False-Positive Probability and Compression Optimization for Tree-Structured Bloom Filters

YONGQUAN FU, National Key Laboratory for Parallel and Distributed Processing; College of Computer Science, National University of Defense Technology
ERNST BIERSACK, Caipy

Bloom filters are frequently used to to check the membership of an item in a set. However, Bloom filters face a dilemma: the transmission bandwidth and the accuracy cannot be optimized simultaneously. This dilemma is particularly severe for transmitting Bloom filters to remote nodes when the network bandwidth is limited. We propose a novel Bloom filter called **BloomTree** that consists of a tree-structured organization of smaller Bloom filters, each using a set of independent hash functions. BloomTree spreads items across levels that are compressed to reduce the transmission bandwidth need. We show how to find optimal configurations for BloomTree and investigate in detail by how much BloomTree outperforms the standard Bloom filter or the compressed Bloom filter. Finally, we use the intersection of BloomTrees to predict the set intersection, decreasing the false-positive probabilities by several orders of magnitude compared to both the compressed Bloom filter and the standard Bloom filter.

CCS Concepts: • **Mathematics of computing → Probabilistic algorithms**; • **Theory of computation → Data structures and algorithms for data management**; **Database query processing and optimization (theory);**

Additional Key Words and Phrases: Set query, Bloom filter, tree, compression, genetic algorithm

ACM Reference Format:
Yongquan Fu and Ernst Biersack. 2016. False-positive probability and compression optimization for tree-structured Bloom filters. ACM Trans. Model. Perform. Eval. Comput. Syst. 1, 4, Article 19 (September 2016), 39 pages.
DOI: http://dx.doi.org/10.1145/2940324

1. INTRODUCTION

1.1. Motivation

With the development of geo-distributed applications, it is often necessary to check whether a set contains a specific item, or whether multiple sets have shared items. For example, CDNs need to locate the requested objects stored in the cache servers or to synchronize the objects in different cache servers [Maggs and Sitaraman 2015], database systems need to perform *semi-join* operations to compute the shared items among remote nodes [Fang et al. 1998], peer-to-peer applications locate replicas from remote users [Cheng and Liu 2009], or enterprise networks deduplicate identical files among hybrid clouds [Eppstein et al. 2011; Fu et al. 2013].

This work was supported by the National Natural Science Foundation of China under Grant No. 61402509.
Authors' addresses: Y. Fu, National Key Laboratory for Parallel and Distributed Processing; College of Computer Science, National University of Defense Technology, 410073, Sanyi Road, Changsha, Hunan Province, China; email: yongquanf@nudt.edu.cn; E. Biersack, CAIPY, 06560 Valbonne, France; email: erbi@e-biersack.eu.

DOI: http://dx.doi.org/10.1145/2940324

A Bloom filter (BF) is a compact data structure answering set queries. The Standard Bloom Filter (SBF) represents the set of items by randomly hashing each item into k locations of a bit array, where k denotes the number of hash functions. To test whether an item belongs to a set, we compute the locations of this item with the same set of hash functions and test whether these bits are all set to one: If yes, then the item is considered to be in the set; otherwise, the item is said to be not contained in the set. Furthermore, we can approximate the union and intersection of multiple sets efficiently with a SBF: applying bitwise "AND" ("OR") operations over two or multiple SBFs yields a new Bloom filter representing the intersection (union) of corresponding sets. The Bloom filter incurs a certain false-positive (FP) probability, since the hash locations corresponding to a particular item may be already set to one by other items. However, a Bloom filter never has false negatives, that is, it will never occur that an item is hashed to the bit array, but the query fails to report the existence of this item. To control the false-positive probability, the size of the BF must grow linearly with the number of items in the set.

The BF has been extensively used as a static data structure for set queries in one site, while with the proliferation of cache servers in CDNs, summarizing the content in different cache servers is important to locate the closest cache that has the requested object [Maggs and Sitaraman 2015]. Unfortunately, each cache server may store millions of objects and the cache contents may vary dynamically; consequently, each cache server needs to periodically exchange the summary with the other servers. As a result, representing the summary as a BF and transferring the whole BF without compression may be expensive. In this case, using a Compressed Bloom Filter (CBF) may decrease the transmission cost [Mitzenmacher 2002]. The CBF usually has only two hash functions, which significantly degrades accuracy compared to the standard Bloom filter with an optimal number of hash functions. As a consequence, there is a gap that BloomTree tries to bridge.

However, the SBF has an interesting dilemma: *Given a fixed-size bit array, selecting the optimal number of hash functions that minimizes the FP probability makes compression ineffective, while using fewer than the optimal number of hash functions increases the compression efficiency but degrades the FP probability*. To simultaneously optimize the false-positive probability and compression efficiency, we need to keep the percentage of bits set to one as small as possible. Unfortunately, there exists little room for optimization, as the bit locations set to one are completely determined by uniform-random hash functions: The power of two choices [Lumetta and Mitzenmacher 2007] and the partitioned hashing [Hao et al. 2007] try to overlap as many bit locations that have already been set to one as possible for incoming items. To that end, they don't use the same hash functions for each item. This approach decreases the FP probability only marginally, since biasing towards specific bits is difficult, as every hash function selects positions uniformly at random from the whole bit array. Further, when we select different hash functions for each item in a filter, the union and intersection of BFs are no longer valid, as an item may be mapped to different bit locations at different BFs.

1.2. BloomTree

We have designed and implemented a novel tree-structured BF called BloomTree that bypasses the performance dilemma by simultaneously decreasing the average FP probability and optimizing the transmission cost. While tree-structured BFs have been proposed previously, they all have regular degrees, that is, each internal vertex has the same number of descendants except the leaf vertices in the bottom level, which unnecessarily limits the design space. Further, due to the difficulties in deriving the false-positive probability of the set-membership query, parameter optimization for tree-structured filters is still an open problem. We address both issues in this article.

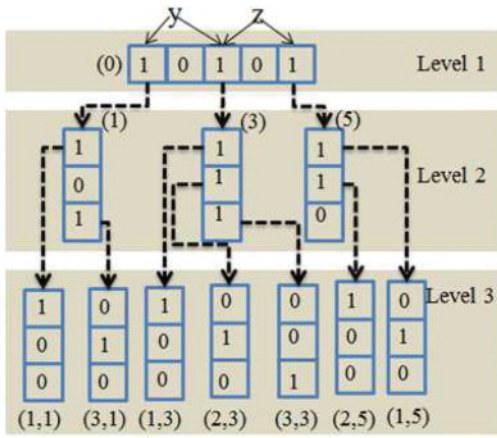


Fig. 1. A logical structure of a three-level BloomTree instance. The numbers of hash functions for the filters in the first, second, and third levels are 2, 2, and 1, respectively.

A BloomTree is logically a tree of BFs that implements the set-membership query, as shown in Figure 1. We can compute the intersection of two BloomTrees efficiently, like the SBF, in order to find the common items in two sets that are represented by these BloomTrees.

Structure: Each node in the BloomTree is an SBF. For each bit of a node at the ith level, where $i < d$, there exists a descendent node at level $(i+1)$. As a result, the degree of each internal vertex depends on the length of the filter, which varies across levels (see Figure 1).

Insert: We recursively insert an item into the BloomTree in a top-down approach. First, at the root of the tree, we insert the item into the SBF and record the hash locations. In the example shown in Figure 1, the root SBF has two hash functions. For a given item y, the hash locations in the root filter are the first and third bits. These two bits correspond to two descendent filters (1) and (3) at the second level. For each of these descendent filters, we recursively insert the item into these SBFs at the second level. Then, we continue the same insertion procedure recursively until we complete the insertion in the descendent filters (1, 1), (3, 1), (2, 3), and (3, 3) at the third level.

Query: Similar to the insertion process, we recursively test the existence of the item in a top-down approach. We first test whether this item is hashed into the root SBF. If so, we select the descendent BFs of the hash locations in the root level and query the descendants. If any of these traversed filters indicates that the item is not in the set, the query process terminates and returns a negative answer immediately. This is because each filter is a standard BF that never causes *false negatives*. Otherwise, when each of these descendent filters claims the item to be in the set, we recursively query the descendants in the next level. For example, to query the set membership of the item y in Figure 1, we calculate the hash locations for y in the first level, which are all ones. As a result, we continue the query on the next level. For y, the descendants in the second level are (1) and (3). Querying the item y on (1) and (3) returns true. Next, we recursively select the descendants of (1) and (3), which are (1, 1), (3, 1), (2, 3), and (3, 3). All of these filters claim that y is in the set. Therefore, the BloomTree claims that the item y is in the set.

False Positives: An item is said to be in the set *if and only if* all visited BFs report that the item is in the set. An FP event in the BloomTree occurs for a membership query *if and only if the visited BFs all report that an item is in the set*, although this item

was never inserted. By ensuring the independence of the false-positive events among participating filters, the overall false-positive probability amounts to the product of the FP probabilities of the participating filters. As a result, BloomTree reduces the FP probability exponentially as the tree grows.

Compression: The BloomTree can be efficiently compressed, similar to the compressed BF, by separately applying arithmetic coding on the bit array representing all filters of one level. We use a pointerless bit array to store the BloomTree in the main memory and compute the hash values with the double-hashing method [Kirsch and Mitzenmacher 2008].

The tree structure provides novel opportunities to optimize the FP probabilities by improving the locality of the hash locations of the data items, which, in turn, reduces the false-positive probability. In Section 5, we derive the FP probability of BloomTree and analyze its sensitivity with respect to the key parameters, namely, the size of the SBFs and the number of hash functions. In Section 6, we optimize the parameters of BloomTree to balance the FP probability and compression efficiency using two different approaches: an exhaustive search method that enumerates every possible configuration and a Genetic algorithm [Goldberg 1989]–based method that evolves toward optimized parameters within a fixed number of rounds. While the exhaustive search always finds the optimal parameters, its time complexity increases exponentially in the number of levels. The Genetic algorithm achieves close-to-optimal results with a fixed time complexity. Therefore, we recommend the Genetic algorithm to obtain optimized parameters for BloomTrees with $d \geq 3$ levels.

We performed extensive experiments to compare BloomTree with the relevant data structures proposed in the literature. Our results confirm that BloomTree obtains a significantly better trade-off between FP probability and the transmission bandwidth.

In summary, this article makes the following contributions:

—We present BloomTree, a compact tree-structured organization of BFs that simultaneously optimizes the FP probability and transmission size.
—We propose a framework to tune the accuracy of the BloomTree filter, balancing the accuracy and transmission size of BloomTree.
—We carry out extensive experiments using synthetic datasets and a real-world trace to confirm that BloomTree finds a better trade-off between the FP probability and compression efficiency than state-of-the-art techniques.

The rest of the article is organized as follows. Section 2 summarizes the most relevant studies that optimize the standard BF. Section 3 states the problem studied in this article. Section 4 presents an overview of BloomTree. Section 5 analyzes the distribution of the FP probabilities of BloomTree. Section 6 optimizes BloomTree in terms of transmission bandwidth and FP probability. Section 7 shows the sensitivity of the FP probabilities as we vary the parameters of BloomTree. Next, Section 8 reports on an extensive performance comparison between BloomTree and related approaches. Section 9 presents our conclusions and discusses future work. Table I lists the key notations used throughout the article.

2. RELATED WORK

BFs and their variants have been extensively studied; it is beyond the scope of this article to survey all of them. We report only the studies most relevant to our own.

2.1. Accuracy-Oriented Optimization

The FP probability of the BF should be controlled to provide a correct response for most set queries. The standard BF uses uniform-random hash functions, therefore, the percentage of ones in the bit array is determined by the hash functions used. In

Table I. Notations Used in BloomTree

I	bit array
m	number of items
k	number of hash functions
P_b	probability of a false positive before the BF is constructed
P_a	posterior FP probability after the BF is built
n	number of items that has been inserted
$n_{\max}$	upper bound on the number of items inserted
d	number of levels of a BloomTree instance
m_i	length of the bit array of a filter at the ith level ($i \geq 1$)
k_i	number of hash functions of a filter at the ith level ($i \geq 1$)
ρ_1	bits per item in the first level
M_{BT}	storage size of a BT instance
W_{BT}	transmission size of a BT instance

Lumetta and Mitzenmacher [2007], an optimized group of hash functions is selected for each new item so that the percentage of ones of the filter does increase very slowly.

To decrease the computing overhead, the partitioned hashing [Hao et al. 2007] first randomly partitions items into a set of disjoint groups, then optimizes the hash functions for each group of items via an exhaustive search process. Unfortunately, experiments [Lumetta and Mitzenmacher 2007; Hao et al. 2007] show that the improvement is marginal, as every hash function maps the item to a random bit over the whole bit array. Worse yet, the partitioned hashing no longer allows for union and intersection operations, since the same item may not be mapped to the same set of bit locations at two filters. To control the FP probability of the multicast-group query over the BF, Li et al. [2011] propose selecting different numbers of hash functions for different multicast-address groups. As a result, different groups see a different query accuracy, which is not suitable for general settings in which queries are expected to see a consistent accuracy. Our work improves the accuracy for the general set queries.

2.2. Access Time–Oriented Optimization

The access time involves the time to compute the hash values and the time to read and set the bit values. To decrease the access time, we should control both components.

The shifting Bloom filter (ShBF) [Yang et al. 2016] trades off between memory consumption and processing speed using a flat filter. ShBF encodes the auxiliary information of an item into a location offset in the standard BF. The double-hashing method [Mitzenmacher and Vadhan 2008] generates a family of independent hash functions using two hash functions as the seeds.

Putze et al. [2009] maintain a bank of equal-size BFs, in which each filter is stored into one cache block in order to decrease cache misses. In fact, this structure can be seen as a two-level BloomTree: at the first level, an all-one bit array with one hash function routes each item to one BF at the second level. On the other hand, our work proposes a general tree-structured BF. In Putze et al. [2009], each query involves only one filter, and the FP probability of a query amounts to that of the selected BF. Our work develops a multilevel tree-structured filter in which multiple BFs collectively encode the existence of an item. Therefore, the FP probability of the query process for our work is much lower since it amounts to the product of the FPs of all BFs participating in the query.

2.3. Transmission Efficiency–Oriented Optimization

To reduce the overhead when transmitting a BF, we can compress the BF before sending it. The CBF [Mitzenmacher 2002] trades off a significantly larger memory size for a

Table II. Comparison of Related Tree-Structured Filters

	Purpose	Topology	Storage	False positive	Query	Compress	Parameter optimize
Ficara et al. [2008]	Counting BF	2-ary tree	A bank of bit arrays	no closed-form equation	traverse a directed acyclic graph	no	no
Koloniari et al. [2011]	Duplicate detection	2-ary tree	A bank of bit arrays plus pointers	the FP probability of any filter	traverse a chain of filters	no	no
Alexiou et al. [2013]	Range interval query	2-ary tree	Two bit arrays	the FP probability of the leaf vertex	traverse a chain of filters	no	no
Yoon et al. [2014]	Key-value query	k_l-ary tree, $k_l \geq 2$	A bit array	the product of the FP probability of a chain of filters	traverse a chain of filters	no	no
Crainiceanu and Lemire [2015]; Solomon and Kingsford [2015]	Summary BFs	varying degree tree	B+ tree	the probability that at least one leaf filter reports an FP event	traverse a chain or subtree of filters	no	no
BloomTree (our work)	Set query	varying degree tree	A bit array	the product of the FP probabilities of all traversed filters	traverse a subtree of the filters	yes	yes

smaller transmission size: A CBF that achieves the same FP probability as an SBF allocates more bits per item, but uses fewer hash functions than an SBF. Therefore, after the insertion of n elements, the bit array I of a CBF will have few bits set to one and many bits that remain set to zero, which can be compressed before transmission.

