What's the Difference? Efficient Set Reconciliation without Prior Context

David Eppstein[1] Michael T. Goodrich[1] Frank Uyeda[2] George Varghese[2,3]

[1]U.C. Irvine [2]U.C. San Diego [3]Yahoo! Research

ABSTRACT

We describe a synopsis structure, the Difference Digest, that allows two nodes to compute the elements belonging to the set difference in a single round with communication overhead proportional to the *size of the difference* times the logarithm of the keyspace. While set reconciliation can be done efficiently using logs, logs require overhead for every update and scale poorly when multiple users are to be reconciled. By contrast, our abstraction assumes no prior context and is useful in networking and distributed systems applications such as trading blocks in a peer-to-peer network, and synchronizing link-state databases after a partition.

Our basic set-reconciliation method has a similarity with the peeling algorithm used in Tornado codes [6], which is not surprising, as there is an intimate connection between set difference and coding. Beyond set reconciliation, an essential component in our Difference Digest is a new estimator for the size of the set difference that outperforms min-wise sketches [3] for small set differences.

Our experiments show that the Difference Digest is more efficient than prior approaches such as Approximate Reconciliation Trees [5] and Characteristic Polynomial Interpolation [17]. We use Difference Digests to implement a generic KeyDiff service in Linux that runs over TCP and returns the sets of keys that differ between machines.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols; E.4 [**Coding and Information Theory**]:

General Terms

Algorithms, Design, Experimentation

1. INTRODUCTION

Two common tasks in networking and distributed systems are *reconciliation* and *deduplication*. In reconciliation, two hosts each have a set of keys and each seeks to obtain the union of the two sets. The sets could be file blocks in a Peer-to-Peer (P2P) system or link state packet identifiers in a routing protocol. In deduplication, on the other hand, two hosts each have a set of keys, and the task is to identify the keys in the intersection so that duplicate data can be deleted and replaced by pointers [16]. Deduplication is a thriving industry: for example, Data Domain [1, 21] pioneered the use of deduplication to improve the efficiency of backups.

Both reconciliation and deduplication can be abstracted as the problem of efficiently computing the *set difference* between two sets stored at two nodes across a communication link. The set difference is the set of keys that are in one set but not the other. In reconciliation, the difference is used to compute the set union; in deduplication, it is used to compute the intersection. Efficiency is measured primarily by the bandwidth used (important when the two nodes are connected by a wide-area or mobile link), the latency in round-trip delays, and the computation used at the two hosts. We are particularly interested in optimizing the case when the set difference is small (e.g., the two nodes have almost the same set of routing updates to reconcile, or the two nodes have a large amount of duplicate data blocks) and when there is no prior communication or context between the two nodes.

For example, suppose two users, each with a large collection of songs on their phones, meet and wish to synchronize their libraries. They could do so by exchanging lists of all of their songs; however, the amount of data transferred would be proportional to the total number of songs they have rather than the size of the difference. An often-used alternative is to maintain a time-stamped log of updates together with a record of the time that the users last communicated. When they communicate again, A can send B all of the updates since their last communication, and vice versa. Fundamentally, the use of logs requires prior context, which we seek to avoid.

Logs have more specific disadvantages as well. First, the logging system must be integrated with any system that can change user data, potentially requiring system design changes. Second, if reconciliation events are rare, the added overhead to update a log each time user data changes may not be justified. This is particularly problematic for "hot" data items that are written often and may be in the log multiple times. While this redundancy can be avoided using a hash table, this requires further overhead. Third, a log has to be maintained for every other user this user may wish to synchronize with. Further, two users A and B may have received the same update from a third user C, leading to redundant communication. Multi-party synchronization is common in P2P systems and in cases where users have multiple data repositories, such as their phone, their desktop, and in the cloud. Finally, logs require stable storage and synchronized time which are often unavailable on networking devices such as routers.

To solve the set-difference problem efficiently without the use of logs or other prior context, we devise a data structure called a

SIGCOMM'11, August 15–19, 2011, Toronto, Ontario, Canada.

Difference Digest, which computes the set difference with communication proportional to the size of the difference between the sets being compared. We implement and evaluate a simple key-synchronization service based on Difference Digests and suggest how it can improve the performance in several contexts. Settings in which Difference Digests may be applied include:

- *Peer-to-peer:* Peer A and B may receive blocks of a file from other peers and may wish to receive only missing blocks from each other.
- *Partition healing:* When a link-state network partitions, routers in each partition may each obtain some new link-state packets. When the partition heals by a link joining router A and B, both A and B only want to exchange new or changed link-state packets.
- *Deduplication:* If backups are done in the cloud, when a new file is written, the system should only transmit the chunks of the file that are not already in the cloud.
- *Synchronizing parallel activations:* A search engine may use two independent crawlers with different techniques to harvest URLs but they may have very few URLs that are different. In general, this situation arises when multiple actors in a distributed system are performing similar functions for efficiency or redundancy.
- *Opportunistic ad hoc networks:* These are often characterized by low bandwidth and intermittent connectivity to other peers. Examples include rescue situations and military vehicles that wish to synchronize data when they are in range.

The main contributions of the paper are as follows:

- *IBF Subtraction:* The first component of the Difference Digest is an Invertible Bloom Filter or IBF [9, 13]. IBF's were previously used [9] for straggler detection at a single node, to identify items that were inserted and not removed in a stream. We adapt Invertible Bloom Filters for set reconciliation by defining a new subtraction operator on whole IBF's, as opposed to individual item removal.
- *Strata Estimator:* Invertible Bloom Filters need to be sized appropriately to be efficiently used for set differences. Thus a second crucial component of the Difference Digest is a new Strata Estimator method for estimating the *size of the difference*. We show that this estimator is much more accurate for small set differences (as is common in the final stages of a file dissemination in a P2P network) than Min-Wise Sketches [3, 4] or random projections [14]. Besides being an integral part of the Difference Digest, our Strata Estimator can be used independently to find, for example, which of many peers is most likely to have a missing block.
- *KeyDiff Prototype:* We describe a Linux prototype of a generic KeyDiff service based on Difference Digests that applications can use to synchronize objects of any kind.
- *Performance Characterization:* The overall system performance of Difference Digests is sensitive to many parameters, such as the size of the difference, the bandwidth available compared to the computation, and the ability to do precomputation. We characterize the parameter regimes in which Difference Digests outperform earlier approaches, such as MinWise hashes, Characteristic Polynomial Interpolation [17], and Approximate Reconciliation Trees [5].

Going forward, we discuss related work in Section 2. We present our algorithms and analysis for the Invertible Bloom Filter and Strata Estimator in Sections 3 & 4. We describe our KeyDiff prototype in Section 5, evaluate our structures in Section 6 and conclude in Section 7.

2. MODEL AND RELATED WORK

We start with a simple model of set reconciliation. For two sets S_A, S_B each containing elements from a universe, $\mathbb{U} = [0, u)$, we want to compute the set difference, D_{A-B} and D_{B-A}, where $D_{A-B} = S_A - S_B$ such that for all $s \in D_{A-B}$, $s \in S_A$ and $s \notin S_B$. Likewise, $D_{B-A} = S_B - S_A$. We say that $D = D_{A-B} \cup D_{B-A}$ and $d = |D|$. Note that since $D_{A-B} \cap D_{B-A} = \emptyset$, $d = |D_{A-B}| + |D_{B-A}|$. We assume that S_A and S_B are stored at two distinct hosts and attempt to compute the set difference with minimal communication, computation, storage, and latency.

Several prior algorithms for computing set differences have been proposed. The simplest consists of hosts exchanging lists, each containing the identifiers for all elements in their sets, then scanning the lists to remove common elements. This requires O($|S_A| + |S_B|$) communication and O($|S_A| \times |S_B|$) time. The run time can be improved to O($|S_A| + |S_B|$) by inserting one list into a hash table, then querying the table with the elements of the second list.

The communication overhead can be reduced by a constant factor by exchanging Bloom filters [2] containing the elements of each list. Once in possession of the remote Bloom Filter, each host can query the filter to identify which elements are common. Fundamentally, a Bloom Filter still requires communication proportional to the size of the sets (and not the difference) and incurs the risk of false positives. The time cost for this procedure is O($|S_A| + |S_B|$). The Bloom filter approach was extended using Approximate Reconciliation Trees [5], which requires O($d \log(|S_B|)$) recovery time, and O($|S_B|$) space. However, given S_A and an Approximate Reconciliation Tree for S_B, a host can only compute D_{A-B}, i.e., the elements unique to S_A.

An exact approach to the set-difference problem was proposed by Minksy *et al.* [17]. In this approach, each set is encoded using a linear transformation similar to Reed-Solomon coding. This approach has the advantage of O(d) communication overhead, but requires O(d^3) time to decode the difference using Gaussian elimination; asymptotically faster decoding algorithms are known, but their practicality remains unclear. Additionally, whereas our Strata Estimator gives an accurate one-shot estimate of the size of the difference prior to encoding the difference itself, Minsky *et al.* use an iterative doubling protocol for estimating the size of the difference, with a non-constant number of rounds of communication. Both the decoding time and the number of communication rounds (latency) of their system are unfavorable compared to ours.

