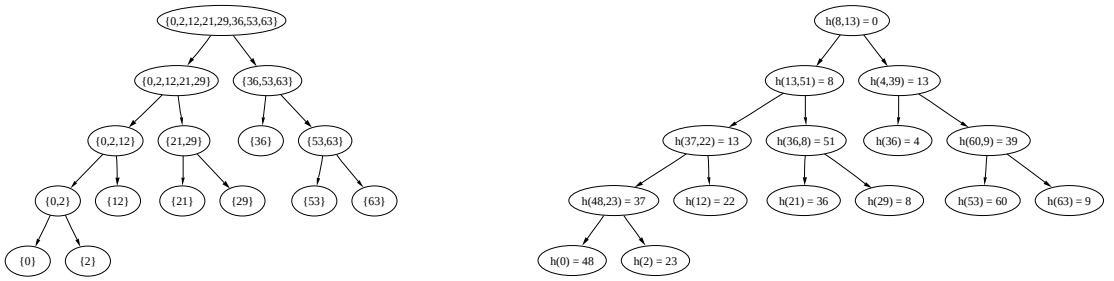


Figure 1: Example of creation and Bloom filtering of an approximate reconciliation tree. (M is $O(\text{poly } |S_A|)$; in this case, $M = |S_A|^2 = 49$, h is 64, and example permutation functions are as shown.

We now describe approximate reconciliation trees, a Bloom filtered representation of the Patricia/Merkle tree combination we have just described. As before, each peer starts by building a Patricia trie of their set along with the associated Merkle tree values. The message A then sends to B is a Bloom filter of the values from the Merkle tree. For B to perform approximate reconciliation, B uses the



(a) Patricia trie of a set

(b) Merkle tree of the same set

Figure 2: An example Merkle tree derived from the Patricia trie of a set. Input values and hash values are from the range $[0, 63]$. Note that there are hash collisions between different nodes in the tree.

same recursive algorithm previously used to traverse an *uncollapsed* trie, to traverse its *collapsed* Patricia trie T_B. Instead of performing a comparison of one trie node against another, B checks the value of a node in T_B by performing a lookup of that value in the Bloom filter provided by A. This tests whether *any* node in T_A has that value. Using a Bloom filter to summarize the Merkle tree values has the following advantages:

- It nearly eliminates the incidence of hash collisions in the Merkle trees since the approximate reconciliation tree can use a large number of bits for each Merkle tree value, summarizing the values in the Bloom filter instead of sending them.
- It avoids complications associated with collapse operations when performing comparisons across Patricia tries, as no explicit bookkeeping is present in the transmitted message.

The obvious disadvantage is that comparisons between nodes in the tries now correspond to Bloom filter lookups, which are less accurate than direct comparisons of Merkle tree values.

Let us consider the accuracy of this approach. First, because we are sending a Bloom filter of the node values, we can use a large number of bits for these values to avoid collisions ($\Theta(\log |S_B|)$ bits suffices with high probability and 64 bits covers most practical situations). We will ignore these collisions henceforth in the analysis. For B to obtain a false positive for an element x in $S_B - S_A$ at depth d in the approximation reconciliation tree for S_B, there must be a false positive for one of the d node values on the path from the root to the leaf representing x in the Bloom filter. If the false positive rate of the Bloom filter sent by A is f, the probability p_x that B identifies x as a member of $S_B - S_A$ is

$$p_x = (1 - f)^d \tag{1}$$

To achieve a constant expected accuracy, f should be at most $O(1/d)$ for most elements. Since individual elements are at depth $\log|S_B| + O(1)$ in the tree of B with high probability as mentioned earlier, the false positive rate of the Bloom filter from A should be $O(1/\log|S_B|)$. This means that A should use $\Omega(\log\log|S_B|)$ bits per element in the Bloom filter. Additionally, the number of hash functions must be $\Theta(\log\log|S_B|)$ to minimize the false positive rate (using only a constant number of hash functions c requires the number of bits per element to be $\Omega(\sqrt[c+1]{\log|S_B|})$).