As discussed earlier, Putze et al. [2009] stores a bank of BFs in cache blocks. To compress the storage space of the filter, Putze et al. [2009] represent the BF using a sorted list of the hash values. Moreover, it needs to store an index data structure for the compression, while our work compresses the tree-structured filter layer by layer using the arithmetic coding without additional storage costs.

2.4. Tree-Structured Optimization

We summarize the key design choices of existing tree-structured filters and discuss our work for comparison in Table II. We can see that only our work performs the parameter optimization, while existing work leaves this as an open problem. Parameter optimization is nontrivial due to the randomized FP probability of the query process. Further, existing work sets the same degree for all vertices in the tree, which simplifies the implementation, but unnecessarily narrows down the design space. Our work uses different degrees across levels to significantly decrease the overall FP probability.

The Blooming tree [Ficara et al. 2008] constructs a binary tree of BFs as an alternative to the counting BF. A Blooming tree has $L+2$ levels for inserting n items that are built level by level, where the level $i+1$ has as many blocks of bits as the number of ones in level i. The final level $L+2$ is composed of an array of counters for recording the insertions and deletions of items.

Being designed to support the key-value query, Yoon et al. [2014] (denoted as BTree) is logically structured as a d-ary complete search tree, but is stored in a flat bit array.

For a (key, v) pair, there exists one unique path from the root vertex to the vth leaf vertex, where each vertex in the path hashes the key into its bit array. BTree meets an FP event when each of the vertices in the traversed chain has aFP event. A vertex in BTree needs to check d sets of hash functions for a query; if the hash locations using any set of hash functions are all set to one, then the vertex has an FP event. Therefore, the FP probability of a vertex amounts to the probability of the union of the FP events caused by d sets of hash functions. While in BloomTree, each vertex has only one group of hash functions, which avoids the accumulation of the FP probabilities in BTree.

Koloniari et al. [2011] detects duplicated events among geo-distributed sites using a binary tree of filters in which the ancestor–descendant link is represented using a pointer. The query process starts from a leaf and traverses towards the root vertex, yielding a chain-structured query process. An FP event occurs whenever any vertex in the chain incurs an FP event. In contrast, we minimize the FP probability to be the product of a subtree of all filters that participate in the query process.

Alexiou et al. [2013] represents range intervals using a binary-tree structured filter. Each vertex represents a subinterval of its ancestor with a small-sized BF. The query process starts at the root, and routes recursively across levels to the descendant that contains the input. The FP probability amounts to the probability that a leaf node is set to true. The tree topology of Alexiou et al. [2013] is encoded independently from the bit values, and uses two bit arrays to represent the tree structure: a bit array encoding the binary tree topology via a breath-first order and a bit array representing the true or false state of all leaf vertices. In our work, the queried filters and the links form a subtree with varying degrees. Moreover, the FP probability of a query amounts to the product of those of all queried filters. Further, we represent the tree structure and the bit values of all filters in the same bit array, avoiding the overhead to index the tree topology.

Bloofi [Crainiceanu and Lemire 2015] summarizes a large number of geo-distributed BFs using a hierarchical structure. The leaves of Bloofi correspond to the BFs that represent a set of items that are created and maintained independently by different owners. The ancestor node of the leaves amounts to the union of these leaves based on the bitwise OR operation. The bitwise OR operation yields a BF having the same size and same set of hash functions with these leaves and preserves the bits that are set to one in these leaves. As a result, an FP event at the leaves is preserved in their ancestor filters and a query returns an FP event as long as any of Bloofi's leaves incurs an FP event. BloomTree, on the other hand, decouples the FP events across levels. Further, Crainiceanu and Lemire [2015] propose a Flat-Bloofi structure to exploit the bit-level parallelism in the processor. A Flat-Bloofi array organizes 64 BFs into a packed-integer array, in which each 64-bit integer corresponds to the first bit of 64 BFs. For L BFs, Crainiceanu and Lemire [2015] need to create $\lceil \frac{L}{64} \rceil$ Flat-Bloofi arrays. Consequently, when L is not divisible by 64, a fraction of bits will be padded with zeros.

The sequence BTree (SBT) [Solomon and Kingsford 2015] summarizes a number of BFs using a tree structure for sequence-based queries. The leaves in the tree represent sequencing experiments using the BF, while the internal node amounts to the union result of its descendants based on the bitwise-OR operation. We can see that the SBT and Bloofi [Crainiceanu and Lemire 2015] have a similar structure. Consequently, in the SBT, the FP events are also correlated across levels.

The LSH forest [Bawa et al. 2005] supports the similarity search using a set of prefix trees. The LSH forest organizes multiple-dimensional points into multiple binary prefix trees. In each prefix tree, the leaf represents a point in the metric space. Each point is assigned a label that is created using a number of independent hash functions. The label of each leaf determines the logical path from it to the root. While in the BloomTree,

the querying or insertion process selects layerwise descendants based on the hashing locations of the item.

3. STANDARD BLOOM FILTER

3.1. Introduction

We now present the background on the Standard Bloom filter and show how to determine its optimal configuration. Let an *item* in a *set* be represented by an item (e.g., a 160-bit hash string) from a universe $U = [0, u)$. The Bloom filter [Bloom 1970] supports approximate membership queries with a compact data structure, by testing whether an item is hashed to a bit array. A *standard Bloom filter* (*SBF*) *BF* holds these items with a bit array I of m bits. Further, we assume that there exist a group of k independently random hash functions $h_1, \ldots, h_k$ with the range $1, \ldots, m$. The bit array is initially set to zeros. For inserting an item y into the filter, we first obtain k indexes in the bit array I by hashing the key y with these k hash functions; then, we set the bits indexed by these k bit locations to one. For testing whether a key z is stored in the filter, we calculate k indexes by hashing z with the aforementioned k hash functions similar to the insertion, and return "The item z is stored in the filter" if all k bits are set to one in the bit array.

3.2. False-Positive Probability

3.2.1. Membership Query. BFs allow for *false-positive* events, that is, returning true to a membership query for an item that is not in the set. The false positives are due to hash collisions in which the hash positions of a key that is not in the set have all been set to 1 by other keys hashed into the BF.

False-Positive Probability. Researchers have developed balls-and-bins-based formulas for tuning the parameters of a BF [Broder and Mitzenmacher 2003; Tarkoma et al. 2012]. Suppose that we insert n items into the BF; the probability that a bit is one approximates

$$1 - (1 - 1/m)^{nk} \approx 1 - e^{-nk/m} \tag{1}$$

The probability P_b that k bit locations that are computed by hashing a given item k times are simultaneously one can be approximated by

$$P_b \approx (1 - e^{-nk/m})^k \tag{2}$$

Furthermore, for convenience, let a constant $\rho = m/n$. Then, we have $P_b = (1 - e^{-k/\rho})^k$.

Broder and Mitzenmacher [2003] show that the optimal number k of hash functions that minimizes P_b is given by

$$k = (\ln 2)\frac{m}{n} = \rho \ln 2. \tag{3}$$

Then, we calculate the minimum FP probability P_b as

$$(0.5)^k \approx 0.6185^{\rho}. \tag{4}$$

Since P_b can be calculated without actually hashing items into the BF, P_b is also called the *a priori* FP probability [Hao et al. 2007]. Several studies independently reveal a large mismatch between the *a priori* formulas and the empirical FP probabilities, especially when the length of the BF is small compared to the number of inserted items [Bose et al. 2008; Christensen et al. 2010]. In fact, P_b only gives a strict lower bound for the real FP probability. Unfortunately, the time complexity of the derived correct formulas is too high to be practically applied to optimize the parameters.

Posterior False-Positive Probability. A simpler, yet powerful, metric is the *posterior FP probability* [Hao et al. 2007]. Let the *filled factor* f of the bit array I denote the

percentage of bits set to one. Given an item z randomly sampled from the universe of items, each bit location computed by hashing the item z with one of k hash functions is uniformly distributed in the whole bit array. As a result, any bit out of k hash locations for item z is set to one with probability f. Accordingly, k bits are simultaneously set to one with a probability

$$P_a = f^k. \tag{5}$$

P_a is called the posterior FP probability, since the filled factor f is known only after we insert items into the BF [Hao et al. 2007; Christensen et al. 2010; Bose et al. 2008]. Recent work [Hao et al. 2007; Christensen et al. 2010; Bose et al. 2008] confirms the correctness of this metric for characterizing the accuracy of the BF. Therefore, P_a is often used to empirically quantify the BF's accuracy.

In this article, *we use the posterior FP probability to quantify the accuracy of BloomTree*, while the *a priori* FP probability P_b incurs large approximation errors, as shown in the Appendix.

3.2.2. Set-Intersection Query. Let S_A and S_B denote two sets of items stored at two remote users A and B. The set intersection S_{AB} of two sets S_A and S_B is defined as the intersect of two sets $S_A \cap S_B$ such that, for $s \in S_{AB}$, $s \in S_A$ and $s \in S_B$ both hold. The intersection of Bloom filters (IBF) are commonly used to perform the *join* operations of multiple databases [Fu and Wang 2012; Fu et al. 2014].

The IBF amounts to the bitwise "AND" results of two bit arrays of the same length. Each bit of the IBF is computed as the bit-wise "AND" value at the same locations in BFs $SBF(A)$ or $SBF(B)$. Therefore, the number of ones in the IBF are usually smaller than that in $SBF(A)$ or $SBF(B)$, that is, lower FPs.

For two filters of the same length and the same number of hash functions, the bitwise AND operation over two filters yields a new filter that preserves the bit values of items in the set intersection and approximately cancels the bits of items that are out of the set intersection. We can see that the accuracy of the intersected filter depends on the probability to cancel out the bits of items that are out of the set intersection.

3.3. Problem Model

We now formulate the problem model that we tackle in this article. We first analyze the trade-off between the FP probability and the bandwidth overhead when using compression. By how much a BF can be compressed depends on the probability $p \in [0, 1]$ that a bit in the array is zero [Mitzenmacher 2002]. When $p = 0.5$, then the entropy of the bit array is at its maximum and no compression gain can be achieved. This is the case for the SBF, when it uses the optimal number of hash functions according to Equation (3). The closer p gets to either one or zero, the higher the potential for compression. We plot the evolution of the FP probabilities and the compression efficiency of the BF as we vary the number of hash functions. We compress the bit array as suggested by Mitzenmacher [2002].

As shown in Figure 2, we see that when the SBF achieves its minimal FP probability, there are no compression gains. However, if we accept higher values for the FP probability, the compression gain can be significant. This leads us to the following observation.

Definition 3.1 (*BF Dilemma*). Choosing the optimal number of hash functions for a BF minimizes the FP probability, but fails to decrease the transmission size; while decreasing or increasing the number of hash functions as compared to the optimal number of hash functions decreases the transmission size after compression, but increases the filter's FP probability significantly.

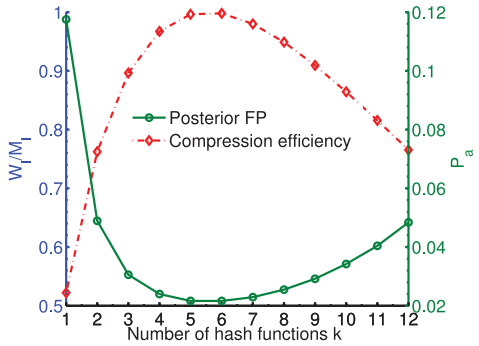


Fig. 2. The compressed efficiency W_I/M_I and the posterior FP probability P_a as a function of the number k of hash functions, where W_I denotes the transmission size after compression, M_I represents the storage size in memory. We fix the number ρ of bits per item to be 8, in which case the optimal number of hash functions amounts approximately to $\lfloor 8 \ln 2 \rfloor = 5$ by Equation (3).

This dilemma shows the inherent limitations of optimizing the trade-off between the bandwidth and the FP probability for SBF. To overcome this dilemma, our key insight is the following: *if we fix the number k of hash functions, the FP probability is determined by the percentage of ones, that is, filled factor, in the bit array. Minimizing the filled factor requires us to map items into bits that have already been set to one*. As a result, to minimize the FP probability, we must increase the *locality* of mapping items to the bit array, so that the hashing locations of new items are biased toward bit locations already set to one. Unfortunately, biasing the mapping is difficult for the SBF that maps items to a flat-structured bit array, since every hash function provides only uniform-random numbers.

To overcome this problem while keeping the simplicity of the BF, we must transform the flat structure of the filter to a hierarchical tree that maps items to proximity regions in the bit array so that the filled factors of many proximity regions increase slower than for the SBF, which leads to a lower FP probability and better compression efficiency, on average.

4. BLOOMTREE

Having presented the optimization dilemma for the BF, we next present in detail the design of BloomTree. First, we present the basic ideas of BloomTree. Then, we present the storage structure and the compression gains. Next, we show how to speed up the insertion and the querying process based on a multithread implementation. Then, we present an efficient approach to generate the hash functions. Finally, we present an intersection operation for BloomTree to accurately estimate the set intersection.

4.1. Overview

Our work aims to solve the BF dilemma when trying to optimize both the FP probability and compression efficiency. For this, we use *coordination* among small-sized BFs based on multilevel filters: an item is assumed to be in the set only when all filters that participate in the query process claim that this item is in the set. As a result, the overall FP probability is significantly decreased. We adopt a novel hierarchical organization of the BF and a layer-wise compression to trade off between the FP probability and the bandwidth consumption.

A BloomTree has d levels of BFs that are organized as a *tree*. The root of the tree is an SBF in the first level. For each bit in an SBF at level i ($i < d$) that is set to one,

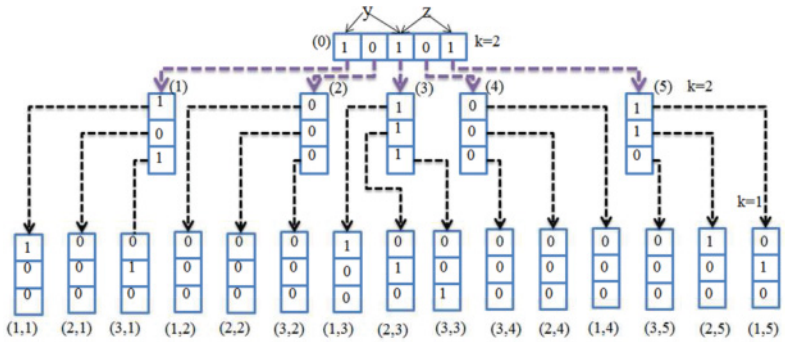


Fig. 3. A three-level BloomTree instance. The numbers of hash functions for the filters in the first, second, and third levels are 2, 2, and 1, respectively.

a *descendent* Bloom filter is appended at level $(i + 1)$. The ith level is also called the *ancestor level* for the $(i + 1)$th level. The level d is called the *leaf level*.

Generally, we represent a BloomTree as a tuple: $\{n_{\max}, d, \rho_1, k_1, m_{i,i>1}, k_{i,i>1}\}$, where $n_{\max}$ denotes the maximal number of items to be inserted, d represents the depth of the BloomTree, ρ_1 denotes the ratio between the number of bits and the number of items in the first-level filter, k_1 is the number of hash functions of the filter in the first level, $m_{i,i>1}$ is the size of the filter in the second and higher levels, and $k_{i,i>1}$ denotes the number of hash functions of the filter in the second and higher levels. We enforce each layer's filters to be of the same length for ease of computation, since varying the sizes of each filter exponentially expands the search space of the BloomTree.