Estimating Set-Difference Size. A critical sub-problem is an initial estimation of the size of the set difference. This can be estimated with constant overhead by comparing a random sample [14] from each set, but accuracy quickly deteriorates when the d is small relative to the size of the sets.

Min-wise sketches [3, 4] can be used to estimate the set similarity ($r = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}$). Min-wise sketches work by selecting k random hash functions $\pi_1, \ldots, \pi_k$ which permute elements within $\mathbb{U}$. Let $\min(\pi_i(S))$ be the smallest value produced by π_i when run on the elements of S. Then, a Min-wise sketch consists of the k values $\min(\pi_1(S))$, …, $\min(\pi_k(S))$. For two Min-wise sketches, M_A and M_B, containing the elements of S_A and S_B, respectively, the set similarity is estimated by the of number of hash function returning the same minimum value. If S_A and S_B have a set-similarity r, then we expect that the number of matching cells in M_A and M_B

will be $m = rk$. Inversely, given that m cells of M_A and M_B do match, we can estimate that $r = \frac{m}{k}$. Given the set-similarity, we can estimate the difference as $d = \frac{1-r}{1+r}(|S_A| + |S_B|)$.

As with random sampling, the accuracy of the Min-wise estimator diminishes for smaller values of k and for relatively small set differences. Similarly, Cormode *et al.* [8] provide a method for dynamic sampling from a data stream to estimate set sizes using a hierarchy of samples, which include summations and counts. Likewise, Cormode and Muthukrishnan [7] and Schweller *et al.* [20] describe sketch-based methods for finding large differences in traffic flows. (See also [19].)

An alternative approach to estimating set-difference size is provided by [11], whose algorithm can more generally estimate the difference between two functions with communication complexity very similar to our Strata Estimator. As with our results, their method has a small multiplicative error even when the difference is small. However, it uses algebraic computations over finite fields, whereas ours involves only simple, and more practical, hashing-based data structures.

While efficiency of set difference *estimation* for small differences may seem like a minor theoretical detail, it can be important in many contexts. Consider, for instance, the endgame of a P2P file transfer. Imagine that a BitTorrent node has 100 peers, and is missing only the last block of a file. Min-wise or random samples from the 100 peers will not identify the right peer if all peers also have nearly finished downloading (small set difference). On the other hand, sending a Bloom Filter takes bandwidth proportional to the number of blocks in file, which can be large. We describe our new estimator in Section 3.2.

3. ALGORITHMS

In this section, we describe the two components of the Difference Digest: an Invertible Bloom Filter (IBF) and a Strata Estimator. Our first innovation is taking the existing IBF [9, 13] and introducing a subtraction operator in Section 3.1 to compute D_{A-B} and D_{B-A} using a single round of communication of size O(d). Encoding a set S into an IBF requires O($|S|$) time, but decoding to recover D_{A-B} and D_{B-A} requires only O(d) time. Our second key innovation is a way of composing several sampled IBF's of fixed size into a new Strata Estimator which can effectively estimate the size of the set difference using O(log($|\mathbb{U}|$)) space.

3.1 Invertible Bloom Filter

We now describe the Invertible Bloom Filter (IBF), which can simultaneously calculate D_{A-B} and D_{B-A} using O(d) space. This data structure encodes sets in a fashion that is similar in spirit to Tornado codes' construction [6], in that it randomly combines elements using the XOR function. We will show later that this similarity is not surprising as there is a reduction between set difference and coding across certain channels. For now, note that whereas Tornado codes are for a fixed set, IBF's are dynamic and, as we show, even allow for fast set subtraction operations. Likewise, Tornado codes rely on Reed-Solomon codes to handle possible encoding errors, whereas IBF's succeed with high probability without relying on an inefficient fallback computation. Finally, our encoding is much simpler than Tornado codes because we use a simple uniform random graph for encoding while Tornado codes use more complex random graphs with non-uniform degree distributions.

We start with some intuition. An IBF is named because it is similar to a standard Bloom Filter—except that it can, with the right settings, be *inverted* to yield some of the elements that were inserted. Recall that in a counting Bloom Filter [10], when a key K is inserted, K is hashed into several locations of an array and a count, `count`, is incremented in each hashed location. Deletion of K is similar except that `count` is decremented. A check for whether K exists in the filter returns true if all locations that K hashes to have non-zero `count` values.

An IBF has another crucial field in each cell (array location) besides the count. This is the `idSum`: the XOR of all key IDs that hash into that cell. Now imagine that two peers, Peer 1 and Peer 2, doing set reconciliation on a large file of a million blocks independently compute IBF's, B_1 and B_2, each with 100 cells, by inserting an ID for each block they possess. Note that a standard Bloom filter would have a size of several million bits to effectively answer whether a particular key is contained in the structure. Observe also that if each ID is hashed to 3 cells, an average of 30,000 keys hash onto each cell. Thus, each `count` will be large and the `idSum` in each cell will be the XOR of a large number of IDs. What can we do with such a small number of cells and such a large number of collisions?

Assume that Peer 1 sends B_1 to Peer 2, an operation only requiring bandwidth to send around 200 fields (100 cells $\times$ 2 fields/cell). Peer 2 then proceeds to "subtract" its IBF B_2 from B_1. It does this cell by cell, by subtracting the `count` and XORing the `idSum` in the corresponding cells of the two IBF's.

Intuitively, if the two peers' blocks sets are almost the same (say, 25 different blocks out of a million blocks), all the common IDs that hash onto the same cell will be cancelled from `idSum`, leaving only the sum of the unique IDs (those that belong to one peer and not the other) in the `idSum` of each cell. This follows assuming that Peer 1 and Peer 2 use the same hash function so that any common element, c, is hashed to the same cells in both B_1 and B_2. When we XOR the `idSum` in these cells, c will disappear because it is XORed twice.

In essense, randomly hashing keys to, say three, cells, is identical to randomly throwing three balls into the same 100 bins for each of the 25 block ID's. Further, we will prove that if there are sufficient cells, there is a high probability that at least one cell is "pure" in that it contains only a single element by itself.

A "pure" cell signals its purity by having its `count` field equal to 1, and, in that case, the `idSum` field yields the ID of one element in the set difference. We delete this element from all cells it has hashed to in the difference IBF by the appropriate subtractions; this, in turn, may free up more pure elements that it can in turn be decoded, to ultimately yield all the elements in the set difference.

The reader will quickly see subtleties. First, a numerical `count` value of 1 is necessary but not sufficient for purity. For example, if we have a cell in one IBF with two keys X and Y and the corresponding cell in the second IBF has key Z, then when we subtract we will get a count of 1, but `idSum` will have the XOR of X, Y and Z. More devious errors can occur if four IDs W, X, Y, Z satisfy $W + X = Y + Z$. To reduce the likelihood of decoding errors to an arbitrarily small value, IBF's use a third field in each cell as a checksum: the XOR of the hashes of all IDs that hash into a cell, but using a different hash function H_c than that used to determine the cell indices. If an element is indeed pure, then the hash sum should be H_c(`idSum`).

A second subtlety occurs if Peer 2 has an element that Peer 1 does not. Could the subtraction $B_1 - B_2$ produce negative values for `idSum` and `count`? Indeed, it can and the algorithm deals with this: for example, in `idSum` by using XOR instead of addition and subtraction, and in recognizing purity by `count` values of 1 *or* -1. While IBF's were introduced earlier [9, 13], whole IBF subtraction is new to this paper; hence, negative counts did not arise in [9, 13].

Figure 1 summarizes the encoding of a set S and Figure 2 gives a small example of synchronizing the sets at Peer 1 (who has keys

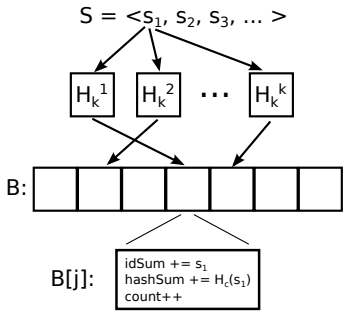


Figure 1: IBF Encode. Hash functions are used to map each element of the set to k cells of the IBF table.

V, W, X and Y) and Peer 2 (who has keys W, Y and Z). Each element is hashed into 3 locations: for example, X is hashed into buckets 1, 2 and 3. While X is by itself in bucket 3, after subtraction Z also enters, causing the `count` field to (incorrectly) be zero. Fortunately, after subtraction, bucket 4 becomes pure as V is by itself. Note that bucket 5 is also pure with Z by itself, and is signaled by a count of -1. Decoding proceeds by first deleting either V or Z, and then iterating until no pure cells remain.