This may mean that while constructing the Bloom filter representing S_A, peer A should know the approximate size of S_B; this is often the case for practical situations [2]. In practice, even if the deviation between set sizes is very large, then the difference between $\log\log|S_B|$ and $\log\log|S_A|$ will not be significant and a small constant factor "padding" the size of the Bloom filter will cover nearly all practical situations. We expect a small fixed number of hash functions will be universally chosen ahead of time.

Using the basic construction described so far, we can prove the following theorem.

Theorem 3 *There exists a one-round protocol with message size $O(|S_A|\log\log|S_B|)$, construction time $O(|S_A|\log|S_A|\log\log|S_B|)$ for A, construction time $O(|S_B|\log|S_B|)$ for B, reconciliation time $O(|S_B - S_A|\log|S_B|\log\log|S_B|)$ (with high probability), and constant accuracy.*

Proof: Peer A constructs a Bloom filter representing S_A in the approximate reconciliation tree as outlined above, using a Bloom filter with $\Theta(\log\log|S_B|)$ bits per node of the tree and $\Theta(\log\log|S_B|)$ hash functions. The message size is then $O(|S_A|\log\log|S_B|)$ and the construction time for A is $O(|S_A|\log|S_A|\log\log|S_B|)$. B can precompute the values in its approximate reconciliation tree in time $O(|S_B|\log|S_B|)$. Each check of a value in the Bloom filter takes $O(\log\log|S_B|)$ time, and there are at most $O(|S_B - S_A|\log|S_B|)$ checks with high probability (as long as the depth of the Patricia tree for S_B is $O(\log|S_B|)$). The false positive probability for the Bloom filter is $O(1/\log|S_B|)$ since there are $\Theta(\log\log|S_B|)$ bits per element and hash functions. Since each element has depth $O(\log|S_B|)$ with high probability, the expected accuracy will be constant. (The exact constant depends on the constant factors hidden in the order notation, and will be investigated experimentally in Section 4.) □

3.3 Improvements

Approximate reconciliation trees as described so far combine some of the better properties of Bloom filters and Merkle trees, namely quicker searches for small numbers of differences without the complications of managing the tree structures. Unfortunately, they inherit a common weakness in tree-based search strategies – incorrect pruning from false positives can result in large numbers of differences being overlooked. For example, if there is a false positive when checking the root of an approximate reconciliation tree, no differences will be found and the sets will be

reported to be identical. Addressing this problem required increasing the number of bits per element and the running times by non-constant factors. In this section, we discuss a series of improvements over basic approximate reconciliation trees to further ameliorate these problems. All of the improvements allow significantly better performance in practice and they collectively enable our final construction for the proof of Theorem 2.

3.3.1 Increased Branching Factor

Our first improvement is very simple: increase the (maximum) branching factor of the trees. This reduces the number of internal nodes, thereby improving the false positive probability by allowing more bits per node. It also decreases the height of the tree, reducing the number of Bloom filter tests by a constant factor. The cost is a potential increase in running time, since in searches for elements of $S_B - S_A$, all children of an internal node must be checked when a match does not occur. This improvement does not change the asymptotic results, but only constant factors. However, larger branching factors are an important part of a real implementation. These benefits are illustrated in our experiments in Section 4.

3.3.2 Correction factors

Our next improvement is based on the inherent redundancy in the Merkle tree structure. If an internal node represents a subset that provides a match between A and B, then each of its children should also match. To double-check that a match is not caused by a false positive at an internal node, we can also check its children. As we demonstrate in Section 4, this can significantly improve the accuracy at the expense of running time.

More generally, we change the search procedure so that it stops searching a path only when more than c consecutive matches are reported, where c is a new parameter we call the *correction factor*. For $c = 0$, this is exactly the same as the basic reconciliation procedure. For $c = 1$, the traversal is pruned after two consecutive matches, for $c = 2$, the traversal is pruned after three consecutive matches, and so on. If the branching factor is b, this slows down B's traversal by a factor of at most b^c. This leads to the following improvement.