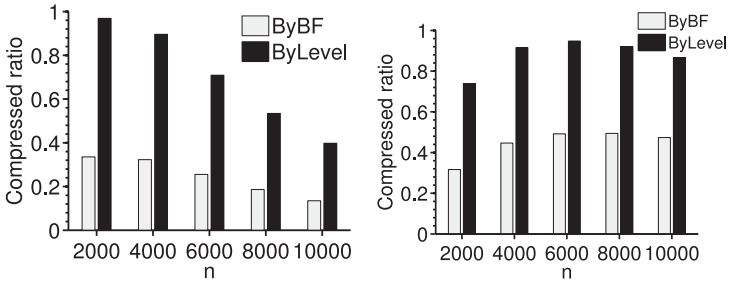
4.2. BloomTree Stored as an Array

We present a novel layout of the BloomTree as a bit array, achieving constant access time to each bit in the tree. A logical BloomTree is physically stored into a bit array. We represent a complete BloomTree in Figure 3, in that each bit of a Bloom filter in an ith level has a descendent BF at the $(i + 1)$th level, where the index $i \in [1, d - 1]$. We do not need to preserve the ancestor–descendent links, but rather compute them on-the-fly (see Section 4.2). The key idea is to store each of these BFs into the bit array in breadth-first order. As a result, the storage structure of the BloomTree is identical to that of the SBF.

We represent a BloomTree using a *bit vector* I of size $M_{BT}(I) = \sum_{i=1}^{d} \prod_{j=1}^{i} m_j$. To keep the ancestor–descendent relations among filters in the data structure without pointers, we lay out filters of a BloomTree via the breadth-first order. Specifically, the set of filters at the ath level are placed ahead of all filters in the $\geq (a + 1)$th levels. For filters at a level, the filters are placed from the left side to the right side.

The breadth-first order storage yields a simple approach to address each filter. For the first-level BF, its bit array corresponds to the subarray $I[1 : m_1]$. For the ath level (for $a > 1$), there are a total of $\prod_{i=1}^{a-1} m_i$ filters. The *leftmost* BF of the ath level starts from the $(\sum_{i=1}^{a-1} \prod_{j=1}^{i} m_j + 1)$th bit in I. The *rightmost* BF of the ath level ends at the $\sum_{i=1}^{a} \prod_{j=1}^{i} m_j$th bit in I. The bth (for $b \in [1, \prod_{i=1}^{a-1} m_i]$) BF at the ath level spans the range

$$I\left[\left(\sum_{i=1}^{a-1}\prod_{j=1}^{i} m_j + (b-1)\,m_a + 1\right) : \left(\sum_{i=1}^{a-1}\prod_{j=1}^{i} m_j + bm_a\right)\right]. \tag{6}$$



(a) A three-level BloomTree (b) A four-level BloomTree

Fig. 4. The compressed ratios of compressing each individual BF in the tree (ByBF) and compressing all filters of one level in the tree (ByLevel), which is calculated as the ratio between the array size after compression and that before compression. For (a), we set $n_{\max} = 10{,}000$, $d = 3$, $m_1 = 10{,}000$, $k_1 = 6$, $m_2 = 4$, $k_2 = 3$, $m_3 = 3$, $k_3 = 3$. For (b), we set $n_{\max} = 10{,}000$, $d = 4$, $m_1 = 10{,}000$, $k_1 = 6$, $m_2 = 4$, $k_2 = 4$, $m_3 = 4$, $k_3 = 3$, $m_4 = 5$, $k_4 = 2$.

4.3. Compressing the BloomTree

The bit array of the BloomTree can be compressed before being transmitted across the network, since the bits that are set to ones are biased toward a subset of bits. For example, from Figure 3, we see that in the second and third level, many filters have all-0 bits or all-1 bits. Therefore, in each level, the percentage of bits reversed from 0 to 1 at each vertex varies significantly. As a result, the whole level can be efficiently compressed.

There are two approaches to compress the BloomTree via the arithmetic coding algorithm [Mitzenmacher 2002]: (i) ByLevel—we compress the bit array storing all filters of one level, since we contiguously allocate the storage space of filters in the BloomTree; (ii) ByBF—we compress the bit array of each BF.

From Figure 4, we see that compressing each individual BF in the tree decreases much more required space than compressing the filters of each level altogether, since items are mapped into the BFs in a nonuniform fashion. Consequently, many BFs in one level are approximately all zeros or all ones, as shown in Figure 8(a) and 8(b).

However, in the case of ByBF, the sender needs to carefully delimit each compressed BF so that the receiver can decompress and reconstruct the original bit array. These delimiters must be transmitted to the receiver. As there are $\prod_{a=1}^{d}\prod_{i=1}^{a-1} m_i$ filters in the BloomTree, where $m_0 = 1$, we need $\prod_{a=1}^{d}\prod_{i=1}^{a-1} m_i - 1$ delimiters, which is prohibitively high. For the ByLevel approach, we need only $(d-1)$ delimiters. Therefore, we choose ByLevel compression.

The space being reduced using the level-wise compression varies with different levels. For a three-level BloomTree, the compressed ratio decreases from 0.9 to 0.4 as we increase the number n of items from 2,000 to 10,000. This is because the average filled factors of the second and the third levels reach close to one with an increasing number of items. As a result, we are able to compress more space using the arithmetic coding. For a four-level BloomTree, the compressed ratio is larger than 0.8 for $n \geq 4{,}000$; this is because the average filled factors of the third and fourth levels are still around 0.5 for $n \geq 4{,}000$. Therefore, the levelwise compression reduces the required space by only a small amount. On the other hand, the small average filled factor implies that the average FP probability of the BloomTree is quite low.

We can derive the optimal size of a levelwise compressed BloomTree using the arithmetic coding as follows:

THEOREM 4.1. *Let d denote the depth of the BloomTree. Let m_i be the length of the filter in the ith level, where $i \in [1, d]$. Let $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$ be the binary entropy function. Let $f_{a,b}$ be the filled factor of the bth filter $BF(a,b)$ in the ath level. A BloomTree of d levels can be compressed using the arithmetic coding to make its size W_{BT}:*

$$W_{BT} = \sum_{a=1}^{d} \left(\left(\prod_{i=1}^{a} m_i \right) H \left(1 - \sum_{b=1}^{\prod_{j=1}^{a-1} m_j} f_{a,b} \Big/ \prod_{j=1}^{a-1} m_j \right) \right). \tag{7}$$

We can see that, in order to maximize the compression benefit, the sum of the filled factors of all filters in the same level should be either close to zero or one.

4.4. Speeding up Insertion and Querying Operations for BloomTree

Checking for the presence of an item in a standard BF is fast, while inserting or querying an item in the BloomTree requires, on average, five times more time than for the SBF, since we need to sequentially traverse many BFs across levels (for a detailed comparison, see Section 6.4.2).

Fortunately, we can exploit the independence between different subtrees to speed up the insertion and querying processes if we insert or query the item in different subtrees concurrently. A well-known trick [Mitzenmacher and Vadhan 2008; Jeffrey and Steffan 2011] to speed up an m-bit BF with k_1 independent hash functions is that each hash function corresponds to a disjoint interval of $\frac{m}{k_1}$ consecutive bits. We can hash the item into k_1 disjoint bit arrays (partitions) of size $\frac{m}{k_1}$ *concurrently* using different hash functions. An item is assumed to be in the set when all partitions indicate that this item is in the set. The partitioned BF has asymptotically the same FP probability as the original one that shares the bits for all hash functions [Mitzenmacher and Vadhan 2008].

In our case, we can partition the BloomTree with k_1 hash functions in the first level into k_1 partitions as follows. First, we partition the root filter into k_1 disjoint intervals of $\frac{m_1}{k_1}$ consecutive bits. Second, we use each of these k_1 partitioned bit arrays as the root filter of a new BloomTree. Third, we organize the BFs in the higher layers of this new BloomTree based on the original ancestor-descendant relationships in the unpartitioned BloomTree. Consequently, we obtain k_1 independent BloomTrees, as shown in Figure 5.

An insertion or a query can be executed concurrently for each of these partitions. When the number of partitions exceeds the number of threads, we can map the partitions to different threads uniformly at random.

Insertion: The insertion in each partition is a top-down process, as shown in Section 1.2. We first hash this item into the root filter of that partition and record the corresponding hash location. Next, we select the descendent filter of the first-level hashing location and insert this item into this descendent filter at the second level. Then, we recursively insert this item into each level. For example, in Figure 5, in the second level, we insert y into (1) in the first partition and into (4) in the second partition; in the third level, we insert y into the descendants of (1), that is, (1, 1) and (3, 1), in the first partition, and the descendants of (4), that is, (3, 4) and (2, 4), in the second partition.

Query: The querying process in each partition follows the same recursive querying procedure as for the nonpartitioned BloomTree (see Section 1.2). If all partitions claim that the queried item is in the set, then the queried item is considered to be in the set; otherwise, the queried item is not in the set. For example, in order to query the item y in the two partitions, we need to traverse the set of filters (0), (1), (1, 1), and (3, 1)

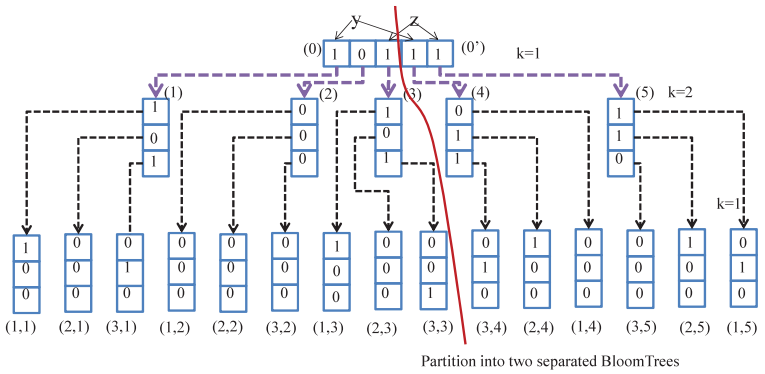


Fig. 5. The partitioned BloomTree for Figure 3. We partition the root filter in Figure 3 to two arrays. Then, we construct two subtrees whose root filters correspond to these two partitioned arrays. The ancestor–descendant links in the higher layers are the same as those in Figure 3. We insert an item to each of these partitioned subtrees concurrently.

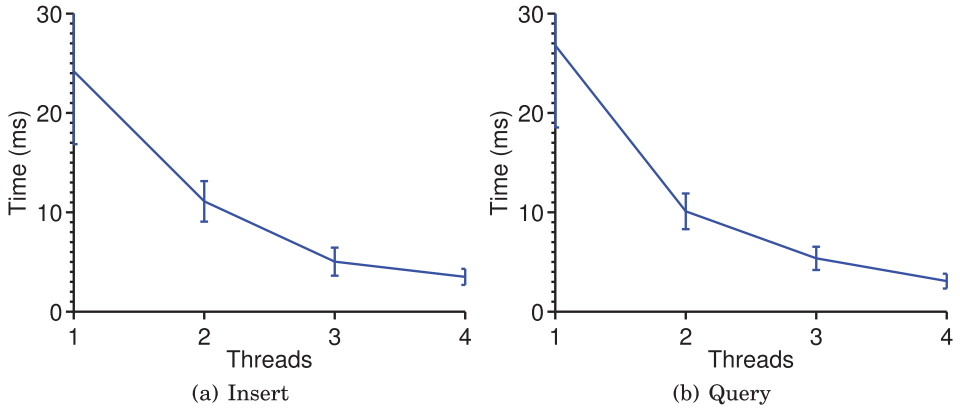


Fig. 6. The average time to insert and query 10,000 items on the BloomTree. We set $d = 3$, $\rho_1 = 1$, $m_2 = 4$, $m_3 = 3$, $k_1 = 12$, $k_2 = 3$, and $k_3 = 2$ for the BloomTree.

in the first partition, and $(0')$, (4), $(3, 4)$, and $(2, 4)$ in the second partition. We can see that all of these filters claim that y is in the set; therefore, the partitioned BloomTree returns that the item y is in the set.

To evaluate the speedup obtained by partitioning, we have performed experiments on a MacBook Pro Intel Core i7 with Quad-core and 16GB memory that allows us to vary the number of concurrent threads from one to four. In order to balance the workload among threads from one to four, we set the number k_1 of hash functions to 12 for the first level. We can see that as long as the number of threads is smaller than 5, different threads can always have the same number of BloomTree partitions.

The experiment is repeated ten times, and we plot in Figure 6 the average time required to insert and to query 10,000 items in the BloomTree. We see that the average insertion and querying time decrease proportionally with the number of threads.

Having stated the structure of the BloomTree, we next present how to implement the hash functions for each SBF in the tree. Then, we introduce the intersection of the BloomTree to support the accurate and efficient set-intersection query.

4.5. Generating Independent Hash Functions for Each Filter

Up to now, we assumed that we choose independent hash functions for each BF in each level since we need to decouple the FP events of different BFs; otherwise, querying multiple filters may not help decrease the FP probabilities. The independence means that all hash functions of the different BFs are drawn from the universe of the hash-function family, in which each hash function H_i maps an item x_i from some universe U to a number y_i that is selected uniformly at random over the entire range, for $i \in [1, k]$, where k denotes the total number of hash functions of all BFs. Specifically, let $P(H_i(x_i) = y_i)$ be the probability that the hash function H_i maps x_i to y_i. The independence implies that $P(H_1(x_1) = y_1 \bigwedge \cdots \bigwedge H_k(x_k) = y_k) = \prod_{i=1}^{k} P(H_i(x_i) = y_i)$.

As all BFs in the BloomTree need to use independent hash functions, the total number of required hash functions increases exponentially as the number d of levels increases. Consequently, the hashing evaluation must scale well.

We generate the hash functions for different Bloom filters based on the double-hashing approach [Kirsch and Mitzenmacher 2008; Hao et al. 2007], which creates a large set of hash functions using the linear combination of two random hashing functions. Kirsch and Mitzenmacher [2008] have provided a theoretical analysis of this approach. The double-hashing approach has been a popular choice to efficiently generate hash functions [Hao et al. 2007; Cha et al. 2010; Zhou et al. 2015]. Our experiments in Appendix B.4 have shown that the BloomTree using the double-hashing approach performs almost the same as that with the independent hashing functions.

Deciding how many hash functions in each filter is orthogonal to the double-hashing method. Selecting a smaller number of hash functions for a filter slows down the process to fill up the bit array with ones, but also decreases the number of filters that participate in the query process. Since the FP probability amounts to the product of the FP probabilities of all participating filters, decreasing the number of participating filters may increase the FP probability.

The double-hashing method generates *a sequence of independent hash functions* using only two CRC hash functions via a polynomial function:

$$H_{\{c_i, i \in [1,k]\}}(x) = h_1(x) + c_i \times h_2(x), \tag{8}$$

where $i \in [1, k]$, x represents an item, c_i denotes the ith *coefficient*, and h_1, h_2 represents two independent hash functions, respectively. Kirsch and Mitzenmacher [2008] proved that varying c_i leads to a set of hash values with good uniform randomness.

For each item y, we cache the two hash values $v_1 = h_1(x)$ and $v_2 = h_2(x)$ calculated by the standard hashing algorithms. Therefore, the hashing algorithms are called only twice for any item.

Let $\chi_{a,b}$ denote the set of coefficients of deriving hashing values by Equation (8) for $BF(a, b)$. We see that the set of hash values for an item x can be computed by $\{H_{\{\chi_{a,b}\}}(x)\}$. We next calculate disjoint sets $\chi_{a,b}$ of coefficients for the BF $BF(a, b)$ as follows:

—For the top-level filter $BF(1, 1)$, let its coefficients be $\chi_{1,1} = [1, k_1]$.
—For the filter $BF(a, b)$ at the ath level ($a > 1$), let the set of coefficients $\chi_{a,b}$ be

$$\chi_{a,b} = \left[\sum_{i=1}^{a-1} m_{i-1} k_i + (b-1) k_a + 1, \sum_{i=1}^{a-1} m_{i-1} k_i + b k_a \right], \tag{9}$$

where $b \leq \prod_{i=1}^{a} m_{i-1}$, $m_0 = 1$.