Into how many cells should each element be hashed? We refer to this parameter as the `hash_count`. If the `hash_count` is too small, say 1, then there will be a high probability of finding pure cells initially, but once a pure element has been recorded and removed there are no other cells from which to remove it. Thus, two or more keys that have been hashed into the same cell cannot be decoded. On the other hand, if `hash_count` is too big, it is unlikely that there will be a pure element by itself to begin the process. We will show that `hash_count` values of 3 or 4 work well in practice.

Encode. First, assume that we have an oracle which, given S_A and S_B, returns the size of the set difference, d. We will describe the construction of such an oracle in Section 3.2. We allocate an IBF, which consists of a table B with $n = \alpha d$ cells, where $\alpha \geq 1$. Each cell of the table contains three fields (`idSum`, `hashSum` and `count`) all initialized to zero.

Additionally, hosts agree on two hash functions, H_c and H_k, that map elements in $\mathbb{U}$ uniformly into the space $[0, h)$, where $h \leq u$. Additionally, they agree on a value, k, called the `hash_count` which is the number of times each element is hashed. The algorithm for encoding a set S into an IBF is given in Algorithm 1 and illustrated in Figure 1. For each element in S, we generate k distinct random indices into B. To do this we recursively call $H_k()$ with an initial input of s_i and take the modulus by n until k distinct indices have been generated. More simply, an implementation could choose to use k independent hash functions. Regardless, for each index j returned, we XOR s_i into $B[j]$.`idSum`, XOR $H_c(s_i)$ into $B[j]$.`hashSum`, and increment $B[j]$.`count`.

Algorithm 1 IBF Encode

for $s_i \in S$ **do**
  for j in HashToDistinctIndices(s_i, k, n) **do**
    $B[j]$.`idSum` = $B[j]$.`idSum` $\oplus s_i$
    $B[j]$.`hashSum` = $B[j]$.`hashSum` $\oplus H_c(s_i)$
    $B[j]$.`count` = $B[j]$.`count` + 1

Subtract. For each index i in two IBF's, B_1 and B_2, we subtract $B_2[i]$ from $B_1[i]$. Subtraction can be done in place by writing the resulting values back to B_1, or non-destructively by writing values

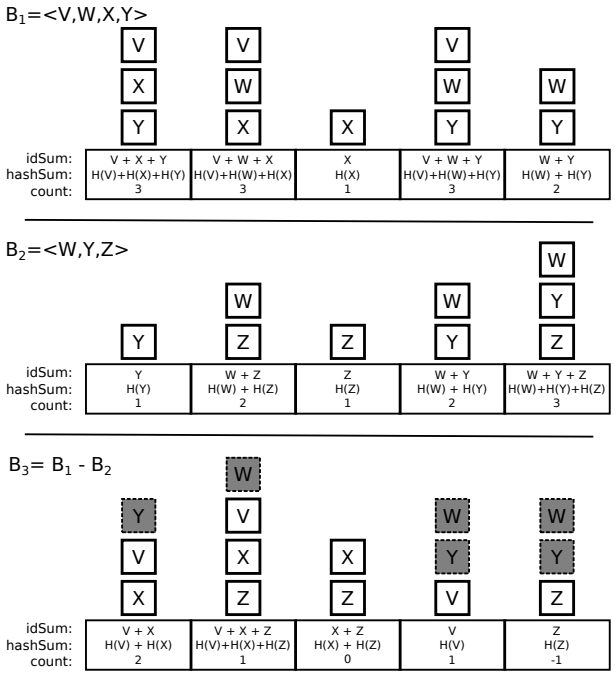


Figure 2: IBF Subtract. IBF B_3 results from subtracting IBF B_2 from IBF B_1 cell by cell. To subtract cells, the `idSum` and `hashSum` fields are XOR'ed, and `count` fields are subtracted. The elements common to B_1 and B_2 (shown shaded) are cancelled during the XOR operation.

to a new IBF of the same size. We present a non-destructive version in Algorithm 2. Intuitively, this operation eliminates common elements from the resulting IBF as they cancel from the `idSum` and `hashSum` fields as shown in Figure 2.

Algorithm 2 IBF Subtract ($B_3 = B_1 - B_2$)

for i in $0, \ldots, n-1$ **do**
  $B_3[i]$.`idSum` $= B_1[i]$.`idSum` $\oplus B_2[i]$.`idSum`
  $B_3[i]$.`hashSum` $= B_1[i]$.`hashSum` $\oplus B_2[i]$.`hashSum`
  $B_3[i]$.`count` $= B_1[i]$.`count`- $B_2[i]$.`count`

Decode. We have seen that to decode an IBF, we must recover "pure" cells from the IBF's table. Pure cells are those whose `idSum` matches the value of an element s in the set difference. In order to verify that a cell is pure, it must satisfy two conditions: the `count` field must be either 1 or -1, and the `hashSum` field must equal H_c(`idSum`). For example, if a cell is pure, then the sign of the `count` field is used to determine which set s is unique to. If the IBF is the result of subtracting the IBF for S_B from the IBF for S_A, then a positive `count` indicates $s \in D_{A-B}$, while a negative count indicates $s \in D_{B-A}$.

Decoding begins by scanning the table and creating a list of all pure cells. For each pure cell in the list, we add the value $s =$`idSum` to the appropriate output set (D_{A-B} or D_{B-A}) and remove s from the table. The process of removal is similar to that of insertion. We compute the list of distinct indices where s is present, then decrement `count` and XOR the `idSum` and `hashSum` by s and $H_c(s)$, respectively. If any of these cells becomes pure after s is removed, we add its index to the list of pure cells.

Decoding continues until no indices remain in the list of pure cells. At this point, if all cells in the table have been cleared (i.e. all

Algorithm 3 IBF Decode ($B \rightarrow D_{A-B}, D_{B-A}$)

```
for i = 0 to n − 1 do
  if B[i] is pure then
    Add i to pureList
while pureList ≠ ∅ do
  i=pureList.dequeue()
  if B[i] is not pure then
    continue
  s=B[i].idSum
  c=B[i].count
  if c > 0 then
    add s to D_{A−B}
  else
    add s to D_{B−A}
  for j in DistinctIndices(s, k, n) do
    B[j].idSum = B[j].idSum ⊕s
    B[j].hashSum = B[j].hashSum ⊕H_c(s)
    B[j].count = B[j].count- c
for i = 0 to n − 1 do
  if B[i].idSum ≠ 0 OR B[i].hashSum ≠ 0 B[i].count ≠ 0
  then
    return FAIL
return SUCCESS
```

fields have value equal to zero), then the decoding process has successfully recovered all elements in the set difference. Otherwise, some number of elements remain encoded in the table, but insufficient information is available to recover them. The pseudocode is given in Algorithm 3 and illustrated in Figure 3.

3.2 Strata Estimator

To use an IBF effectively, we must determine the approximate size of the set difference, d, since approximately $1.5d$ cells are required to successfully decode the IBF. We now show how to estimate d using $O(\log(u))$ data words, where u is the size of the universe of set values. If the set difference is large, estimators such as random samples [14] and Min-wise Hashing [3, 4] will work well. However, we desire an estimator that can accurately estimate very small differences (say 10) even when the set sizes are large (say million).

Flajolet and Martin (FM) [12] give an elegant way to estimate set sizes (not differences) using $\log(u)$ bits. Each bit i in the estimator is the result of sampling the set with probability $1/2^i$; bit i is set to 1, if at least 1 element is sampled when sampling with this probability. Intuitively, if there are $2^4 = 16$ distinct values in the set, then when sampling with probability $1/16$, it is likely that bit 4 will be set. Thus the estimator returns 2^I as the set size, where I is the highest strata (i.e., bit) such that bit I is set.

While FM data structures are useful in estimating the size of two sets, they do not help in estimating the size of the difference as they contain no information that can be used to approximate which elements are common. However, we can *sample the set difference* using the same technique as FM. Given that IBF's can compute set differences with small space, we use a hierarchy of IBF's as strata. Thus Peer A computes a logarithmic number of IBF's (strata), each of some small fixed size, say 80 cells.

Compared to the FM estimator for set sizes, this is very expensive. Using 32 strata of 80 cells is around 32 Kbytes but is the only estimator we know that is accurate at very small set differences and yet can handle set difference sizes up to 2^{32}. In practice, we build a lower overhead composite estimator that eliminates higher strata and replaces them with a MinWise estimator, which is more accurate for large differences. Note that 32 Kbytes is still inexpensive when compared to the overhead of naively sending a million keys.

Proceeding formally, we stratify $\mathbb{U}$ into $L = \log(u)$ partitions, $P_0, \ldots, P_L$, such that the range of the ith partition covers $1/2^{i+1}$ of $\mathbb{U}$. For a set, S, we encode the elements of S that fall into partition P_i into the ith IBF of the Strata Estimator. Partitioning $\mathbb{U}$ can be easily accomplished by assigning each element to the partition corresponding to the number of trailing zeros in its binary representation.