Theorem 4 *There exists a one-round protocol with message size $O(|S_A|)$, construction time $O(|S_A| \log |S_A|)$ for A, construction time $O(|S_B| \log |S_B|)$ for B, reconciliation time $O(|S_B - S_A| \log^2 |S_B|)$ (with high probability), and constant accuracy.*

Proof: (Sketch.) We use the same approximate reconciliation tree construction as before, but use only a constant number of bits per node and hash functions in the Bloom filter. This makes the probability of an individual false positive during the search a constant. To make up for this, B uses a correction factor of $\Theta(\log \log |S_B|)$, which maintains the constant accuracy. □

3.3.3 Improved Bit Allocation

One implicit assumption in the basic approximate reconciliation tree construction is that all nodes have equal worth, since we use the same number of bits per node. But as noted earlier, false positives near the root of tree can lead to many or all of the differences being missed, an effect only partially mitigated by use of correction factors. Merely adding a few more bits to the root (and its close descendants) can significantly alleviate this.

At the opposite end of the tree, false positives at the leaves are impossible to correct. Therefore, the accuracy of approximate reconciliation trees is at best as accurate as the Bloom filter tests at the leaves, regardless of what is done for internal nodes. This further implies that the accuracy of an approximate reconciliation tree is no better than that of a Bloom filter of the same size. It also suggests that any schemes can reduce the number of bits used to describe internal nodes without compromising the accuracy would be desirable. In fact, one can prove that a reasonable approach is to use a separate Bloom filter for each level of the tree, whereby $\Theta(h)$ bits are used to represent each node at height h in the tree. A more elegant approach described in the following section.

3.3.4 Leveraging Random Tree Structure

A problem we have noted previously is that if we do not use a Bloom filter to represent nodes, then there may be difficulties in determining which nodes correspond in the peers' approximate reconciliation tree. On the other hand, using a Bloom filter increases the probability of an individual false positive. If possible, it would be desirable to have both the higher accuracy of a Merkle tree without the complications of reconstructing the tree structure.

To achieve such a hybrid, we first observe that given a random trie with n nodes (Patricia or otherwise), the first $\log n - 2\log\log n$ levels are complete with high probability. [2] Since these top levels are complete, reconstruction can be skipped and the values for these levels may simply be enumerated in some fixed order. Given this setup, B can access nodes in the first $\log|S_A| - 2\log\log|S_A|$ levels directly, yielding a lower false positive rate; B then switches to testing nodes in the lower levels of the tree with a Bloom filter. Even better, since there are only $\Theta(|S_A|/\log^2|S_A|)$ of these values, we can adopt the idea of varying numbers of bits per elements by using large hashes of $\Theta(\log|S_A|)$ bits without significantly affecting the number of bits available for the leaves and other lower levels. Specifically, this uses only $\Theta(|S_A|/\log|S_A|)$ bits but manages to avoid false positives in the upper levels with high probability.

[2] This is basically a balls and bins problem with n balls and $n/\log^2 n$ bins. A simple calculation shows that all bins have at least one ball with high probability.

3.4 Improved Theoretical Result

Using the various improvements of Section 3.3 and essentially the same analysis used piece-wise before, we can now prove Theorem 2.

Theorem 2 There exists a one-round protocol with message size $O(|S_A|)$, construction time $O(|S_A| \log |S_A|)$ for A, construction time $O(|S_B| \log |S_B|)$ for B, reconciliation time $O(|S_B - S_A| \log |S_B|)$ (with high probability), and constant accuracy.

Proof: A constructs an approximate reconciliation tree with the following components. First, values for the top $\log |S_A| - 2 \log \log |S_A|$ complete levels are explicitly sent using $O(\log |S_A|)$ bits per value. Values for the remaining internal nodes are sent in a Bloom filter using $\Theta(|S_A|)$ bits and a constant number of hash functions; similarly, the leaf nodes are sent in a separate Bloom filter using $\Theta(|S_A|)$ bits and a constant number of hash functions. Note that, without loss of generality, we may assume that $|S_B| \leq 2|S_A|$. Otherwise, the leaf Bloom filter itself can be used to satisfy the condition of the theorem.