It is trivial to see that the set of coefficients for different filters are disjoint among each other. The overall hashing computational complexity for querying or inserting items is

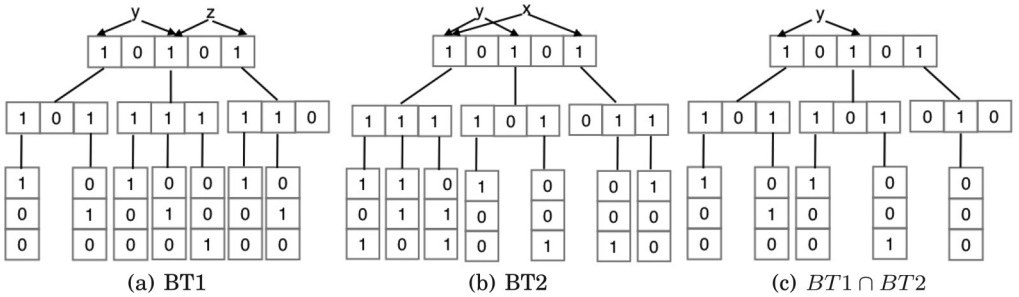


Fig. 7. The intersection of two BloomTrees $BT1$ and $BT2$. The intersection of $BT1$ and $BT2$ is represented by $BT1 \cap BT2$.

reduced to

$$2 \times \text{Hash} + O(1) \approx O(\text{Hash}), \tag{10}$$

where Hash denotes the complexity of calculating one hash value by standard hashing algorithms, such as the CRC function.

4.6. Intersection of BloomTrees

We next show how to insect BloomTrees to estimate items that are common to two sets.

Querying the set intersection with individual BloomTree instances incurs a failure probability that amounts to the FP probability of a BloomTree, while estimating the set intersection via the intersection of BloomTrees is able to reduce the failure probability after the intersection operation of two BloomTrees since there are fewer ones in the bit arrays. Intuitively, a bit location in the intersection of a BloomTree is set to one by either some item that is in the set intersection or two items that are not in the set intersection. However, the latter case rarely occurs, as items not in the set intersection are usually mapped to some different branches in the tree due to the uniform hashing across levels. Therefore, most ones in the intersection of the BloomTree are set by the set intersection. Take, for example, Figure 7: the bits that are set to one by the set intersection $\{y\}$ are preserved at the intersection BloomTree BT1 $\cap$ BT2. For the bits set by items z only in BT1 and those by x only in BT2, most of these bits are changed to zeros in BT1 $\cap$ BT2.

In order to compute the intersection of BloomTrees, we need to ensure that the same item will be inserted into the same set of filters and for each filter at the same position. To that end, for two BloomTrees, we choose the same length of the bit arrays and apply the same set of hash functions for each pair of filters at the same location in two BloomTrees.

We can estimate the set intersection via the intersection of BloomTrees similar to the intersection of standard BFs. Take two sets S_A and S_B stored in two remote nodes, which are represented by two BloomTrees $BT1$ and $BT2$ that have the same parameters, that is, the same level d, size m_i, and the number k_i of hash functions of SBFs in each level i. We see that the hash functions are the same for the same position in two BloomTrees. As a result, the same item will be inserted into the same set of filters; for each filter at the same position in the tree, the same item is always hashed into the same bit locations. As a result, we compute the intersection of BFs at the same position in two BloomTrees $BT1$ and $BT2$ and reorganize these intersected filters as a new BloomTree according to the same positions of corresponding filters, which leads to the BloomTree $BT1 \cap BT2$.

Figure 7 plots an example of the intersection of two three-level BloomTrees between Peer 1 with items y and z and Peer 2 having an item y. The all-zero SBFs are omitted

for brevity. We compute the intersection of $BT1$ (Figure 7(a)) and $BT2$ (Figure 7(b)) $BT1 \cap BT2$, as shown in Figure 7(c). We then query $BT1 \cap BT2$ with the set $\{y, z\}$ or $\{y, x\}$, respectively, and treat the items reported in the set by $BT1 \cap BT2$ as the estimated set intersection. The same approach also holds for computing the set intersection for more than two sets.

5. FP PROBABILITY OF BLOOMTREE

Having introduced the design of the BloomTree, we next analyze the FP probability of a BloomTree instance.

5.1. Derivation of the FP Probability

Thanks to the tree-style insertion process, items are spread across a small number of filters, where each filter will receive fewer items across levels accordingly. We can see that, at each level, an item is not hashed to the whole bit array of that level, but only to a subset of bits in that level. Further, some of the bits set for a given item have common prefixes through the tree, so those bits that are set to one tend to be more clustered than in the case of a flat Bloom filter. Accordingly, the set of bits set to one is biased toward a subset of bits than the whole bit array. As a result, the FP probability per filter can be controlled. Meanwhile, we still use the uniform-random hash functions for each bit array, which keeps the simplicity of Bloom filters.

An FP event occurs for a membership query *if and only if the visited BFs all report that the item is in the set*, that is, each BF queried on the tree reports an FP event for this item. BloomTree uses independent hash functions for each SBF in the BloomTree. As a result, the occurrences of FP events between different SBFs are explicitly decoupled, and we can use multiple BFs to collaboratively detect the FP events. Therefore, the FP probability of a BloomTree amounts to the product of the FP probabilities of the queried SBFs.

We use the posterior FP probability to accurately model the distribution of the BloomTree's FP probability. Our experiments show that the *a priori* FP-probability framework for SBF is not suitable for the BloomTree due to large errors (see Appendix A). Accordingly, the posterior FP probability of the BloomTree amounts to the product of the posterior FP probabilities of the queried BFs.

Example of the FP probability of a set query: For Figure 3, we first calculate the hash positions for the item y in the first level. We find that the first and third bits are both set to one, implying that the item y is hashed into the top-level filter. For two descendent filters (1) and (3) in the second level, we next test whether the item y is hashed into these two filters. Assume that the bit locations are all set to one, that is, the item is hashed into (1) and (3). We then select the descendants of the filters (1) and (3), that is, filters $(1, 1)$, $(3, 1)$, $(2, 3)$, and $(3, 3)$, and test whether the item y is hashed into them. If all visited filters report that the item y is hashed into them, then the BloomTree claims that the item y is in the set. The posterior FP probability of the query amounts to the product of the posterior FP probabilities of each of the traversed filters, that is,

$$\underbrace{0.36}_{\text{Firstlevel}} \times \underbrace{(0.44 \times 1)}_{\text{Secondlevel}} \times \underbrace{(0.33 \times 0.33 \times 0.33 \times 0.33)}_{\text{Thirdlevel}} \approx 0.0019.$$

5.2. Computing the FP Probability of BloomTree

We next define the posterior FP probability of the BloomTree. We can clearly see that the FP probability of the BloomTree depends on the set of selected BFs. An FP event occurs when an item that is not in the set is claimed to be in the set by each SBF that is queried. As discussed in Section 5.1, the FP events of different SBFs on the BloomTree

are independent of each other because of the independent hash functions; the FP probability of the BloomTree can be computed as the product of the FP probability of all participating BFs during the query process.

Let a BloomTree BT represent a set of n items. Given an incoming item $y \notin S$, the *posterior FP probability* $P_a(BT, y)$ for querying the item y can be recursively written as

$$P_a(BT, y) = P_a(R(BT)) \prod_{BT_y \in ST(R(BT).h(y), BT)} P_a(BT_y, y), \tag{11}$$

where the function $R(BT)$ returns the root filter of the BloomTree instance BT, the function $ST()$ returns a set of new BloomTrees with roots as the descendent filters of bits indexed by $R(BT).h(y)$, and the function $h()$ computes the set of indexes for an item y with the hash functions for a Bloom filter. From Equation (11), we can see that the posterior FP probability depends on the set of selected BFs at each level.

Solving Equation (11) requires recursively multiplying the posterior FP probabilities of returned filters by these two functions. The recursive function terminates at the leaf filters since these leaves no longer have descendants.

To represent the central trend of the distributions of the FP probabilities, we compute the geometric mean value $\overline{P_a}$ of the FP probabilities since it is well known that the geometric mean is more robust than the arithmetic mean, while the latter is significantly influenced by a few values that are much larger than the other values:

$$\overline{P_a}(BT) = \sqrt[N_b]{\prod_{j=1}^{N_b} P_a(BT, x_j)}, \tag{12}$$

where $x_j \in \tilde{S}$, the set $\tilde{S}$ represents a set of sampled items over the universe, and N_b denotes the number of samples ($N_b = 5000$ by default).

Further, in Appendix B, we analyze the distribution of the numbers of items per filter. We show that the number of items generally follows the Poisson distribution.

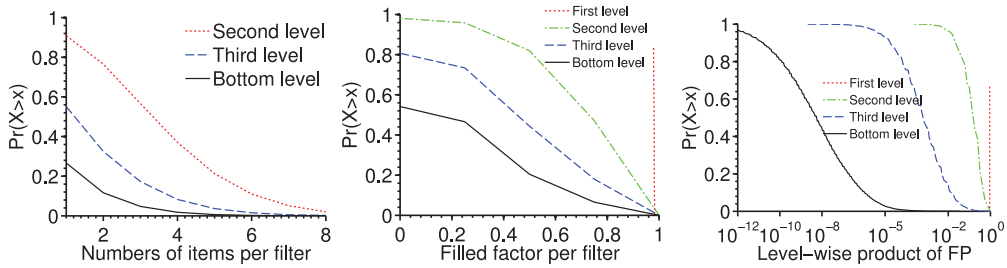
5.3. Empirical Distribution of the Filled Factors

Having shown that the posterior FP probabilities are random variables, we next present the distributions of the filled factors, that is, the percentage of ones in the bit array, across different filters in the BloomTree. The filled factor of a filter amounts to the probability of mapping an item to a bit that has been set to one by a perfectly random hash function.

We create BloomTree instances, then insert 10,000 items into each BloomTree. We insert each item into the BloomTree with the perfectly random hash functions. Then, we calculate the number of items and the filled factor of each filter. We finally compute the complementary cumulative distribution functions (CCDF) of these two metrics for filters at each level, which are plotted in Figures 8(a) and 8(b).

From Figure 8(a), we see that the tree structure decreases the number of items hashed into a filter as we move toward the leaf level. As at each level i, there exists one descendent filter at each bit of a filter $BF(x, i)$ of size m_i, the number of items mapped to a bit of $BF(x, i)$ amounts to $\frac{1}{m_i}$ times of the number of items inserted into the filter $BF(x, i)$.

In Appendix B, we provide a Poisson distribution–based approximation of the expected number of items across filters in the BloomTree, which matches well with the empirical distribution of the number of items in different filters in Appendix B.4. Unfortunately, we cannot derive the exact number of items in each filter using a closed-form function.



(a) The levelwise number of items hashing into the BFs.

(b) The levelwise filled factors of BFs.

(c) The levelwise product of the posterior FP probabilities of BFs.

Fig. 8. The CCDFs of the number of items, the filled factors of BFs, and the product of the posterior FP probabilities defined by Equation (13) at each level in the BloomTree. We set $n = n_{\max} = 10{,}000$, $d = 4$, $\rho_1 = 1$, $k_1 = 4$, $m_{i,i>1} = 4$, $k_{i,i>1} = 2$.

We see that items at each level are nonuniformly distributed. Different filters at the same level may receive a different number of items. This is due to the tree-structured mapping of items: each item is only mapped to a subtree of filters at each level depending on the ancestor filters in the tree, instead of all filters of the whole tree.

Further, since we cannot control the ratio between the number of items and the length of the Bloom filter for levels $l \geq 2$, the number of items, inserted into a BF, could be larger than a filter's length. As a result, some filters can be filled up with all ones.

We next plot the distributions of filled factors of filters at each level, as shown in Figure 8(b). We see that the filled factors at each level vary significantly because of the skewed distributions of items hashed into each filter.

5.4. Levelwise Posterior FP Probability

Having shown that filled factors are nonuniformly distributed, we next plot the posterior FP probability of the BloomTree by levels. For a BloomTree instance, we enumerate the possible paths of querying an item across different levels. Given an item y, we calculate the product of posterior FP probabilities of involved BFs at each level i:

$$P_a(BT, y, i) = \prod_{\{l_{y,i}\}} P_a(BF(l_{y,i}, i)), \tag{13}$$

where $\{l_{y,i}\}$ denotes the set of indexes of involved BFs in level i. Each product value of one query path across the tree provides one sample of the possible posterior FP probability of the BloomTree.

From Figure 8(c), we see that the product of FP probabilities decreases by orders of magnitude with increasing levels. This is because the number of BFs involved in each membership queries increases exponentially with increasing number of levels, while the posterior FP probability of each BF decreases quickly since we use the same BF size for all levels ≥ 2.

For the same configuration of the BloomTree as in Figures 8(a), 8(b), and 8(c), we now look at the coefficient of variance (CoV) of the posterior FP probabilities per level. The CoV is computed as the ratio between the standard deviation and the geometric mean. The CoV values of the levelwise products of the posterior FP probabilities of set queries are 0.95, 4.38, and 40.06 for the second, third, and fourth level, respectively. This means that the products of the FP probabilities vary much more significantly with increasing levels, as we already saw in Figure 8(c).

6. OPTIMIZING THE BLOOMTREE

Having empirically evaluated the BloomTree's FP probabilities, we next propose to optimize the BloomTree.

6.1. Challenges

Finding optimized parameters for BloomTree is challenging due to the following reasons:

—The size and the number of BFs are integers, which makes the optimization objective belong to an integer-programming problem. Minimizing the posterior FP probability while keeping the transmission bandwidth under certain bounds is a multiobjective nonlinear integer-programming problem [wikipedia.org 2016a], which is NP-hard in general.
—The number of items in each BF follows the Poisson distribution as shown in Appendix B and is not strictly independent among different BFs, due to the *negative dependence* [Dubhashi and Ranjan 1998] between the number of items of the BFs that share the same ancestor in the tree structure.
—The posterior FP probability and the compressed size do not have closed form expressions; therefore, both must be approximated using simulations.

6.2. Optimization Framework

Assuming that the hash functions map items to bit locations uniformly at random, then no matter whether an item is synthetic or real, its hashing locations will be drawn uniformly at random in each SBF in the BloomTree. As a result, the expected geometric FP probability and the expected compressed size of the BloomTree that is constructed using synthetic items should match those of the BloomTree that is built using the same number of real items from the set unknown in advance. Therefore, we next optimize the expected geometric FP probability and the expected compressed size using the BloomTree that are created using synthetic items drawn uniformly at random from the universe. In Section 8.2.5, the trace-based experiments confirm the effectiveness of the optimized parameters.

Our main objective is to *trade off the posterior FP probability and the transmission size*. Further, we would like to consume less bandwidth than the SBF with the optimal number of hash functions and the same FP probability.