A host then transmits the Strata Estimator for its set to its remote peer. For each IBF in the Strata Estimator, beginning at stratum L and progressing toward stratum 0, the receiving host subtracts the corresponding remote IBF from the local IBF, then attempts to decode. For each successful decoding, the host adds the number of recovered elements to a counter. If the pair of IBF's at index i fails to decode, then we estimate that the size of the set difference is the value of the counter (the total number of elements recovered) scaled by 2^{i+1}. We give the pseudocode in Algorithms 4 and 5.

We originally designed the strata estimator by starting with stratum 0 and finding the first value of $i \geq 0$ which decoded successfully, following the Flajolet-Martin strategy and scaling the amount recovered by 2^i. However, the estimator in the pseudocode is much better because it uses information in *all* strata that decode successfully and not just the lowest such strata.

In the description, we assumed that the elements in S_A and S_B were uniformly distributed throughout $\mathbb{U}$. If this condition does not hold, the partitions formed by counting the number of low-order zero bits may skew the size of our partitions such that strata i does not hold roughly $|S|/2^{i+1}$. This can be easily solved by choosing some hash function, H_z, and inserting each element, s, into the IBF corresponding to the number of trailing zeros in $H_z(s)$.

Algorithm 4 Strata Estimator Encode using hash function H_z

```
for s ∈ S do
  i = Number of trailing zeros in H_z(s).
  Insert s into the i-th IBF
```

Algorithm 5 Strata Estimator Decode

```
count = 0
for i = log(u) down to −1 do
  if i < 0 or IBF_1[i] − IBF_2[i] does not decode then
    return 2^{i+1} × count
  count += number of elements in IBF_1[i] − IBF_2[i]
```

The obvious way to combine the Strata Estimator and IBF is to have node A request an estimator from node B, use it to estimate d, then request an IBF of size $O(d)$. This would take *two* rounds or at least two round trip delays. A simple trick is to instead have A initiate the process by sending its own estimator to B. After B receives the Strata Estimator from A, it estimates d and replies with an IBF, resulting in *one* round to compute the set difference.

4. ANALYSIS

In this section we review and prove theoretical results concerning the efficiency of IBF's and our stratified sampling scheme.

THEOREM 1. *Let S and T be disjoint sets with d total elements, and let B be an invertible Bloom filter with $C = (k+1)d$ cells, where $k =$* `hash_count` *is the number of random hash functions in B, and with at least $\Omega(k \log d)$ bits in each hashSum field. Suppose that (starting from a Bloom filter representing the empty set) each*

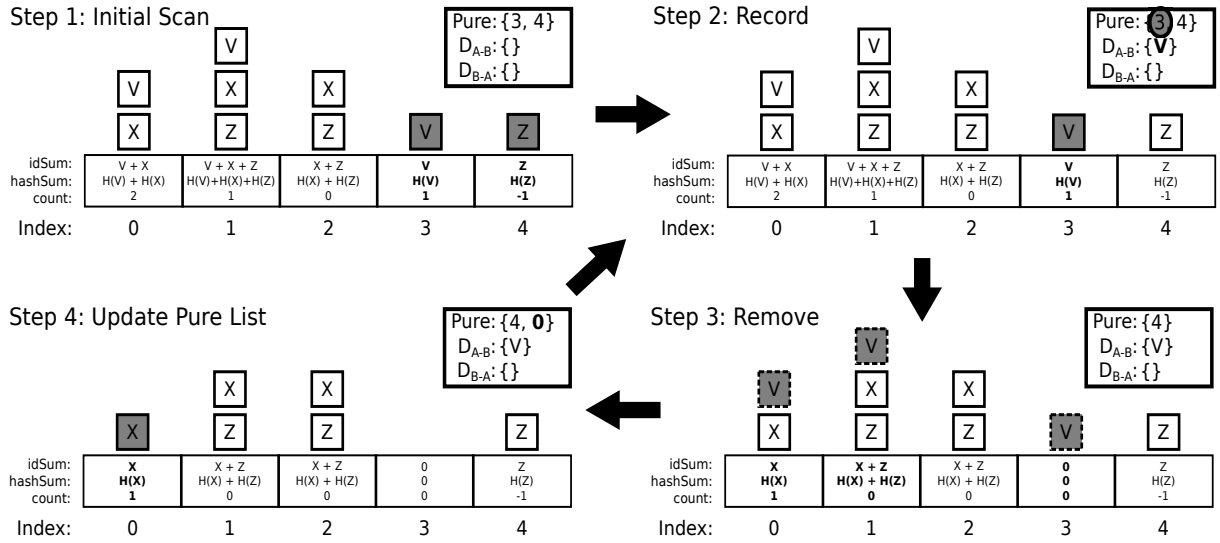


Figure 3: IBF Decode. We first scan the IBF for pure cells and add these indices (3&4) to the Pure list (Step 1). In Step 2, we dequeue the first index from the Pure list, then add the value of the `idSum` to the appropriate output set ($V \rightarrow D_{A-B}$). We then remove V from the IBF by using the k hash functions to find where it was inserted during encoding and subtracting it from these cells. Finally, if any of these cells have now become pure, we add them to the Pure list (Step 4). We repeat Steps 2 through 4 until no items remain in the Pure list.

item in S is inserted into B, and each item in T is deleted from B. Then with probability at most $O(d^{-k})$ the IBFDecode operation will fail to correctly recover S and T.

PROOF. The proof follows immediately from the previous analysis for invertible Bloom filters (e.g., see [13]). □

We also have the following.

COROLLARY 1. *Let S and T be two sets having at most d elements in their symmetric difference, and let B_S and B_T be invertible Bloom filters, both with the parameters as stated in Theorem 1, with B_S representing the set S and B_T representing the set T. Then with probability at most $O(d^{-k})$ we will fail to recover S and T by applying the IBFSubtract operation to B_S and B_T and then applying the IBFDecode operation to the resulting invertible Bloom filter.*

PROOF. Define $S' = S - T$ and $T' = T - S$. Then S' and T' are disjoint sets of total size at most d, as needed by Theorem 1. □

The corollary implies that in order to decode an IBF that uses 4 independent hash functions with high probability, then one needs an overhead of $k + 1 = 5$. In other words, one has to use $5d$ cells, where d is the set difference. Our experiments later, however, show that an overhead that is somewhat less than 2 suffices. This gap is narrowed in [13] leveraging results on finding a 2-core (analogous to a loop) in random hypergraphs.

Let us next consider the accuracy of our stratified size estimator. Suppose that a set S has cardinality m, and let i be any natural number. Then the ith stratum of our strata estimator for S has expected cardinality $m/2^i$. The following lemma shows that our estimators will be close to their expectations with high probability.

LEMMA 1. *For any $s > 1$, with probability $1 - 2^{-s}$, and for all $j < i$, the cardinality of the union of the strata numbered j or greater is within a multiplicative factor of $1 \pm O(\sqrt{s2^i/m})$ of its expectation.*

PROOF. Let Y_j be the size of the union of strata numbered j or greater for $j = 0, 1, \ldots, i$, and let μ_j be its expectation, that is, $\mu_j = E(Y_j) = m/2^j$. By standard Chernoff bounds (e.g., see [18]), for $\delta > 0$,

$$\Pr\left(Y_j > (1+\delta)\mu_j\right) < e^{-\mu_j \delta^2/4}$$

and

$$\Pr\left(Y_j < (1-\delta)\mu_j\right) < e^{-\mu_j \delta^2/4}.$$

By taking $\delta = \sqrt{4(s+2)(2^i/m)\ln 2}$, which is $O(\sqrt{s2^i/m})$, we have that the probability that a particular Y_j is not within a multiplicative factor of $1 \pm \delta$ of its expectation is at most

$$2e^{-\mu_j \delta^2/4} \leq 2^{-(s+1)2^{i-j}}.$$

Thus, by a union bound, the probability that any Y_j is not within a multiplicative factor of $1 \pm \delta$ of its expectation is at most

$$\sum_{j=0}^{i} 2^{-(s+1)2^{i-j}},$$

which is at most 2^{-s}. □

Putting these results together, we have the following:

THEOREM 2. *Let ϵ and δ be constants in the interval $(0, 1)$, and let S and T be two sets whose symmetric difference has cardinality d. If we encode the two sets with our strata estimator, in which each IBF 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 set difference within a factor of $1 \pm \delta$ of d.*

PROOF. By the same reasoning as in Corollary 1, each IBF at level i in the estimator is a valid IBF for a sample of the symmetric difference of S and T in which each element is sampled with probability $1/2^{i+1}$. By Theorem 1, having each IBF be of $C = (k+1)g$

cells, where $k = \lceil \log 1/\epsilon \rceil$ and $g \geq 2$, then we can decode a set of g elements with probability at least $1 - \epsilon/2$.