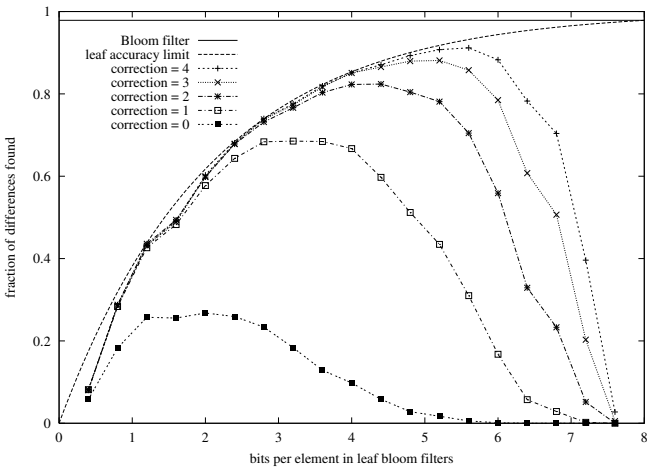
In the top levels of the tree, enough bits are used in the hashes to ensure that a false positive occurs along a path with probability only $O(\log |S_A|/|S_A|)$. For the remaining levels, B will use a correction level of $\Theta(\log \log \log |S_A|) = \Theta(\log \log \log |S_A|)$. This is only sufficient to guarantee a constant probability of a false positive along a path through the first $\log |S_A| + O(\log \log |S_A|)$ levels of the tree, but this depth encompasses a constant fraction of the leaf nodes, ensuring a constant accuracy. □

4 Experiments

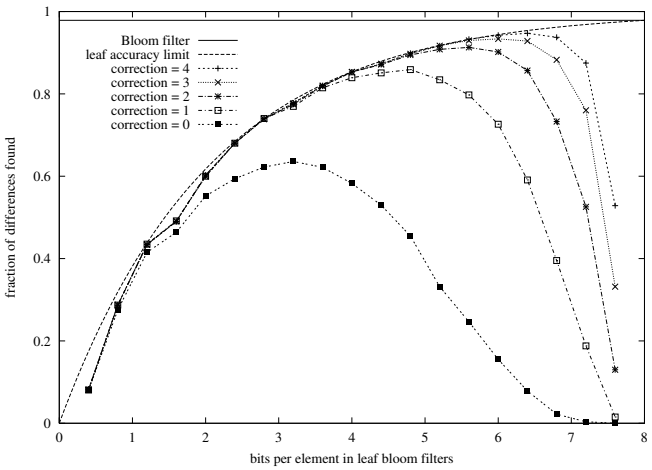
Our experiments focus on practical applications such as [2] and focus on the accuracy achievable with various implementations. Most of our experiments deal with sets of 10,000 random 32-bit elements with 100 differences. The total message size was 8 bits per element while the hashes used internally have 64 bits. Each data point shown is the average of 1,000 samples.