Given a configuration $\vec{x}$, where $\vec{x} = (m_i, k_i)$ for $i \in [1, d]$, let $P_a(BT(\vec{x}))$ be the posterior FP probability of the BloomTree. Let $W_{BT}(BT(\vec{x}))$ be the transmission size of the BloomTree after compression. As the hash locations computed for each filter may lead to different branches in the hierarchical structure, we see that the FP probability and bandwidth are both random variables. Consequently, let $\{(BT(\vec{x}))\}$ be N independent BloomTree instances that are created by inserting $n_{\max}$ items that are drawn uniformly at random from the universe. We approximate the posterior FP probability $P_a(BT(\vec{x}))$ with the geometric-mean FP probability $\overline{P_a}(\{BT(\vec{x})\})$ of Equation (12), the bandwidth $W_{BT}(BT(\vec{x}))$ with the approximated bandwidth $\overline{W}_{BT}(\{(BT(\vec{x}))\})$ computed as $\overline{W_{BT}}(\{(BT(\vec{x}))\}) = \frac{1}{N}\sum_{j=1}^{N} W_{BT}(BT_j(\vec{x}))$.

We determine the optimal number of hash functions for SBF based on Equation (3) in order to minimize its FP probability. We compute the storage size of the standard Bloom filter that has the same FP probability as that of the BloomTree as

$$\overline{W}_{SBF}(\{(BT(\vec{x}))\}) = n_{max} \log_{0.6185} \overline{P_a}(\{(BT(\vec{x}))\}) \tag{14}$$

based on Equation (4), since no space can be compressed for an SBF with the optimal number of hash functions [Mitzenmacher 2002].

Then, we formulate a multiobjective optimization objective that minimizes the geometric FP probability $\overline{P_a}(\{(BT(\vec{x}))\})$, the compressed size $\overline{W}_{BT}(\{(BT(\vec{x}))\})$, and the ratio $\frac{\overline{W}_{BT}(\{(BT(\vec{x}))\})}{\overline{W}_{SBF}(\{(BT(\vec{x}))\})}$ between the bandwidth of the compressed BloomTree and that of the SBF. The parameter space Θ includes any combinations of integers for the BFs in each level. A solution $\vec{x}$ is said to *dominate* another solution $\vec{x}'$ ($\vec{x} \prec \vec{x}'$) if and only if

—$\overline{P_a}(\{(BT(\vec{x}))\}) < \overline{P}_a(\{(BT(\vec{x}'))\})$ and
—$\overline{W}_{BT}(\{(BT(\vec{x}))\}) < \overline{W}_{BT}(\{(BT(\vec{x}'))\})$ and
—$\frac{\overline{W}_{BT}(\{(BT(\vec{x}))\})}{\overline{W}_{SBF}(\{(BT(\vec{x}))\})} < \frac{\overline{W}_{BT}(\{(BT(\vec{x}'))\})}{\overline{W}_{SBF}(\{(BT(\vec{x}'))\})}$

hold simultaneously.

We can see that there exist sets of solutions that cannot improve any one objective of $\{\overline{P}_a(\{(BT(\vec{x}))\}), \overline{W}_{BT}(\{(BT(\vec{x}))\}), \frac{\overline{W}_{BT}(\{(BT(\vec{x}))\})}{\overline{W}_{SBF}(\{(BT(\vec{x}))\})}\}$ without making another one of these objectives increase, that is, they are not dominated by any other solutions [wikipedia.org 2016b]. Generally, we define the *Pareto-front* solutions Θ as

$$\Theta = \{\vec{X} \in X : \{\vec{X}' \in X : \vec{X}' \prec \vec{X}, \vec{X}' \neq \vec{X}\} = \emptyset\}. \tag{15}$$

6.3. Algorithm

Having formulated the multiobjective optimization framework, we introduce two methods to find the optimal parameters. The first approach is based on the exhaustive search method that enumerates the entire parameter space. The second approach is based on the Genetic algorithm [Goldberg 1989] that evolves toward the optimal parameters.

6.3.1. Exhaustive Search. In order to find all Pareto-front solutions, a simple approach is to exhaustively evaluate the parameters in the search space. We set the total number of rounds to the upper bound of the search space. At each round, we perform the following steps:

—Select a vector $\vec{x}$ of parameters for the BloomTree that has not been evaluated.
—Create a number N ($N = 500$ by default) of random BloomTree instances using synthetic items that are drawn uniformly at random from the universe.
—Calculate the averaged geometric posterior FP probability $\overline{P_a}(\{(BT(\vec{x}))\})$ of the BloomTree instances.
—Calculate the average compressed size $\overline{W}_{BT}(\{(BT(\vec{x}))\})$ of the BloomTree instances.
—Compute the ratio between the average compressed size $\overline{W}_{BT}(\{(BT(\vec{x}))\})$ and the storage size of the SBF with the optimized number of hash functions and the FP probability being $\overline{P_a}(\{(BT(\vec{x}))\})$.
—Record $\overline{P_a}(\{(BT(\vec{x}))\})$, $\overline{W}_{BT}(\{(BT(\vec{x}))\})$, and $\frac{\overline{W}_{BT}(\{(BT(\vec{x}))\})}{\overline{W}_{SBF}(\{(BT(\vec{x}))\})}$ and the corresponding parameter vector $\vec{x}$ into a hash table.
—If all parameter combinations have been considered, calculate all Pareto-front points using the records stored in the hash table and stop.

As the parameter space theoretically consists of all integers, it is practically impossible to enumerate all parameter combinations. Based on the sensitivity evaluation of the parameters in Section 7.7, we can prune the parameter space as follows:

—C1: For the first level, we limit the number of hash functions to $k_1 \in [2, 8]$, while the ratio ρ_1 between the size of the first-level filter and the maximum number $n_{\max}$ of items is given as the input parameter.
—C2: For the second and higher levels, we limit the length m_i of the BF to $m_i \in [2, 8]$ and the number k_i of hash functions to $k_i \in [1, 4]$ for $i \in [2, d]$.

We can see that the recommended parameters in Section 7.7 are subsets of the intervals bounded by the constraints C1 and C2.

6.3.2. Genetic Algorithm. Having stated the exhaustive search process, we present a Genetic algorithm [Goldberg 1989]-based approach that maintains a population of candidate BloomTree parameters and selects more fit candidate parameters stochastically in multiple rounds.

We represent the candidate (called genetic representation in the Genetic algorithm terminology) as a vector of double-precision numbers $\vec{X} = [m_2, \cdots m_d, k_1, k_2, \cdots k_d]$. For a two-level BloomTree, there are three variables in $\vec{X}$, while for a d-level BloomTree, there are $2d - 1$ variables. We limit the range of each variable based on the constraints C1 and C2 stated earlier. We minimize the same multiobjective function as that in the exhaustive search approach.

We implement the Genetic algorithm using the MATLAB "gamultiobj" toolbox. We maintain at most 15 candidates in each round, while the maximum number of optimization rounds is 100. Meanwhile, the optimization also terminates if the average relative change of the best-fit objective over 50 rounds is not larger than a threshold (10^{-4} by default).

6.3.3. Evaluation. We next compare the performance of the exhaustive search method and the Genetic algorithm.

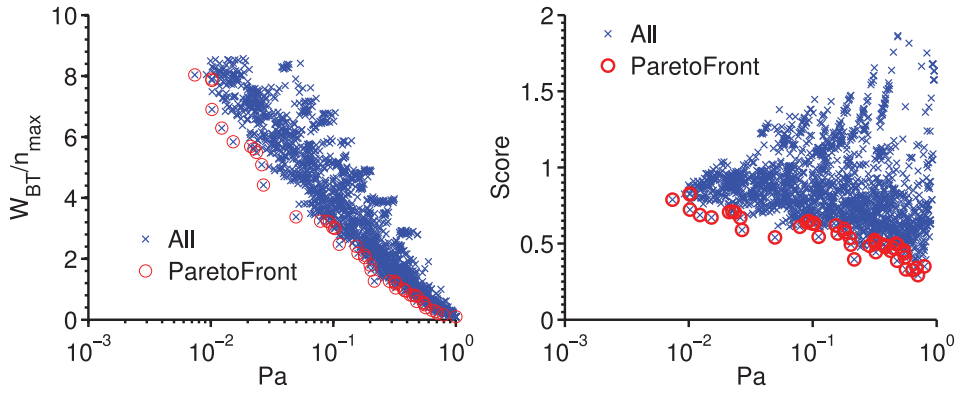
Experimental Setup: We generate synthetic items from a universe U of 64b numbers. Each item is uniformly sampled from this universe. We then construct empty BloomTree instances. Next, we hash the set of items into the BloomTree instances and calculate the posterior FP probability of the BloomTree. According to the membership query process of the BloomTree, each query must iterate over a subtree of Bloom filters across the ancestor-descendent links in a BloomTree instance. For the top level, we select k_1 bits that are set to one uniformly at random. Then, for each descendent filter of these k_1 bits, we recursively query the descendants. We terminate at the bottom level. Finally, the posterior FP probability of a BloomTree amounts to the product of posterior FP probabilities of these filters. We calculate the geometric-mean *posterior FP probability* of each BloomTree instance by Equation (12).

We choose the two-level BloomTree as a case study. We set the ratio ρ_1 between the length of the bit array of the first-level filter and the number $n_{\max}$ of items to one. We set the maximum number $n_{\max}$ of items to 10,000. We set the number n of items in the synthetic set to be $n_{\max}$ by default. The average FP probability of a BloomTree increases with increasing number n of items, since we set more bits to one at each level. As a result, when $n = n_{\max}$, the BloomTree has the largest average FP probability. Therefore, our experiments show the upper bound of the FP probability. Varying the ratio $\frac{n}{n_{\max}}$ changes the FP probabilities, but the same conclusion still holds.

Results:

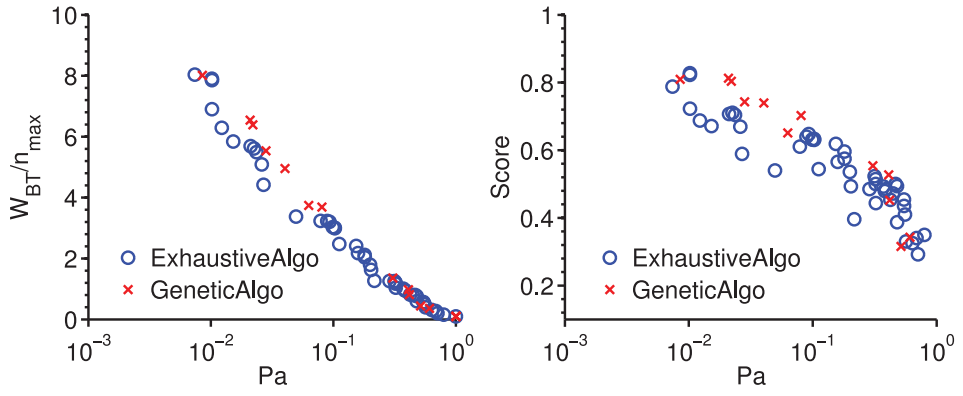
(i) *Exhaustive Search*: We first study the trade-off between the transmission bandwidth and the FP probability of the parameter space, as shown in Figure 9(a). We see that the Pareto-front solutions achieve better trade-offs than the non-Pareto-front solutions. For example, we can use less than 4b per item to keep the geometric FP probability smaller than 0.1, while the SBF with the optimal number of hash functions needs at least 5b per item in order to keep the same FP probability.

We next compare the trade-offs between the FP probability and the ratio between the BloomTree's bandwidth and the SBF's bandwidth. From Figure 9(b), we see that the Pareto-front solutions reduce the bandwidth by more than a half compared to the SBF. Consequently, BloomTree is able to obtain better trade-offs than the optimal SBF.



(a) $\overline{P_a}$ versus $\frac{W_{BT}}{n_{\max}}$ (b) $\overline{P_a}$ versus scores

Fig. 9. The whole set of solutions and the Pareto-front solutions for the exhaustive search process. Let the *score* be the ratio between the bandwidth of the BloomTree and that of the SBF with the same geometric FP probability. We set the maximal set size $n_{\max}$ to 10,000, d to 2, and ρ_1 to 1.



(a) $\overline{P_a}$ versus $\frac{W_{BT}}{n_{\max}}$ (b) $\overline{P_a}$ versus scores

Fig. 10. The Pareto-front solutions of the Genetic algorithm and the exhaustive search process. We set the maximal set size $n_{\max}$ to 10,000, d to 2, and ρ_1 to 1.

(ii) *Genetic algorithm*: Having presented the performance of the exhaustive search algorithm, we next study the trade-offs found by the Genetic algorithm. As the exhaustive search algorithm obtains the optimal Pareto-front solutions for the whole parameter space, we plot the Pareto-front solutions of the exhaustive search as the baseline performance. Figure 10 shows the trade-offs between the FP probability and the bandwidth and those between the FP probability and the ratio between the BloomTree's bandwidth and the SBF's bandwidth. The Genetic algorithm finds fewer Pareto optimal solutions than the exhaustive search because the Genetic algorithm stochastically creates candidate solutions in each round, which does not guarantee the completeness of the solutions.

Although the exhaustive search method finds more Pareto-front solutions, it does not scale well, as it needs to evaluate $7 \cdot (7 \cdot 4)^{d-1} = 7 \cdot 28^{d-1}$ parameter combinations. For example, the number of parameter combinations (iteration rounds) is 196 for $d = 2$, 5488 for $d = 3$, and 153664 for $d = 4$, while the Genetic algorithm's worst time

Table III. Some Optimal Configurations for BloomTree

d	ρ_1	m_2	m_3	m_4	m_5	k_1	k_2	k_3	k_4	k_5	$\overline{P_a}$	$\frac{W_{BT}}{W_{CBF}}$	$\frac{W_{BT}}{W_{SBF}}$	$\frac{W_{BT}}{M_{BT}}$	$\frac{M_{BT}}{n_{\max}}$	$\frac{H_{BT}}{H_{SBF}}$	$\frac{T^Q_{BT}}{T^Q_{SBF}}$
2	0.5	3	–	–	–	6	1	–	–	–	0.84	0.33	0.39	0.096	1.5	6	2.36
3	0.5	4	3	–	–	5	3	2	–	–	0.21	0.43	0.55	0.21	8.5	25	6.20
4	0.5	5	3	3	–	6	3	2	2	–	1.02×10^{-5}	0.67	0.71	0.51	33	8.25	5.93
5	0.5	4	4	2	3	6	3	3	1	2	2.01×10^{-12}	0.93	0.90	0.67	74.6	6.15	5.46
2	1	4	–	–	–	6	3	–	–	–	0.38	0.38	0.49	0.20	4.88	12	3.53
3	1	4	3	–	–	6	3	2	–	–	0.0026	0.59	0.69	0.50	16.6	7.5	4.46
4	1	4	4	5	–	6	3	3	2	–	2.55×10^{-22}	0.91	0.85	0.87	98.6	1.86	2.50
5	1	2	4	2	7	4	1	3	1	1	5.37×10^{-13}	1.35	1.30	0.55	139	1.1	1.13

Note: The size n of the itemset is set to $n_{\max} = 10^8$. $\rho_1 = \frac{m_1}{n_{\max}}$. Let M_{BT}, W_{BT} denote the storage size and the transmission size of a BloomTree instance, respectively. Let $\overline{P_a}$ be the geometric-mean posterior FP probability of the BloomTree. Let W_{CBF} and W_{SBF} be the transmission sizes of the CBF and SBF instances. Let H_{BT} and H_{SBF} be the total numbers of hash functions for BloomTree and SBF that are evaluated for a query process, respectively. Let T^Q_{BT} and T^Q_{SBF} be the average time to insert an item for BloomTree and SBF, respectively.

complexity is fixed. Therefore, we recommend the exhaustive search method for $d = 2$ and the Genetic algorithm for $d \geq 3$ levels.

6.4. Optimal BloomTree Configurations

We next report some optimized parameters. We compare the transmission size and querying speed for BloomTree, the SBF and the CBF [Mitzenmacher 2002]. For a fair comparison, we create CBF and SBF instances whose geometric-mean posterior FP probabilities amount to those of the BloomTree instances. The bandwidth of the SBF amounts to the size of its bit array, while the bandwidth sizes of CBF and BloomTree amount to their sizes after compression.