We first consider the case when $d \leq c_0^2 \delta^{-2} \log(1/\epsilon)$, where c_0 is the constant in the big-oh of Lemma 1. That is, the size of the symmetric difference between S and T is at most a constant depending only on ϵ and δ. In this case, if we take

$$g = \max\left\{2, \lceil c_0^2 \delta^{-2} \log(1/\epsilon) \rceil\right\}$$

and $k = \lceil \log 1/\epsilon \rceil$, then the level-0 IBF, with $C = (k+1)d$ cells, will decode its set with probability at least $1 - \epsilon/2$, and our estimator will learn the exact set-theoretic difference between S and T, without error, with high probability.

Otherwise, let i be such that $d/2^i \approx c_0^2 \delta^2 / \log(1/\epsilon)$ and let $g = \max\left\{2, \lceil c_0^2 \delta^{-2} \log(1/\epsilon) \rceil\right\}$ and $k = \lceil \log 1/\epsilon \rceil$, as above. So, with probability $1 - \epsilon/2$, using an IBF of $C = (k+1)d$ cells, we correctly decode the elements in the ith stratum, as noted above. By Lemma 1 (with $s = \lceil \log 1/\epsilon \rceil + 1$), with probability $1 - \epsilon/2$, the cardinality of the number of items included in the ith and higher strata are within a multiplicative factor of $1 \pm \delta$ of its expectation. Thus, with high probability, our estimate for d is within a $1 \pm \delta$ factor of d. □

Comparison with Coding: Readers familiar with Tornado codes will see a remarkable similarity between the decoding procedure used for Tornado codes and IBF's. This follows because set reconciliation and coding are equivalent. Assume that we have a systematic code that takes a set of keys belonging to a set S_A and codes it using check words C_1 through C_n. Assume further that the code can correct for up to d erasures and/or insertions. Note that Tornado codes are specified to deal only with erasures not insertions.

Then, to compute set difference A could send the check words C_1 through C_n *without* sending the "data words" in set S_A. When B receives the check words, B can treat its set S_B as the data words. Then, together with the check words, B can compute S_A and hence the set difference. This will work as long as S_B has at most d erasures or insertions with respect to set S_A.

Thus any code that can deal with erasures and insertions can be used for set difference, and vice versa. A formal statement of this equivalence can be found in [15]. This equivalence explains why the existing deterministic set difference scheme, CPI, is analogous to Reed-Solomon coding. It also explains why IBF's are analogous to randomized Tornado codes. While more complicated Tornado-code like constructions could probably be used for set difference, the gain in space would be a small constant factor from say $2d$ to $d + \epsilon$, the increased complexity is not worthwhile because the real gain in set reconciliation is going down from $O(n)$ to $O(d)$, where n is the size of the original sets.

5. THE KEYDIFF SYSTEM

We now describe *KeyDiff*, a service that allows applications to compute set differences using Difference Digests. As shown in Figure 4, the KeyDiff service provides three operations, `add`, `remove`, and `diff`. Applications can add and remove keys from an instance of KeyDiff, then query the service to discover the set difference between any two instances of KeyDiff.

Suppose a developer wants to write a file synchronization application. In this case, the application running at each host would map files to unique keys and add these keys to a local instance of KeyDiff. To synchronize files, the application would first run KeyDiff's `diff` operation to discover the differences between the set stored locally and the set stored at a remote host. The application can then perform the reverse mapping to identify and transfer the files that differ between the hosts.

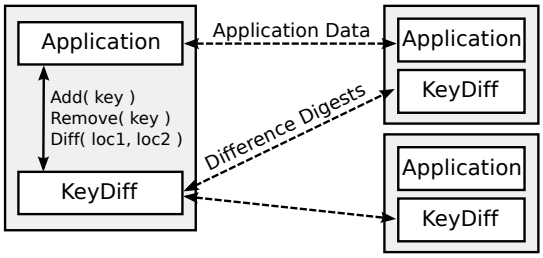


Figure 4: KeyDiff computes the set difference between any two instances and returns this information to the application.

The KeyDiff service is implemented using a client-server model and is accessed through an API written in C. When a client requests the difference between its local set and the set on a remote host, KeyDiff opens a TCP connection and sends a request containing an estimator. The remote KeyDiff instance runs the estimation algorithm to determine the approximate size of the difference, then replies with an IBF large enough for the client to decode with high-probability. All requests and responses between KeyDiff instances travel over a single TCP connection and the `diff` operation completes with only a single round of communication.

KeyDiff provides faster `diff` operations through the use of pre-computation. Internally, KeyDiff maintains an estimator structure that is statically sized and updated online as keys are added and removed. However, computing the IBF requires the approximate size of the difference, a value that is not know until after the estimation phase. In scenarios where computation is a bottleneck, KeyDiff can be configured to maintain several IBF's of pre-determined sizes online. After the estimation phase, KeyDiff returns the best pre-computed IBF. Thus, the computational cost of building the IBF can be amortized across all of the calls to `add` and `remove`.

This is reasonable because the cost of incrementally updating the Strata Estimator and a few IBF's on key update is small (a few microseconds) and should be much smaller than the time for the application to create or store the object corresponding to the key. For example, if the application is a P2P application and is synchronizing file blocks, the cost to store a new block on disk will be at least a few milliseconds. We will show in the evaluation that if the IBF's are precomputed, then the latency of `diff` operations can be 100's of microseconds for small set differences.

6. EVALUATION

Our evaluation seeks to provide guidance for configuring and predicting the performance of Difference Digests and the KeyDiff system. We address four questions. First, what are the optimal parameters for an IBF? (Section 6.1). Second, how should one tune the Strata Estimator to balance accuracy and overhead? (Section 6.2). Third, how do IBF's compare with the existing techniques? (Section 6.3). Finally, for what range of differences are Difference Digests most effective compared to the brute-force solution of sending all the keys? (Section 6.4).

Our evaluation uses a $\mathbb{U}$ of all 32-bit values. Hence, we allocate 12 bytes for each IBF cell, with 4 bytes given to each `idSum`, `hashSum` and `count` field. As input, we created pairs of sets containing *keys* from $\mathbb{U}$. The keys in the first set, S_A, were chosen randomly without replacement. We then chose a random subset of S_A and copied it to the second set, S_B. We exercised two degrees of freedom in creating our input sets: the number of keys in S_A, and the size of the difference between S_A and S_B, which we refer to as the experiment's *delta*. For each experiment we created 100

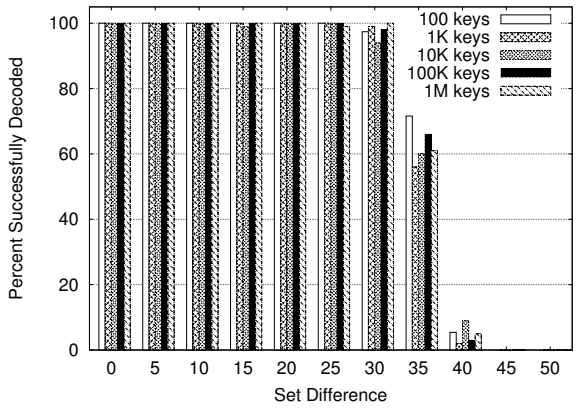


Figure 5: Rate of successful IBF decoding with 50 cells and 4 hash functions. The ability to decode an IBF scales with the size of the set difference, not the size of the sets.

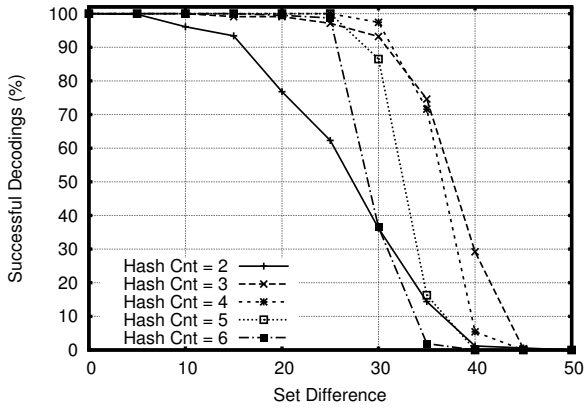


Figure 6: Probability of successful decoding for IBF's with 50 cells for different deltas. We vary the number of hashes used to assign elements to cells (`hash_count`) from 2 to 6.

file pairs. As our objective is set reconciliation, we consider an experiment successful only if we are able to successfully determine all of the elements in the set difference.

6.1 Tuning the IBF

We start by verifying that IBF size scales with the size of the set difference and not the total set size. To do so, we generated sets with 100, 1K, 10K, 100K and 1M keys, and deltas between 0 and 50. We then compute the set difference using an IBF with 50 cells.

Figure 5 shows the success rate for recovering the entire set difference. We see that for deltas up to 25, the IBF decodes completely with extremely high probability, regardless of set size. At deltas of 45 and 50 the collision rate is so high that no IBF's are able to completely decode. The results in Figure 5 confirm that decoding success is independent of the original set sizes.

Determining IBF size and number of hash functions. Both the number of IBF cells and `hash_count` (the number of times an element is hashed) are critical in determining the rate of successful decoding. To evaluate the effect of `hash_count`, we attempted to decode sets with 100 keys and deltas between 0 and 50 using an IBF with 50 cells and `hash_count`'s between 2 and 6. Since the size of the sets does not influence decoding success, these results are representative for arbitrarily large sets. We ran this configuration for 1000 pairs of sets and display our results in Figure 6.