4.1 Accuracy Experiments

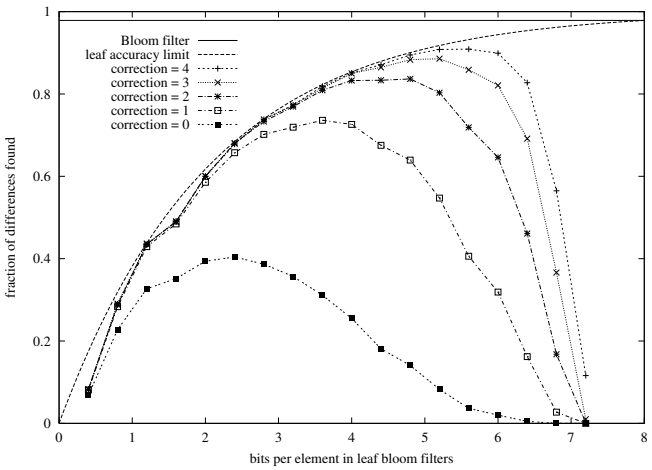
Figure 3 shows the results of our main set of experiments. Figure 3(a) shows the most basic implementation and the effects of varying the correction factor and redistributing bits between leaves and internal nodes. For comparison, there is also a curve of the accuracy of a the leaf Bloom filter by itself. This curve corresponds to arbitrarily high correction factor; that is, it corresponds to the accuracy of a Bloom filter with the same number of bits. Increasing the correction level used by B brings the accuracy closer to that of the leaf Bloom filter. At the same time, the optimal distribution of bits allocates more bits to the leaves as correction reduces the internal false positive rate.



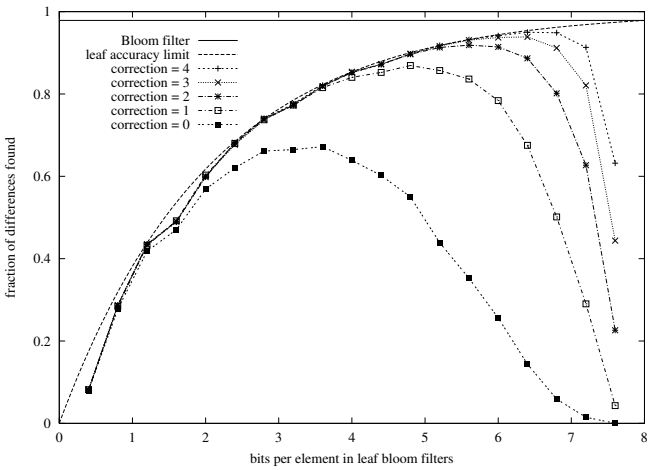
(a) branching factor 2, only bloom filters



(b) branching factor 4, only bloom filters



(c) branching factor 2, explicit top



(d) branching factor 4, explicit top

Figure 3: Comparison of various improvements. $|S_A| = |S_B| = 10000$, $|S_B - S_A| = 100$, 8 bits per element.

Figure 3(b) shows the result of simply increasing the branching factor from two to four. This produces a dramatic increase in accuracy since the tree depth is halved and the number of internal nodes drops.

Figure 3(c) shows the result of explicitly giving the values for the upper (complete) levels of the tree. There is a definite improvement over the basic approach, but not as much as changing the branching factor. This improvement enhances the scalability of approximate reconciliation trees, however; as the set sizes grow, the correction factor necessary to keep the accuracy constant grows more slowly with this improvement. Figure 3(d) shows the combine effect of these approaches.

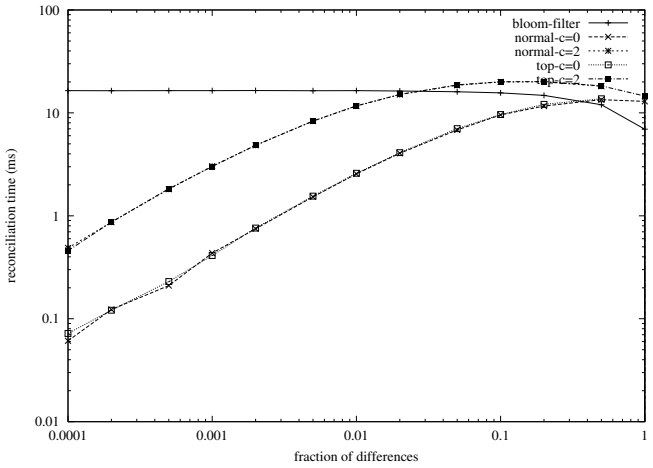


Figure 4: Speed comparison of various methods. $|S_A| = |S_B| = 10000$, $|S_B - S_A| = 100$, 8 bits per element.