The optimized parameters are summarized in Table III. For each row in the table, we show the mean posterior FP probabilities for BloomTree, the ratios between BloomTree, SBF, and CBF using the following ratios:

— $\frac{W_{BT}}{W_{CBF}}$: the ratio of the bandwidth of BloomTree and that of the CBF.
— $\frac{W_{BT}}{W_{SBF}}$: the ratio of the bandwidth of BloomTree and that of the SBF.
— $\frac{W_{BT}}{M_{BT}}$: the ratio of the bandwidth of BloomTree and the storage size of the BloomTree.
— $\frac{H_{BT}}{H_{SBF}}$: the ratio of the number of hash functions that are evaluated for a query process for BloomTree and that for the SBF with the optimal number of hash functions.
— $\frac{T^Q_{BT}}{T^Q_{SBF}}$: the ratio of the average time to query an item in BloomTree and that in the SBF with the optimal number of hash functions.

6.4.1. False-Positive Probability versus Transmission Size. The results shown in Table III indicate that:

(1) BloomTree's average FP probabilities decrease exponentially with increasing levels. BloomTree's accuracy is determined by close-to-leaf Bloom filters. The filters in the middle levels of the BloomTree serve as the selector to choose SBFs close to the leaves. Increasing the number d of levels means that an exponential number of close-to-leaves Bloom filters are added to the membership queries. Therefore, BloomTree has exponentially decreasing FP probabilities as the depth d increases.

(2) The transmission size required for transmitting the BloomTree is around 20% to 80% lower than its storage size. For $d \leq 4$, the BloomTree reduces the transmission

size by around 20% to 50% compared to the SBF, and by 9% to 67% compared to the CBF. BloomTree can be compressed efficiently, since there are plenty of zeros in the bottom level in the BloomTree. The BFs of the middle- portions of a BT are usually filled up fast with ones, which leads to a higher compression gain. The SBF cannot be compressed since we use the optimal number of hash functions. CBF can be efficiently compressed since only two hash functions are used.

(3) Increasing ρ_1 from 0.5 to 1 decreases BloomTree's FP probabilities by several orders of magnitude, but slightly decreases the size saving of BloomTree. Increasing ρ_1 yields larger first-level bit arrays. When we fix the number $n_{\max}$ of items, the filled factor of the first-level filter decreases and the numbers of items inserted into each descendent branch decreases.

From Table III, the parameters of BloomTrees for $\rho_1 = 0.5$ and 1 are similar; therefore, the filled factors of most filters in the BloomTree will be decreased. As a result, BloomTree's FP probabilities decrease with increasing ρ_1 from 0.5 to 1.

We can see that BloomTree's saved transmission size over SBF and CBF decreases when we change ρ_1 from 0.5 to 1 since, for $\rho_1 = 1$, the BloomTree's storage increases by several times.

(4) Increasing the number d of levels decreases the transmission size saving of BloomTree From Table III, we see that the ratios $\frac{W_{BT}}{W_{CBF}}$ and $\frac{W_{BT}}{W_{SBF}}$ increase as the depth d increases. BloomTree with $d = 5$, $\rho_1 = 1$ is poorer than CBF and SBF. This is because the size of the middle-portion filters in BloomTree is only 2b to 4b; these filters are quickly filled with all ones with an increasing number of items. As a result, FP probability of the BloomTree is dominated by the bottom-level BFs, and increasing the number of layers causes more bits in the middle portion of the BloomTree to be useless in controlling the FP probabilities.

6.4.2. Speed Comparison. We next compare the access time between BloomTree and SBF. Generally, the access time consists of two components: (i) hashing time, that is, the time to compute the hashing locations; and (ii) memory access time, that is, the time to set or query the hashing bits.

—*Hashing time*: For the double-hashing mechanism, we only need two hash seeds to generate multiple hash functions. As a result, the hashing time consists of the time Λ_2 to create two hashing seeds and that of generating the hash functions. Let $\Delta \ll \Lambda_2$ be the average time to calculate one different hash value using the hash seeds. Then, we can represent the hashing time for BloomTree and SBF as follows:

$$\begin{cases} T_{BT}\,(Hash) = \Lambda_2 + H_{BT} \cdot \Delta \\ T_{SBF}\,(Hash) = \Lambda_2 + H_{SBF} \cdot \Delta \end{cases}, \tag{16}$$

where H_{BT} represents the number of hash functions in BloomTree and H_{SBF} denotes the number of hash functions in the SBF.

—*Memory access time*: As the hash function maps the elements to bit locations that are selected uniformly at random from the whole bit array, there exists little locality between different hashing values. Therefore, the memory access time can be approximated as the product of the number of unique hash functions and the average time T_{bit} to manipulate 1b. The memory access time $T_{BT}\,(Memory)$ of BloomTree with H_{BT} hash functions and that $T_{SBF}\,(Memory)$ of SBF with H_{SBF} hash functions can be represented as

$$\begin{cases} T_{SBF}\,(Memory) \approx H_{SBF} \cdot T_{bit} \\ T_{BT}\,(Memory) \approx H_{BT} \cdot T_{bit} \end{cases}. \tag{17}$$

Having presented the components of the access time, we next compute the ratio of the access time of the BloomTree and that of the SBF:

$$SpeedRatio = \frac{T_{BT}\,(Hash) + T_{BT}\,(Memory)}{T_{SBF}\,(Hash) + T_{SBF}\,(Memory)} \approx \frac{H_{BT} \cdot T_{bit} + \Lambda_2 + H_{BT} \cdot \Delta}{H_{SBF} \cdot T_{bit} + \Lambda_2 + H_{SBF} \cdot \Delta}, \tag{18}$$

which can be rewritten as

$$\frac{H_{BT} \cdot (T_{bit} + \Delta) + \Lambda_2}{H_{SBF} \cdot (T_{bit} + \Delta) + \Lambda_2} = \frac{H_{BT} + \frac{\Lambda_2}{T_{bit}+\Delta}}{H_{SBF} + \frac{\Lambda_2}{T_{bit}+\Delta}} = \frac{H_{BT} + \psi}{H_{SBF} + \psi}, \tag{19}$$

where $\psi = \frac{\Lambda_2}{(T_{bit}+\Delta)}$. We can see that ψ is a constant and is generally greater than one, since the hash function's time complexity is higher than that of accessing a bit in the memory [Broder and Mitzenmacher 2003; Putze et al. 2009]. By dividing the number H_{SBF} in the numerator and the denominator of the rightmost equation in Equation (19), we have that

$$SpeedRatio = \frac{\frac{H_{BT}}{H_{SBF}} + \frac{\psi}{H_{SBF}}}{1 + \frac{\psi}{H_{SBF}}} = 1 + \frac{\frac{H_{BT}}{H_{SBF}} - 1}{1 + \frac{\psi}{H_{SBF}}}. \tag{20}$$

We make several observations based on Equation (20):

—*Sublinear increment*: The SpeedRatio increases in a sublinear rate as we increase the ratio $\frac{H_{BT}}{H_{SBF}}$ between the number of hash functions of the BloomTree and that of the SBF, since the denominator $1 + \frac{\psi}{H_{SBF}}$ is greater than one;

—*Phase transition*: When the SBF has a few hash functions, that is, H_{SBF} is much smaller than ψ, we see that the denominator $1 + \frac{\psi}{H_{SBF}}$ dominates the SpeedRatio. As a result, the SpeedRatio increases marginally as the ratio $\frac{H_{BT}}{H_{SBF}}$ increases. In contrast, when H_{SBF} is close to ψ, we see that the numerator dominates the SpeedRatio; therefore, the SpeedRatio increases sublinearly as the ratio $\frac{H_{BT}}{H_{SBF}}$ increases.

From Table III, we see that the number of hash functions of the BloomTree is 1.1 to 25 times larger than that of the SBF. This is because the number of hash functions that are evaluated in the query process for the BloomTree increases exponentially with the depth d, while the SBF chooses a fixed number of hash functions.

Further, the query time of the BloomTree is about 1.13 to 6.2 times larger than that of the SBF, since the BloomTree evaluates more hash functions than the SBF and makes more accesses to memory than the SBF. Moreover, the SpeedRatio $\frac{T^Q_{BT}}{T^Q_{SBF}}$ experiences a phase transition when the number of levels increases from 2 or 3 to 4 or 5: When $d =$ 2 or 3, the SpeedRatio is much smaller than the ratio $\frac{H_{BT}}{H_{SBF}}$, but it is comparable to the ratio $\frac{H_{BT}}{H_{SBF}}$ for $d = 4$ or 5. This is because the number H_{SBF} of hash functions for the SBF is smaller than 10 when $d = 2$ or 3, but increases by a factor of two to seven when $d = 4$ or 5.

7. PARAMETER SENSITIVITY

We next evaluate the sensitivity of the BloomTree's accuracy with respect to the parameter settings using the same experimental setup as in Section 6.3.3.

7.1. Scalability of the BloomTree

We first evaluate the dynamics of the FP probabilities of the BloomTree as we vary the number $n = n_{\max}$ of items of the BloomTree. We fix other parameters for the BloomTree

to be constant: $d = 4$, $\rho_1 = 0.5$, $k_1 = 6$, $m_{i,i>1} = 4$, and $k_{i,i>1} = 2$. We compute the geometric-mean probability of the BloomTree at each tested number n.

From Figure 11(a), we see that, for a varying maximum number $n_{\max}$ of the items to be inserted into the BloomTree, the FP probability does not vary too much. This is expected since the expected numbers of items across filters are approximately constant.

First, the expected number of items inserted into each bit in the first level amounts to $\frac{k_1}{\rho_1}$ according to Equation (22) in Appendix B.1, where k_1 denotes the number of hash functions in the first level and ρ_1 denotes the ratio between the number n of items and the size of the first-level filter. Since we fix k_1 and ρ_1, the expected number of items in each bit of the first-level filter is constant.

Second, since we fix the size and the number of hash functions of the filters in the second and higher levels, the expected number of items inserted to the filters in the second and higher levels recursively depends on the expected number of items inserted to each bit in the first level according to Appendices B.2 and B.3. Since the latter is constant, we can see that the former is also approximately constant. As a result, the expected FP probability of the BloomTree does not vary too much with an increasing number of items.

Having confirmed that BloomTree is able to consistently control the FP probability as we vary the number of items, we next fix the number of items to 10,000 and evaluate how the posterior FP probabilities of the BloomTree vary as we change its parameters.

7.2. Depth d of BloomTree

We start by evaluating BloomTree's performance by varying its depth d. Figure 11(b) plots the posterior FP probability of BloomTree when we increase the depth d from two to seven. We see that the FP probability of BloomTree instances drops exponentially and the decreasing rate becomes even larger as the depth d increases. Although the number of SBFs participating in the membership queries increases exponentially with increasing levels, the FP probability of SBFs decreases with increasing levels, since the BFs have the same lengths from the second to the leaf levels.

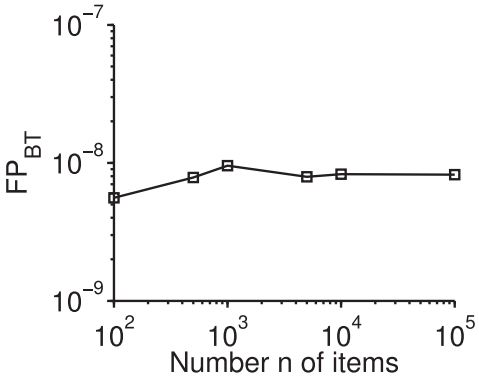
7.3. Ratio ρ_1 between the Size of the Top-Level Filter and the Number n of Items

We then characterize the BloomTree's posterior FP probability as we vary the ratio ρ_1 between the length of the first-level BF and the number of items. From Figure 11(c), we see that the BloomTree's FP probability decreases exponentially with increasing ρ_1, since larger ρ_1 decreases the number of bits set to ones for the first-level BF, which then reduces the FP probability of descendent SBFs of the top level. The FP probability for $d < 4$ decreases more quickly than that of the three-level BloomTree since the former has more SBFs than the latter for participating in the membership queries. We set the default ratio ρ_1 to be 0.5 for the following experiments.

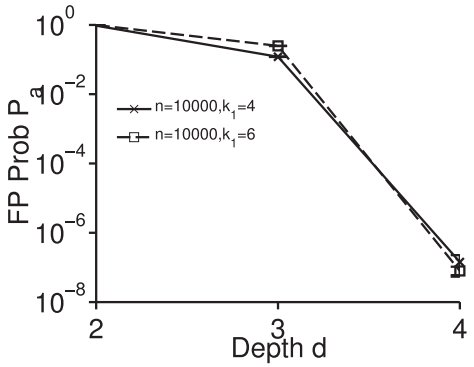
7.4. Size $m_{i,i>1}$ of SBFs in $\geq$ 2 Levels

We next test the BloomTree's FP probability as we change the length of SBFs in level 2 and higher. Setting different parameters for different levels or even different SBFs in the same level is feasible, as we discussed in Section 6.2.

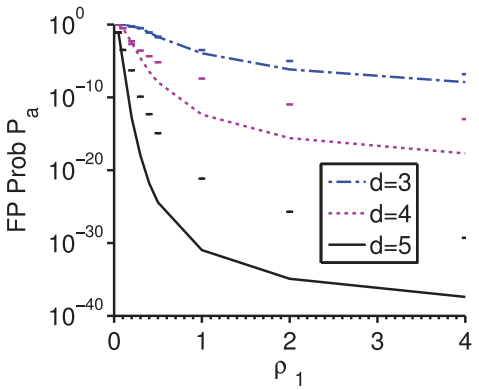
Figure 11(d) shows that the BloomTree's posterior FP probability decreases exponentially when we increase the BF size $m_{i,i>1}$ from two to ten. Although the number of SBFs participating in the membership queries does not change since we fix the number of hash functions for each BF, SBFs in $\geq$2 levels reduce their FP probabilities as we enlarge the bit arrays of these SBFs. As a result, the BloomTree's FP probability decreases quickly.



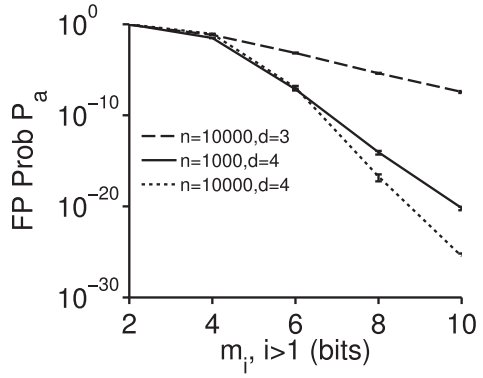
(a) Varying the number n of items.



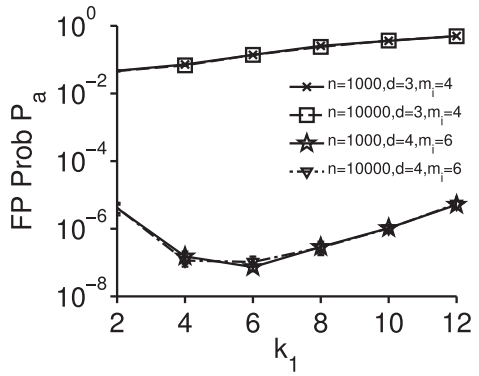
(b) Varying the depth d.