For deltas less than 30, `hash_count` = 4 decodes 100% of the time, while higher and lower values show degraded success rates. Intuitively, lower hash counts do not provide equivalent decode rates since processing each pure cell only removes a key from a small number of other cells, limiting the number of new pure cells that may be discovered. Higher values of `hash_count` avoid this problem but may also decrease the odds that there will initially be a pure cell in the IBF. For deltas greater than 30, `hash_count` = 3 provides the highest rate of successful decoding. However, at smaller deltas, 3 hash functions are less reliable than 4, with approximately 98% success for deltas from 15 to 25 and 92% at 30.

To avoid failed decodings, we must allocate IBF's with more than d cells. We determine appropriate memory overheads (ratio of IBF's cells to set-difference size) by using sets containing 100K elements and varying the number of cells in each IBF as a proportion of the delta. We then compute the memory overhead required for 99% of the IBF's to successfully decode and plot this in Figure 7.

Deltas below 200 all require at least 50% overhead to completely decode. However, beyond deltas of 1000, the memory overhead reaches an asymptote. As before, we see a `hash_count` of 4 decodes consistently with less overhead than 5 or 6, but interestingly, `hash_count` = 3 has the lowest memory overhead at all deltas greater than 200.

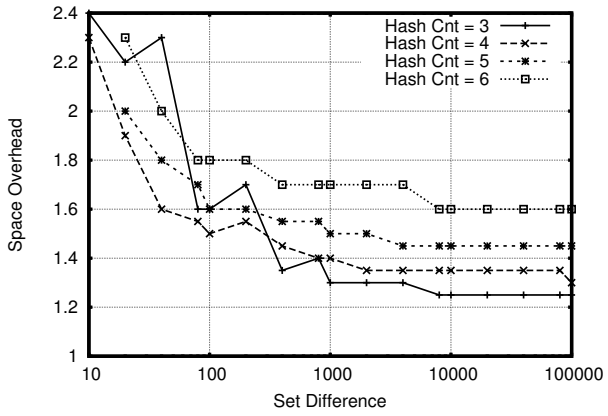


Figure 7: We evaluate sets containing 100K elements and plot the minimum space overhead (IBF cells/delta) required to completely recover the set difference with 99% certainty.

6.2 Tuning the Strata Estimator

To efficiently size our IBF, the Strata Estimator provides an estimate for d. If the Strata Estimator over-estimates, the subsequent IBF will be unnecessarily large and waste bandwidth. However, if the Strata Estimator under-estimates, then the subsequent IBF may not decode and cost an expensive transmission of a larger IBF. To prevent this, the values returned by the estimator should be scaled up so that under-estimation rarely occurs.

In Figure 8, we report the scaling overhead required for various strata sizes such that 99% of the estimates will be greater than or equal to the true difference. Based on our findings from Section 6.1, we focus on Strata Estimators whose fixed-size IBF's use a `hash_count` of 4. We see that the scaling overhead drops sharply as the number of cells per IBF is increased from 10 to 40, and reaches a point of diminishing returns after 80 cells. With 80 cells per stratum, any estimate returned by a Strata Estimator

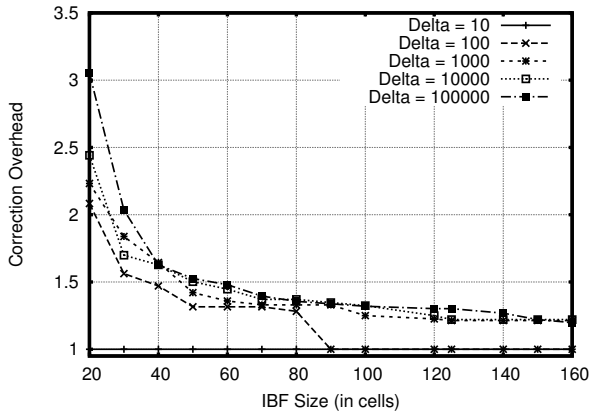


Figure 8: Correction overhead needed by the Strata Estimator to ensure that 99% of the estimates are greater than or equal to the true value of d when using strata IBF's of various sizes.

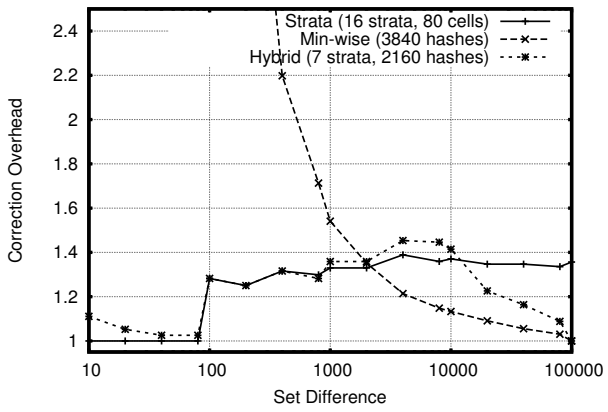


Figure 9: Comparison of estimators when constrained to 15.3 KB. We show the scaling overhead to ensure that 99% of estimates are greater than or equal to the true delta.

should be scaled by a factor of 1.39 to ensure that it will be greater than or equal to d 99% of the time.

Strata Estimator vs. Min-wise. We next compare our Strata Estimator to the Min-wise Estimator [3, 4] (see Section 2). For our comparison we used sets with 100K elements and deltas ranging from 10 to 100K. Given knowledge of the approximate total set sizes a priori, the number of strata in the Strata estimator can be adjusted to conserve communication costs by only including partitions that are likely to contain elements from the difference. Thus, we choose the number of strata to be $\lfloor \log_2(d_{max}) \rfloor$, where d_{max} is the largest possible difference. Since our largest delta is 100K, we configure our estimator with 16 strata, each containing 80 cells per IBF. At 12 bytes per IBF cell, this configuration requires approximately 15.3 KB of space. Alternatively, one could allocate a Min-wise estimator with 3840 4-byte hashes in the same space.

In Figure 9, we compare the scaling overhead required such that 99% of the estimates from Strata and Min-wise estimators of the same size are greater than or equal to the true delta. We see that the overhead required by Min-wise diminishes from 1.35 to 1.0 for deltas beyond 2000. Strata requires correction between 1.33 to 1.39 for the same range. However, the accuracy of the Min-wise estimator deteriorates rapidly for smaller delta values. In fact, for all deltas below 200, the 1st percentile of Min-wise estimates are 0, resulting in infinite overhead. Min-wise's inaccuracy for small deltas is expected as few elements from the difference will be included in the estimator as the size of the difference shrinks. This makes Min-wise very sensitive to any variance in the sampling process, leading to large estimation errors. In contrast, we see that the Strata Estimator provides reasonable estimates for all delta values and is particularly good at small deltas, where scaling overheads range between 1.0 and 1.33.

Hybrid Estimator. The Strata Estimator outperforms Min-wise for small differences, while the opposite occurs for large differences. This suggests the creation of a hybrid estimator that keeps the lower strata to accurately estimate small deltas, while augmenting more selective strata with a single Min-wise estimator. We partition our set as before, but if a strata does not exist for a partition, we insert its elements into the Min-wise estimator. Estimates are performed by summing strata as before, but also by including the number of differences that Min-wise estimates to be in its partition.

For our previous Strata configuration of 80 cells per IBF, each strata consumes 960 bytes. Therefore, we can trade a strata for 240 additional Min-wise hashes. From Figure 9 we see that the Strata Estimator performs better for deltas under 2000. Thus, we would like to keep at most $\lfloor \log_2(2000) \rfloor = 10$ strata. Since we expect the most selective strata to contain few elements for a difference of 2000, we are better served by eliminating them and giving more space to Min-wise. Hence, we retain 7 strata, and use the remaining 8640 bytes to allocate a Min-wise estimator with 2160 hashes.

Results from our Hybrid Estimator are plotted in Figure 9 with the results from the Strata and Min-wise estimators. We see that the Hybrid Estimator closely follows the results of the Strata Estimator for all deltas up to 2000, as desired. For deltas greater than 2000, the influence of errors from both Strata and Min-wise cause the scaling overhead of the Hybrid estimator to drift up to 1.45 (versus 1.39% for Strata), before it progressively improves in accuracy, with perfect precision at 100K. While the Hybrid Estimator slightly increase our scaling overhead from 1.39 to 1.45 (4.3%), it also provides improved accuracy at deltas larger than 10% where over-estimation errors can cause large increases in total data sent.

Difference Digest Configuration Guideline. By using the Hybrid Estimator in the first phase, we achieve an estimate greater than or equal to the true difference size 99% of the time by scaling the result by 1.45. In the second phase, we further scale by 1.25 to 2.3 and set `hash_count` to either 3 or 4 depending on the estimate from phase one. In practice, a simple rule of thumb is to construct an IBF in Phase 2 with twice the number of cells as the estimated difference to account for both under-estimation and IBF decoding overheads. For estimates greater than 200, 3 hashes should be used and 4 hashes otherwise.