4.2 Speed Experiments

We examine the relative speeds of the various approaches in Figure 4. We compare Bloom filters against basic approximate reconciliation trees and those with the explicit (complete) top levels, both with correction levels of zero and two. All of the approximate reconciliation trees use a branching factor of four and two Bloom filters, one for leaf nodes and one for internal nodes. The distribution of bits for the Bloom filters was chosen to maximize accuracy. For small numbers of differences, Bloom filters take significantly longer to reconcile. The reconciliation time taken by Bloom filters is roughly constant, but drops slightly as the number of differences increases. (This is because elements in the set will require computing all of the hash functions, while other elements may stop early once the Bloom filter reveals the element is not in the set.) The reconciliation time for approximate reconciliation trees grows roughly linearly with the number of differences and is initially very small. As with Bloom filters, there is also a drop in the time to reconcile when nearly all the elements are different.

As expected, approximate reconciliation trees using higher correction factors are slower. The difference between using the basic approach and explicitly listing the values of top levels is very small, except with small numbers of differences. For small numbers of differences, there is an appreciable constant factor in favor of the explicit listing since the bit-wise comparisons of these values are faster than performing Bloom filter tests. For larger numbers of differences, the number of nodes in these top levels is a small fraction of the total nodes traversed so the difference is negligible.

When using a correction level of two, approximate reconciliation trees are faster if the number of differences is fewer than 2% of $|S_B|$. Without correction, they are faster if the number of differences is fewer than 30% of $|S_B|$, but at the cost of significantly decreased accuracy. We note that besides changing with the correction level,

these crossover points vary with all the other parameters such as $|S_B|$, the number of bits per element, and the precise approach used (since they scale in different ways). The trend, however, is always the same; for small numbers of differences, approximate reconciliation trees are faster, but their advantage deteriorates as the correction level increases.

5 Conclusions

We have introduced approximate reconciliation of set differences as a useful primitive for peer-to-peer applications. Besides noting the efficacy of a Bloom-filter based solution, we have designed approximate reconciliation trees, an enhanced solution that minimizes reconciliation time. Experiments demonstrate that approximate reconciliation trees are suitably efficient for practical use.

We leave several open theoretical and practical questions. On the practical side, in future work we plan to consider how to best use approximate reconciliation methods to speed up exact reconciliation computations. On the theoretical side, it seems worthwhile to consider lower bounds on approximate reconciliation, in terms of the computation as well as the amount of communication necessary in bits and rounds.

References

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

[2] Byers, J., Considine, J., Mitzenmacher, M., and Rost, S. Informed content delivery across adaptive overlay networks. In *Proc. of ACM SIGCOMM '02* (August 2002).

[3] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1994.

[4] Fan, L., Cao, P., Almeida, J., and Broder, A. Summary cache: A scalable wide-area cache sharing protocol. *IEEE/ACM Trans. on Networking 8(3)* (2000), 281–293. A preliminary version appeared in Proc. of SIGCOMM '98.

[5] Knessl, C., and Szpankowski, W. Limit laws for heights in generalized tries and PATRICIA tries. In *Proc. LATIN 2000* (2000).

[6] Knuth, D. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[7] Merkle, R. Protocols for public key cryptosystems. In *IEEE Symposium on Security and Privacy* (1980), pp. 122–134.

[8] Minsky, Y., and Trachtenberg, A. Efficient reconciliation of unordered databases. Tech. Rep. TR1999-1778, Cornell University, 1999.

[9] Minsky, Y., and Trachtenberg, A. Practical set reconciliation. Tech. Rep. BU ECE 2002-01, Boston University, 2002.

[10] Minsky, Y., Trachtenberg, A., and Zippel, R. Set reconciliation with nearly optimal communication complexity. In *ISIT* (Washington, DC, June 2001).

[11] Mitzenmacher, M. Compressed bloom filters. In *Proc. of the 20th Annual ACM Symposium on Principles of Distributed Computing* (2001), pp. 144–150. To appear in IEEE/ACM Trans. on Networking.