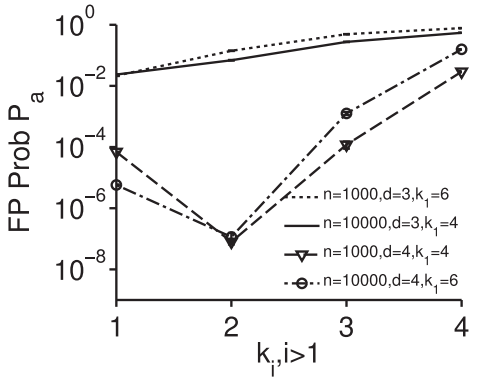
(c) Varying the ratio ρ_1.



(d) Varying the BF sizes $m_{i,i>1}$ ($i > 1$).



(e) Varying the number k_1 of hash functions.



(f) Varying the number $k_{i,i>1}$ of hash functions of the second and higher levels.

Fig. 11. The geometric-mean posterior FP probabilities of BloomTree as we vary a parameter at a time. By default, we set $n_{\max} = n$, $d = 4$, $\rho_1 = 0.5$, $m_1 = 0.5 * n_{\max}$, $k_1 = 6$, $m_{i,i>1} = 4$, and $k_{i,i>1} = 2$.

7.5. Number k_1 of Hash Functions

We then vary the number of hash functions for the first-level BF to see whether the BloomTree is able to keep low FP probabilities with a small number of hash functions. We confirm that from Figure 11(e). The four-level BloomTree reaches the lowest FP probability by using four to six hash functions, while the three-level BloomTree steadily increases the FP probability with increasing hash functions. This is because a large number k_1 of hash functions causes the SBFs of the other levels to be filled with many ones, yielding much higher FP values for these SBFs. While for the four-level BloomTree, a too small k_1 value yields a small number of SBFs responsible for answering the membership queries, since it uses too few filters for answering the queries.

7.6. Number $k_{i,i>1}$ of Hash Functions

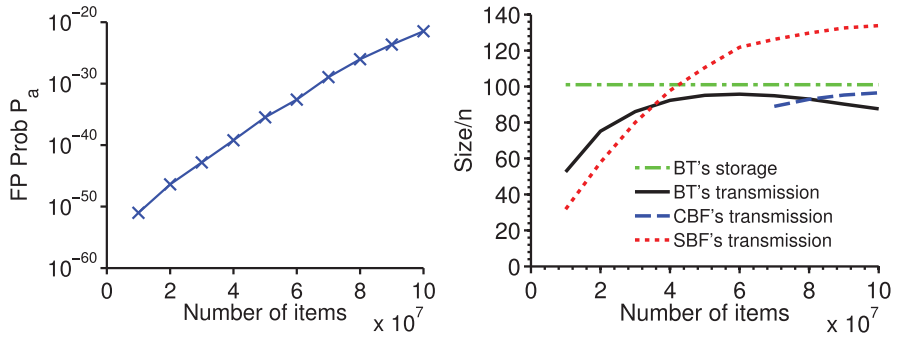
We have seen that the FP probability of the BloomTree decreases exponentially as the BF size m_i for levels $i \geq 2$ increases. We now fix the BF size $m_{i,i>1}$ and study the FP probability of the BloomTree when we vary the number k_i of hash functions for SBFs at levels $i \geq 2$.

From Figure 11(f), we see that the FP probability of the BloomTree with three levels decreases with increasing $k_{i,i>1}$, while for a BloomTree with four levels, the FP probability first drops when increasing $k_{i,i>1}$ from one to two and then increases as in the case of the three-level BloomTree. For the three-level BloomTree, most SBFs in the second and third levels have more than 50% of the bits set to one after changing $k_{i,i>1}$ from one to two, as the BF size $m_{i,i>1}$ is only four; for the four-level BloomTree, the number of SBFs at the leaf level increases exponentially when increasing $k_{i,i>1}$ from one to two, which outweighs the FP degradation of the individual BFs.

7.7. Summary of Findings

From our experiments, we can draw the following conclusions. (i) Fixing the ratio between the number n of items and the maximum number $n_{\max}$ of items yields consistent FP probabilities for an increasing number of items for a configuration $\{d, \rho_1, k_1, m_{i,i>1}, k_{i,i>1}\}$ (Figure 11(a)). (ii) Increasing the depth d will exponentially decrease the FP probability (Figure 11(b)). (iii) Increasing the ratio ρ_1 at the first layer always decreases the FP probability, while there exists a "sweet spot" for $\rho_1 \approx 1$ (Figure 11(c)). (iv) The number of hash functions at each level should be fine-tuned in order to obtain the local minimum of the posterior FP probability (Figures 11(e) and 11(f)). (v) The size $m_{i,i>1}$ of the SBFs at the second and higher layers should be modest. First, recall that the expected number of items inserted into each descendent filter amounts to the expected number of items inserted to the corresponding bit of its ancestor filter. Increasing the size $m_{i,i>1}$ decreases the expected number of items inserted to each BF in the second and higher layers, which exponentially decreases the overall FP probability of the BloomTree (Figure 11(d)). Second, the overall storage size increases with increasing filter size for the levels $d > 1$. For ease of analysis, let's assume that the filter for the levels $d > 1$ all have the same size, say, m. Then, the total storage size amounts to $m_1 + m_1 \sum_{i=2}^{d} m^{i-1}$, where m_1 denotes the filter size for the first level. For $m = 5$, a three-level filter is of size $31 * m_1$; a four-level filter is $146 * m_1$.

As a result, we recommend the following default configuration: (i) the number of levels $d \in \{2, 3, 4\}$; (ii) the ratio between the size of the root filter and the number of items $\rho_1 = \frac{m_1}{n} = 1$; (iii) the number k_1 of hash functions for the first layer $k_1 \in \{4, 5, 6\}$; (iv) the number $k_{i,i>1}$ of hash functions for the second and higher levels $k_{i,i>1} \in \{1, 2\}$; and (v) a modest size $m_{i,i>1}$ for the BFs at the second and higher levels $m_{i,i>1} \in \{2, 3, 4\}$.



(a) FP probability of BloomTree (b) Transmission size

Fig. 12. The FP probabilities and the transmission sizes as a function of the numbers of set items. The upper bound $n_{\max}$ of the set size is set to 10^8. We set $d = 4$, $\rho_1 = \frac{m_1}{n_{\max}} = 1$, $m_2 = 4$, $m_3 = 4$, $m_4 = 5$, $k_1 = 6$, $k_2 = 3$, $k_3 = 3$, $k_4 = 2$.

8. PERFORMANCE COMPARISON

The goal of BloomTree is to simultaneously reduce the transmission size and achieve a low FP probability. In this section, we compare the performance of BloomTree with the SBF, the CBF, and two more closely related data structures for theset query and the set-intersection query.

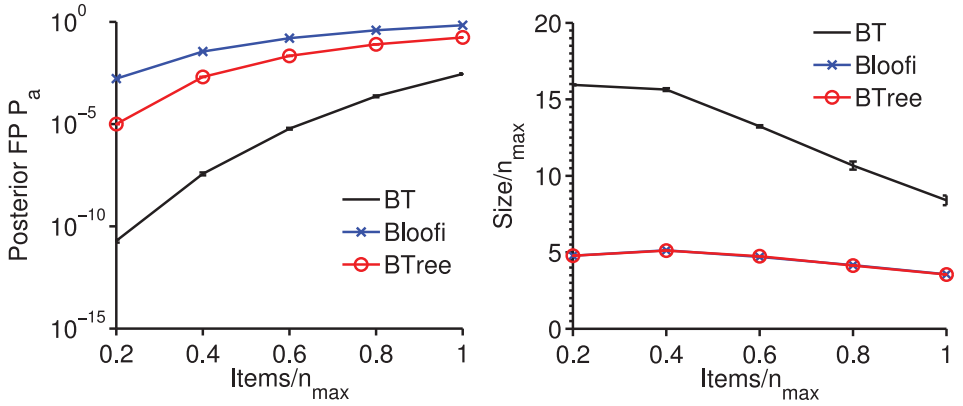
8.1. Set Query

In this section, we report the posterior FP probability of the set query and the transmission size. We compare BloomTree with SBF, CBF, and two related tree-structured BFs: the Bloofi [Crainiceanu and Lemire 2015] and the tree-structured filter (BTree) [Yoon et al. 2014].

8.1.1. Comparison with SBF and CBF. We calculate the posterior FP probability of each BloomTree instance according to Equation (11). Querying a BloomTree visits different branches that depend on the hashing locations for the querying item. Therefore, for each BloomTree instance, we perform 10,000 membership queries on the BloomTree. We then compute the geometric mean values of the posterior FP probabilities. We repeat each experiment ten times and compute the average values and their 95% confidence intervals.

We next present the growth of transmission size of BloomTree as we add items to this BloomTree. We set the BloomTree's parameters according to Table III. The upper bound $n_{\max}$ of set size is 10^8. For fair comparison, we plot the transmission sizes of CBF and SBF that have the same geometric-mean posterior FP probability with that of the BloomTree.

From Figure 12, we see that BloomTree's FP probability increases as we keep adding new items, since the percentage of bits set to one at each level monotonically increases. Further, the BloomTree's transmission size varies with increasing items because of the compression. We see that BloomTree requires the smallest transmission size compared to the CBF and the SBF as we continue to increase the number of items. For example, CBF's transmission size becomes infinity for a wide range of set sizes, since we have to increase CBF's bit array to be infinity to match the BloomTree's FP probability, as the number of hash functions of CBF is far from the optimal value. SBF is quite different from the CBF, since SBF uses the optimal number of hash functions, which helps SBF control its transmission size to be within feasible regions. SBF's transmission size increases when the set size grows to 0.5 times of the upper bound $n_{\max}$, then decreases



(a) False positive probability. (b) Compression

Fig. 13. The FP probabilities and the compression sizes for BloomTree, Bloof,i and BTree.

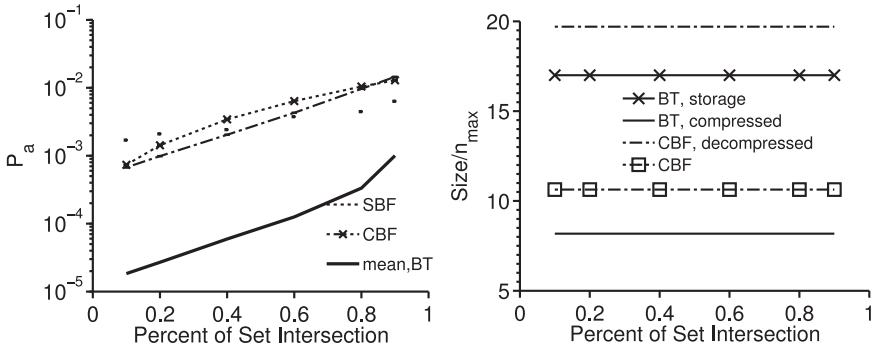
afterwards, since the size of the SBF's bit array depends on both the FP probability that decreases exponentially and the set size that keeps increasing, which naturally leads to a maximum point.

Finally, BloomTree's transmission size first increases to its storage size, but then decreases again after the set size reaches 40% of the upper bound $n_{\max}$. This is because the amount of compression at each level is inversely proportional to the binary entropy of the bit array corresponding to each level, which is maximized when the percentage p of zeros in each bit array is 0.5, but decreases when p deviates from 0.5. For the BloomTree, the percentage p of zero-valued bits first increases to 0.5, which reduces the gap between p and 0.5, but then becomes larger than 0.5, yielding a larger gap that leads to more space being compressed.

8.1.2. Comparison with Tree-Structured Filters. We now compare the BloomTree with two tree-structured filters, Bloofi [Crainiceanu and Lemire 2015] and BTree [Yoon et al. 2014]. We compute the geometric FP probabilities and the compression efficiency of each method. We enforce that all methods have the same storage sizes to ensure a fair comparison. We use a three-level BloomTree with $\rho_1 = 1$, $m_2 = 4$, $m_3 = 3$, $k_1 = 6$, $k_2 = 3$, and $k_3 = 2$. We set the recommended parameters for Bloofi [Yoon et al. 2014] and BTree [Crainiceanu and Lemire 2015]. We set the upper bound $n_{\max}$ of the number of items to 10^8.

Figure 13(a) presents the variations of the posterior FP probabilities as more items are inserted into the filters. We see that the BloomTree has the smallest FP probabilities, which is two to five orders of magnitude smaller than for the BTree and three to seven orders of magnitude smaller than for the Bloofi. The BloomTree uses a subtree of filters to collaboratively detect the FP events, while the BTree only uses a chain of filters for the detection of FPs. The BTree is better than the Bloofi since the BTree uses a chain of independent BFs to detect the FP events, while the Bloofi only detects the FP events using the leaf filters, since the internal nodes in the Bloofi have correlated FP probabilities with the leaf filters.

Figure 13(b) shows the dynamics of the ratio between the transmission size and the upper bound $n_{\max}$ as a function of the number of items inserted. We can see that the BloomTree requires two to three times more bits than the BTree and Bloofi, while the Bloofi and BTree have similar transmission sizes. This is because the BloomTree has more BFs than the Bloofi and BTree, and it concentrates the bits set to one to a small portion of the bits in each level; the filled factors of BFs in the BloomTree's



(a) Accuracy. (b) Transmission sizes.

Fig. 14. Comparing the SBF and CBF with the BloomTree as we vary the size of the set intersection.

bottom level are still far from one, while the filled factors of most BFs in each level in the Bloofi and BTree are close to one. Consequently, the total space being compressed at each level for the BloomTree is less than the Bloofi and BTree.

8.2. Set-Intersection Query

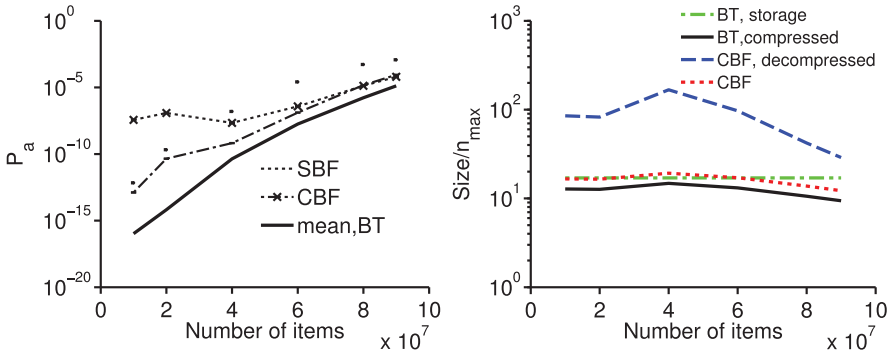
We next evaluate and compare the performance of estimating the set intersection by intersecting the BFs.

8.2.1. Experimental Setup. The intersection of BloomTrees is also a new BloomTree. Therefore, we enumerate the set of query processes across the tree. For each query, we compute the product of the posterior FP probabilities of these filters selected at each query process, which makes one sample of the posterior FP probabilities of the intersection of the BloomTrees.

We generate a pair of BloomTrees $BT(A)$ and $BT(B)$ according to the maximum number n_{max} of items that is set to 10^8. We vary the size of the set intersection S_{AB} and the sizes of two sets S_A and S_B to be general enough for diverse networking applications. We compute the minimum of the transmission sizes for $BT(A)$ and $BT(B)$ after compression. Let the minimum transmission size be W_{BT}. Then, we compute the geometric mean values of posterior FP probabilities for $BT(A)$ and $BT(B)$. We next create instances for SBF and CBF and compute the posterior FP probability for the intersection of SBF or CBF instances, respectively.

For fair comparison, we seek to set the same transmission sizes for SBFs and CBFs with those of the BloomTree. For the SBF, we have a closed-form solution by Equations (2) and (4). However, given an expected transmission size, reversely computing the storage size for the CBF has no closed-form solutions. Therefore, we have to choose the storage size for the CBF to approximate the same transmission size with that of the BloomTree. Accordingly, we also plot the storage size and transmission size of the CBF for comparison. In fact, the CBF's transmission size is always larger than that of the BloomTree. We set the BloomTree's default parameters as: d to 3, ρ_1 to 1, m_2 to 4, m_3 to 3, k_1 to 6, k_2 to 3, and k_3 to 2. Varying these parameters changes the outcomes, but the same conclusions hold consistently.