6.3 Difference Digest vs. Prior Work

We now compare Difference Digests to Approximate Reconciliation Trees (`ART`) [5], Characteristic Polynomial Interpolation (`CPISync`) [17], and simply trading a sorted list of keys (`List`). We note that `ART`'s were originally designed to compute *most but not all* the keys in $S_A - S_B$. To address this, the system built in [5] used erasure coding techniques to ensure that hosts received pertinent data. While this approach is reasonable for some P2P applications it may not be applicable to or desirable for all applications described in Section 1. In contrast, `CPISync` and `List` are always able to recover the set difference.

Figure 10 shows the data overhead required by the four algorithms. Given that `ART`'s were not designed for computing the

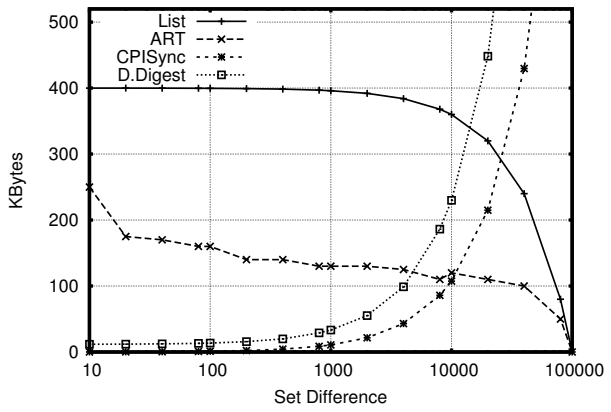


Figure 10: Data transmission required to reconcile sets with 100K elements. We show the space needed by ART and Difference Digests to recover 95% and 100% of the set difference, respectively, with 99% reliability.

complete set difference, we arbitrarily choose the standard of 95% of the difference 99% of the time and plot the amount of data required to achieve this level of performance with ART. For CPISync, we show the data overhead needed to recover the set difference. In practice, the basic algorithm must be run with knowledge of the difference size, or an interactive approach must be used at greater computation and latency costs. Hence, we show the best case overhead for CPISync. Finally, we plot the data used by Difference Digest for both estimation and reconciliation to compute the complete difference 99% of the time.

The results show that the bandwidth required by List and ART decreases as the size of the difference increases. This is intuitive since both List and the ART encode the contents of S_B, which is diminishing in size as the size of the difference grows ($|S_B| = |S_A| - |D|$). However, ART outperforms List since its compact representation requires fewer bits to capture the nodes in B.

While the size of the Hybrid Estimator stays constant, the IBF grows at an average rate of 24 Bytes (three 4-byte values and a factor of 2 inflation for accurate decoding) per key in the difference. We see that while Difference Digests and CPISync have the same asymptotic communication complexity, CPISync requires less memory in practice at approximately 10 bytes per element in the difference.

Algorithm Selection Guidelines. The results show that for small differences, Difference Digests and CPISync require an order of magnitude less bandwidth that ART and are better up to a difference of 4,000 (4%) and 10,000 (10%), respectively. However, the fact that ART uses less space for large deltas is misleading since we have allowed ART to decode only 95% of the difference. CPISync provides the lowest bandwidth overhead and decodes deterministically, making it a good choice when differences are small and bandwidth is precious. However, as we discuss next, its computational overhead is substantial.

6.4 KeyDiff Performance

We now examine the benefits of Difference Digest in system-level benchmarks using KeyDiff service described in Section 5. We quantify the performance of KeyDiff using Difference Digests versus ART, CPISync, and List. For these experiments, we deployed KeyDiff on two dual-processor quad-core Xeon servers running Linux 2.6.32.8. Our test machines were connected via 10 Gbps Ethernet and report an average RTT of 93μs.

Computational Overhead. In the KeyDiff model, the applications at the client and server both add keys to KeyDiff. When diff is called the client requests the appropriate data from the server to compute the set difference. We begin by evaluating the computation time required to add new keys to KeyDiff and the time required by the server to generate its response when using each algorithm. For this experiment, we added 1 million keys to the KeyDiff server then requested a structure to decode a difference of 100. In Table 1, we show the average time required for these operations.

For adding new keys, we see that List and ART are slower than both CPISync and IBF since both perform $\log(|S|)$ operations — the List to do an ordered insertion, and ART to update hashes along a path in its tree. In contrast, CPISync and IBF simply store the new keys to an unordered list until they learn the size of the structure to build from the client's request.

For server compute time, we note that the latencies correspond closely with the number of memory locations each algorithm touches. List is quickest at 6.545 msec, as the server only needs to read and serialize the keys. In contrast, IBF and CPISync must allocate and populate appropriately-sized structures by scanning their stored lists of keys. For IBF this requires updating 4 cells per key, while CPISync must evaluate 100 linear equations, each involving all 1M keys. The time for ART is roughly twice that of IBF as it must traverse its tree containing $2|S|$ nodes, to build a Bloom Filter representation of its set.

At the bottom of Table 1, we show the add and server-compute times for the three estimation algorithms. Note that since the length of the IBF's in the Strata and Hybrid estimators are static, they can be updated online. We also note that although the Hybrid estimator maintains 2160 Min-wise hash values, its addition times are significantly lower that the original Min-wise estimator. This occurs because the Min-wise structure in the Hybrid estimator only samples the elements not assigned to one of the seven strata. Thus, while the basic Min-wise estimator must compute 3840 hashes for each key, the Hybrid Min-wise estimator only computes hashes for approximately $1/2^8$ of the elements in the set. Since each of the estimators is updated during its add operations, minimal server compute time is required to serialize each structure.

Costs and Benefits of Incremental Updates. As we have seen in Table 1, the server computation time for many of the reconciliation algorithms is significant and will negatively affect the speed of our diff operation. The main challenge for IBF and CPISync is that the size of the difference must be known before a space-efficient structure can be constructed. We can avoid this runtime computation by maintaining several IBF's of predetermined sizes within KeyDiff. Each key is added to all IBF's and, once the size of the difference is know, the smallest suitable IBF is returned. The number of IBF's to maintain in parallel will depend on the computing and bandwidth overheads encountered for each application.

In Table 1, we see that the time to add a new key when doing precomputation on 8 IBF's takes 31μs, two orders of magnitude longer than IBF without precomputation. However, this reduces the server compute time to from 3.9 seconds to 22μs, a massive improvement for diff latency. For most applications, the small cost (10's of microseconds) for incremental updates during each add operation should not dramatically affect overall application performance, while the speedup during diff (seconds) is a clear advantage.

Diff Performance. We now look at time required to run the diff operation. As we are primarily concerned with the performance of computing set differences as seen by an application built on top of these algorithms, we use incremental updates and measure

Reconciliation Algorithm	Add (μs)	Serv. Compute
List (sorted)	1.309	6.545 msec
ART	1.995	6,937.831 msec
CPISync (d=100)	0.216	34,051.480 msec
IBF (no precompute)	0.217	3,957.847 msec
IBF (1x precompute)	3.858	0.023 msec
IBF (8x precompute)	31.320	0.022 msec

Estimation Algorithms (precompute)		
Min-wise (3840 hashes)	21.909	0.022 msec
Strata (16x80 cells)	4.224	0.021 msec
Hybrid (7x80 cells + 2160 hash)	4.319	0.023 msec

Table 1: Time required to add a key to KeyDiff and the time required to generate a KeyDiff response for sets of 1M keys with a delta of 100. The time per `add` call is averaged across the insertion of the 1M keys.

the wall clock time required to compute the set difference. Difference Digests are run with 15.3 KB dedicated to the Estimator, and 8 parallel, precomputed IBF's with sizes ranging from 256 to 400K cells in factors of 4. To present the best case scenario, CPISync was configured with foreknowledge of the difference size and the correctly sized CPISync structure was precomputed at the server side. We omit performance results for ART as it has the unfair advantage of only *approximating* the membership of D_{A-B}, unlike the other algorithms, which return all of D_{A-B} and D_{B-A}.

For our tests, we populated the first host, A with a set of 1 million, unique 32-bit keys, S_A, and copied a random subset of those keys, S_B, to the other host, B. From host A we then query KeyDiff to compute the set of unique keys. Our results can be seen in Figure 11a. We note that there are 3 components contributing to the latency for all of these methods, the time to generate the response at host B, the time to transmit the response, and the time to compare the response to the set stored at host A. Since we maintain each data structure online, the time for host B to generate a response is negligible and does not affect the overall latency.

We see from these results that the List shows predictable performance across difference sizes, but performs particularly well relative to other methods as the size of the difference increases beyond 20K. Since the size of the data sent *decreases* as the size of the difference increases, the transmission time and the time to sort and compare at A decrease accordingly. On the other hand, the Difference Digest performs best at small set differences. Since the estimator is maintained online, the estimation phase concludes very quickly, often taking less than 1 millisecond. We note that precomputing IBF's at various sizes is essential and significantly reduces the latency by the IBF's construction at host B at runtime. Finally, we see that even though its communication overhead is very low, the cubic decoding complexity for CPISync dramatically increases its latency at differences larger than 100.

In considering the resources required for each algorithm, List requires $4|S_B|$ bytes in transmission and touches $|S_A| + |S_B|$ values in memory. Difference Digest has a constant estimation phase of 15.3KB followed by an average of $24d$ bytes in transmission and $3d \times$ `hash_count` memory operations (3 fields in each IBF cell). Finally, CPISync requires only $10d$ bytes of transmission to send a vector of sums from it's linear equations, but d^3 memory operations to solve its matrix.

If our experimental setup were completely compute bound, we would expect List to have superior performance for large differences and Difference Digest to shine for small difference. If we assume a hash_count of 4, then Difference Digest's latency is $12d$, while List's is $|S_A| + |S_B| = 2|S_A| - d$. Thus, they will have equivalent latency at $d = \frac{2}{12+1}|S_A|$, or a difference of 15%.

Guidance for Constrained Computation. We conclude that, for our high-speed test environment, precomputed Difference Digests are superior for small deltas (less than 2%), while sending a full list is preferable at larger difference sizes. We argue that in environments where computation is severely constrained relative to bandwidth, this crossover point can reach up to 15%. In such scenarios, precomputation is vital to optimize diff performance.

Varying Bandwidth. We now investigate how the latency of each algorithm changes as the speed of the network decreases. For this we consider bandwidths of 10 Mbps and 100Kbps, which are speeds typical in wide-area networks and mobile devices, respectively. By scaling the transmission times from our previous experiments, we are able to predict the performance at slower network speeds. We show these results in Figure 11b and Figure 11c.

As discussed previously, the data required by List, CPISync, and Difference Digests is $4|S_B|$, $10d$ and roughly $24d$, respectively. Thus, as the network slows and dominates the running time, we expect that the low bandwidth overhead of CPISync and Difference Digests will make them attractive for a wider range of deltas versus the sorted List. With only communication overhead, List will take $4|S_B| = 4(|S_A| - d)$, which will equal Difference Digest's running time at $d = \frac{4}{24+4}|S_A|$, or a difference of 14%. As the communication overhead for Difference Digest grows due to widely spaced precomputed IBF's the trade off point will move toward differences that are a smaller percentage of the total set size. However, for small to moderate set sizes and highly constrained bandwidths KeyDiff should create appropriately sized IBF's on demand to reduce memory overhead and minimize transmission time.

Guidance for Constrained Bandwidth. As bandwidth becomes a predominant factor in reconciling a set difference, the algorithm with the lowest data overhead should be employed. Thus, List will have superior performance for differences greater than 14%. For smaller difference sizes, IBF will achieve faster performance, but the crossover point will depend on the size increase between precomputed IBF's. For constrained networks and moderate set sizes, IBF's could be computed on-demand to optimize communication overhead and minimize overall latency.

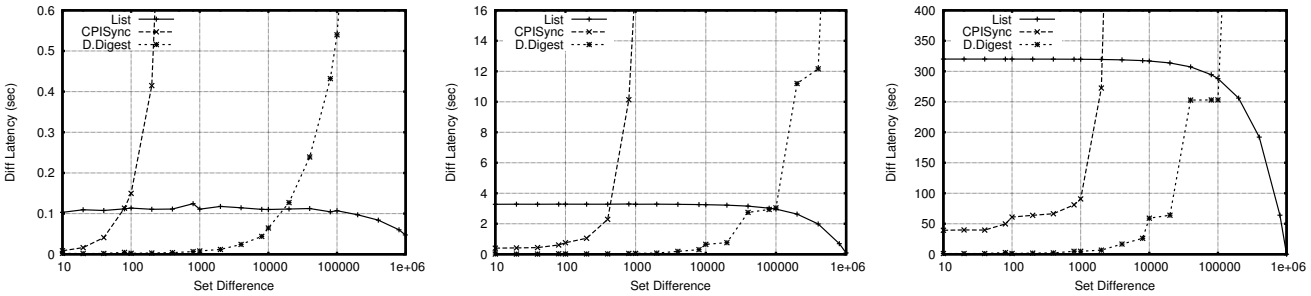
7. CONCLUSIONS

We have shown how Difference Digests can efficiently compute the set difference of data objects on different hosts using computation and communication proportional to the size of the set difference. The constant factors are roughly 12 for computation (4 hash functions, resulting in 3 updates each) and 24 for communication (12 bytes/cell scaled by two for accurate estimation and decoding).

The two main new ideas are whole set differencing for IBF's, and a new estimator that accurately estimates small set differences via a hierarchy of sampled IBF's. One can think of IBF's as a particularly simple random code that can deal with both erasures and insertions and hence uses a simpler structure than Tornado codes but a similar decoding procedure.

We learned via experiments that 3 to 4 hash functions work best, and that a simple rule of thumb is to size the second phase IBF equal to twice the estimate found in the first phase. We implemented Difference Digests in a KeyDiff service run on top of TCP that can be utilized by different applications such as Peer-to-Peer file transfer. There are three calls in the API (Figure 4): calls to add and delete a key, and a call to find the difference between a set of keys at another host.

In addition, using 80 cells per strata in the estimator worked well and, after the first 7 strata, augmenting the estimator with Min-



(a) Measured - 1.2 Gbps (b) Modeled - 10 Mbps (c) Modeled - 100 Kbps

Figure 11: Time to run KeyDiff `diff` for $|S_A|$= 1M keys and varying difference sizes. We show our measured results in 11a, then extrapolate the latencies for more constrained network conditions in 11b and 11c.

wise hashing provides better accuracy. Combined as a Difference Digest, the IBF and Hybrid Estimator provide the best performance for differences less than 15% of the set size. This threshold changes with the ratio of bandwidth to computation, but could be estimated by observing the throughput during the estimation phase to choose the optimal algorithm.

Our system level benchmarks show that Difference Digests must be precomputed or the latency for computation at runtime can swamp the gain in transmission time compared to simple schemes that send the entire list of keys. This is not surprising as computing a Difference Digest touches around *twelve* 32-bit words for each key processed compared to *one* 32-bit word for a naive scheme that sends a list of keys. Thus, the naive scheme can be 10 times faster than Difference Digests *if we do no precomputation*. However, with precomputation, if the set difference is small, then Difference Digest is ten times or more faster. Precomputation adds only a few microseconds to updating a key.

While we have implemented a generic Key Difference service using Difference Digests, we believe that a KeyDiff service could be used by some application involving either reconciliation or deduplication to improve overall user-perceived performance. We hope the simplicity and elegance of Difference Digests and their application to the classical problem of set difference will also inspire readers to more imaginative uses.

8. ACKNOWLEDGEMENTS

This research was supported by grants from the NSF (CSE-2589 & 0830403), the ONR (N00014-08-1-1015) and Cisco. We also thank our shepherd, Haifeng Yu, and our anonymous reviewers.

9. REFERENCES

[1] Data domain. http://www.datadomain.com/.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, 1970.

[3] A. Broder. On the resemblance and containment of documents. *Compression and Complexity of Sequences, '97*.

[4] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60:630–659, 2000.

[5] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *SIGCOMM*, 2002.

[6] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, 1998.

[7] G. Cormode and S. Muthukrishnan. What's new: finding significant differences in network data streams. *IEEE/ACM Trans. Netw.*, 13:1219–1232, 2005.

[8] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. *VLDB '05*.

[9] D. Eppstein and M. Goodrich. Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters. *IEEE Trans. on Knowledge and Data Engineering*, 23:297–306, 2011.

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

[11] J. Feigenbaum, S. Kannan, M. J. Strauss, and M. Viswanathan. An approximate L^1-difference algorithm for massive data streams. *SIAM Journal on Computing*, 32(1):131–151, 2002.

[12] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. of Computer and System Sciences*, 31(2):182 – 209, 1985.

[13] M. T. Goodrich and M. Mitzenmacher. Invertible Bloom Lookup Tables. *ArXiv e-prints*, 2011. 1101.2245.

[14] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. *STOC*, 1998.

[15] M. Karpovsky, L. Levitin, and A. Trachtenberg. Data verification and reconciliation with generalized error-control codes. *IEEE Trans. Info. Theory*, 49(7), july 2003.

[16] P. Kulkarni, F. Douglis, J. Lavoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *USENIX ATC*, 2004.

[17] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Trans. Info. Theory*, 49(9):2213 – 2218, 2003.

[18] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge Univ. Press, 1995.

[19] S. Muthukrishnan. Data streams: Algorithms and applications. *Found. Trends Theor. Comput. Sci.*, 2005.

[20] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M.-Y. Kao, and G. Memik. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Trans. Netw.*, 15:1059–1072, 2007.

[21] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08*.