8.2.2. Varying the Set Intersection. We test whether the size of the set intersection impacts query accuracy. We set the same size for two sets S_A and S_B, and change the size of the set intersection. We see in Figure 14 that all three schemes see a degradation of their FP probability with increasing size of the set intersection, since more items in



(a) Accuracy. (b) Transmission sizes.

Fig. 15. Comparing the SBF and CBF with the BloomTree as we vary the number of items in the sets. We fix the percentage of set intersection to 0.5.

the set intersection yield a higher percentage of bits set to one in the intersected filters, which increases the overall FP probability.

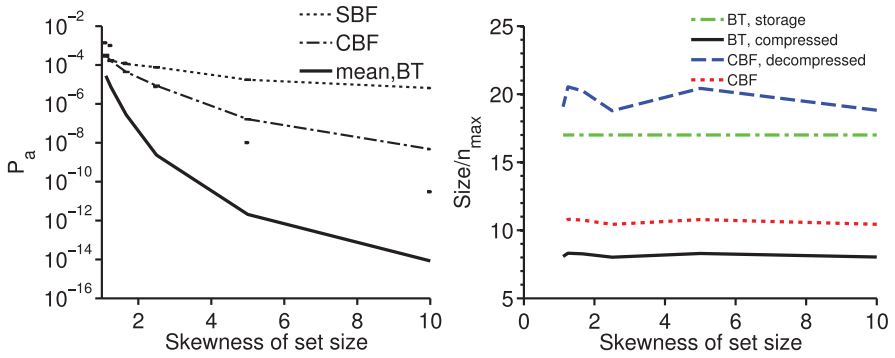
The BloomTree's FP probability is one to two orders of magnitude smaller than those of the SBF and CBF, since the SBF and CBF instances scatter items that are not in the set intersection over the whole bit array, which increases the number of bits set to one in the intersection of filters, thus degrades the FP probability. Further, we found that the 99th percentiles of FP probabilities of the BloomTree are worse than the ones for the CBF or SBF. This is due to the variance of the number of items in different BFs. The same trends hold for the following evaluation.

8.2.3. Varying the Number of Items in the Set. We next evaluate whether the number of items in the sets affects the prediction accuracy. We set the same size for two sets and fix the percentage of set intersection to 0.5 times the set size. We then vary the number of items in two sets from $5 * 10^6$ to 10^8.

In Figure 15, we see for all three schemes an increase in their FP probability with increasing number of items in the set, since more bits in the intersected filters will be set to one. However, the BloomTree has several orders of magnitude smaller geometric mean FP probabilities than SBF and CBF, since the BloomTree concentrates items that are not in the set intersection over a small percentage of filters thanks to its tree structure.

8.2.4. Varying the Skewness of Set Size. Since two sets do not necessarily have the same size, we next evaluate how a difference in the set size affects the prediction accuracy. We fix one set S_A to $n_A = 10^8$ and vary the size n_B of the smaller set S_B from 5×10^6 to 10^8. We set the size of the set intersection of S_A and S_B to 0.5 times the size of the set S_B. The skewness of two sets is calculated as $\frac{10^8}{n_B} \geq 1$.

From Figure 16, we see that increasing the skewness of two sets decreases the FP probabilities of all filters, since the intersection of filters cancels more ones as the set S_B has fewer items. The BloomTree has orders of magnitude smaller FP probabilities than SBF and CBF. This is because BloomTree concentrates items into a small subset of filters across the tree structure. Therefore, items are likely to be hashed into a more disjoint set of filters with increasing levels. Therefore, the probability that a bit is set to one in the intersection of filters decreases exponentially as the level increases. While the SBF and CBF spread each item over the whole bit array, a bit in the intersection of filters can be set to one by any item in two sets S_A and S_B. Therefore, the intersections



(a) Accuracy. (b) Transmission sizes.

Fig. 16. Comparing the SBF and CBF with the BloomTree as we vary the skewness of sets.

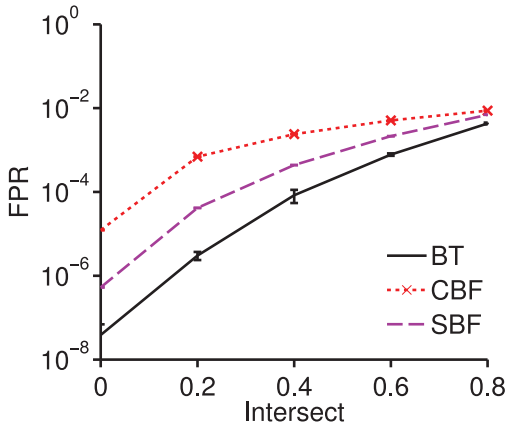


Fig. 17. The variation of the FP rates as we vary the percentage of the items in the set intersection.

of the SBF and CBF have a much larger number of bits set to one than the BloomTree, which leads to higher FP probabilities.

8.2.5. Trace-Based Experiments. We next use the real-world packet traces of a data center [Benson et al. 2010] to evaluate the performance of the BloomTree intersection. We sample two sets of two million packets uniformly at random from the dataset and vary the size of the intersection of these two sets. To evaluate estimation accuracy, we choose two million packets that are not in the set intersection from the datasets. We represent each set using a BloomTree and estimate the intersection of two sets using the BloomTree intersection approach. We define the false positive rate (FPR) of the BloomTree intersection as the percentage of packets that are incorrectly claimed to be in the set intersection. We use a three-level BloomTree with ρ_1 set to 1, m_2 to 4, m_3 to 3, k_1 to 6, k_2 to 3, and k_3 to 2. We repeat the experiments ten times and report the average FPRs.

Figure 17 plots the average FPRs with increasing percentages of the set intersection. We see that the BloomTree has the smallest FPR, which is one half to two orders of magnitude smaller than that of the SBF, and three orders of magnitude smaller than that of the CBF. The superiority of the BloomTree is expected, since the BloomTree

intersection cancels many bits that are set to ones by items that are not in the set intersection.

Also, the SBF is better than the CBF, since the SBF uses the optimal number of hash functions to minimize the FPR, while the number of hash functions for the CBF is far smaller than that in the SBF.

9. CONCLUSIONS AND FUTURE WORK

Many geo-distributed applications require efficient approximate set queries. Bloom filters are ideal for trading off query accuracy, storage space, transmission size, and query speed. We have proposed a new variant of the BF called BloomTree that uses a tree topology, which concentrates items into a small number of bit locations in the tree structure, thus increases the locality of the bits set to one. Extensive experiments show that the BloomTree obtains a nice trade-off between the FP probability and the transmission bandwidth. Further, we propose an efficient intersection method for the BloomTree to predict the items that are common to two sets, which decreases the FP probability by orders of magnitude compared to existing BFs.

The BloomTree can be significantly compressed to reduce its transmission size thanks to its tree-structured organization of small SBFs. To determine the optimal configuration of a BloomTree, we compare an exhaustive search approach with the Genetic algorithm. We find that the Genetic algorithm is preferable for $d \geq 3$.

Future Work: There are several interesting directions to be pursued. First, an open question is to optimize the BloomTree based on a closed-form theoretical analysis. Second, we do not consider deleting items from the BloomTree. Third, the SBF can be halved by performing the OR operation between the first and the second half [Yu et al. 2009]. It would be interesting to investigate whether the BloomTree can be extended to support these operations.

APPENDICES

A. MISMATCH BETWEEN THE ESTIMATED FILLED FACTOR AND THE GROUND-TRUTH FILLED FACTOR

The *a priori* FP probability estimates the filled factor of each filter using Equation (1). Therefore, we validate the suitability of using the *a priori* FP probability for the BloomTree by comparing the estimated filled factor and the ground-truth filled factor of each filter in the BloomTree.

We construct a BloomTree instance and record the numbers of inserted items and the ground-truth filled factor of each filter. Then, we calculate the estimated filled factor of each filter in the tree via $(1 - 1/m)^{nk}$ based on Equation (1), where m is the size of that filter, k is the number of hash functions, and n is the number of items inserted into that filter.

From Figure 18, we see that the ground-truth filled factors are different from the estimated values across layers. Although the bottom layer has two to three times lower errors than those of the middle-portion layers, over 90% of filters have estimation errors greater than 0.1. Further, over 60% of the middle-portion filters have errors greater than 0.4.

From our experiments, we see that the *a priori* FP probability fails to give accurate BloomTree FP probabilities. Therefore, we do not use the *a priori* FP to guide the experiments and parameter optimization, but use the posterior FP probability that uses the filled factor of the bit array to quantify the BloomTree's accuracy. The downside is that it is difficult to determine optimal parameters for the BloomTree.

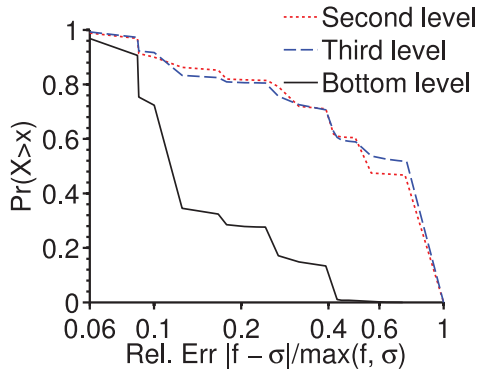


Fig. 18. The CCDFs of the levelwise relative error between the estimated filled factor σ and the ground-truth value f of each filter. We set $n = 10{,}000$, $d = 4$, $\rho_1 = 1$, $k_1 = 4$, $m_{i,i>1} = 4$, $k_{i,i>1} = 2$.

Next, we analyze the probability distribution of the numbers of items of each filter in a BloomTree.

B. DISTRIBUTION OF NUMBERS OF ITEMS ACROSS LEVELS

Suppose that we insert n items into the BloomTree. We can see that we insert n items into the first-level BF. We next derive the number of items inserted into each BF in the levels ≥ 2.

B.1. Number of Items Hashed in Each Bit in the First Level

Since each bit of the filter in the top level corresponds to a descendent filter, the number of items inserted into a bit in the first level amounts to the number of items inserted into the corresponding descendent SBF in the second level. Therefore, we first compute the number Y_i^1 of items inserted into the ith bit in the first-level BF for $i \in [1, m_1]$.

Let $m_1 = \rho_1 n_{\max}$ be the length of the first-level BF. Since we choose k_1 hash functions for the first level, we set a total number nk_1 of bits to one; therefore, we see that $Y_i^1 \leq nk_1$. As the number n of inserted items is upper bounded by $n_{\max}$, we see that the number Y_i^1 of items that is inserted to the ith bit is upper bounded by $n_{\max}k_1$; therefore, Y_i^1 can be approximated using the Poisson distribution. The probability of $Y_i^1 = x$ is computed as

$$\Pr\left(Y_i^1 = x\right) = \frac{(\lambda)^x\, e^{-(\lambda)}}{x!} = \frac{\left(\frac{n}{n_{\max}} \cdot \frac{k_1}{\rho_1}\right)^x e^{-\left(\frac{n}{n_{\max}} \cdot \frac{k_1}{\rho_1}\right)}}{x!} \tag{21}$$

for $x \in \{1, 2, \ldots, nk_1\}$.

We can also derive the expected number of items λ hashed into a bit as

$$\lambda = \frac{nk_1}{m_1} = \frac{nk_1}{\rho_1 n_{\max}} = \frac{n}{n_{\max}} \cdot \frac{k_1}{\rho_1}. \tag{22}$$

B.2. Number of Items in the Second Level

Let Y_i^2 ($i \in [1, m_1]$) be the number of items inserted into the ith filter $BF_{2,i}$ in the second level. Since the ith filter $BF_{2,i}$ corresponds to the descendant of the ith bit of the filter in the top level, we can see that $Y_i^2 = Y_i^1$ holds. Therefore, we can represent Y_i^2 as follows, based on Equation (21). Further, the expected number of items for Y_i^2 are determined by Equation (22).

B.3. Number of Items in Higher Levels

Given the bth BF $BF_{a,b}$ at the ath ($a \geq 2$) level in the BloomTree, let Y_b^a denote the number of items inserted into the BF $BF_{a,b}$. We next compute the number of items hashed into the descendants of $BF_{a,b}$.

For an SBF $BF_{a,b}$, we represent its k_a ($a \geq 1$) descendent BFs as $\{BF_{(a+1),b_i}\}$ ($i \in [1, k_a]$) in the $(a+1)$th level. We can see that the total number of items inserted into these k_a descendent BFs satisfies that

$$\sum_{i=1}^{k_a} Y_{b_i}^{a+1} \leq Y_b^a k_a. \tag{23}$$

As a result, the number $Y_{b_i}^{a+1}$ of items of the BF $BF_{(a+1),b_i}$ is upper bounded by $Y_b^a k_a$. Since Equation (23) is recursive, we see that $Y_b^a k_a$ itself has an upper bound. Therefore, we also approximate the number of items based on the Poisson distribution.

Similar to Section B.1, the number $Y_{b_i}^{a+1}$ of items of the BF $BF_{(a+1),b_i}$ amounts to the number of items hashed into the corresponding bit in its ancestor $BF_{a,b}$. The probability of having x items in a descendent BF at the $(a+1)$th level amounts to

$$f\left(x, \lambda_{b_i}^{a+1}\right) = \Pr\left(Y_{b_i}^{a+1} = x\right) = \frac{\left(\lambda_{b_i}^{a+1}\right)^x e^{-(\lambda_{b_i}^{a+1})}}{x!} = \frac{\left(\frac{Y_b^a k_a}{m_a}\right)^x e^{-(\frac{Y_b^a k_a}{m_a})}}{x!}, \tag{24}$$

where $x \in [0, Y_b^a k_a]$.

The expectation $\lambda_{b_i}^{a+1}$ of the number of items for the BF $BF_{(a+1),b_i}$ can be computed as

$$\lambda_{b_i}^{a+1} = E\left[Y_{b_i}^{a+1}\right] = \frac{Y_b^a k_a}{m_a}. \tag{25}$$

By calculating the expectation of two sides in Equation (25), we have a recursive equation between the expectation of the expected number $\lambda_{b_i}^{a+1}$ of the items hashed to a descendant $BF_{(a+1),b_i}$ and the expected number λ_b^a of items hashed to its ancestor $BF_{a,b}$: $E[\lambda_{b_i}^{a+1}] = E[\frac{Y_b^a k_a}{m_a}] = \frac{k_a}{m_a} E[Y_b^a] = \frac{k_a}{m_a} \lambda_b^a$. Further, the variance $\text{var}(Y_{b_i}^{a+1})$ of the items of different descendent BFs is $\text{var}(Y_{b_i}^{a+1}) = E[Y_{b_i}^{a+1}] = \lambda_{b_i}^{a+1}$.

B.4. Example

Next, we use the empirical distribution of the numbers of items across filters in Figure 8(a) to test the accuracy of the Poisson distribution–based approximation of the numbers of items. Recall that $n = n_{\max} = 10{,}000$, $d = 4$, $\rho_1 = 1$, $k_1 = 4$, $m_{i,i>1} = 4$, and $k_{i,i>1} = 2$. First, for a filter in the second level, we can estimate the expected number of items based on Equation (22) and Equation (25): $\lambda_2 = \frac{n}{n_{\max}} \times \frac{k_1}{\rho_1} = 1 \times \frac{4}{1} = 4$. Second, for a filter in the third level, the expected number $\lambda_{b_i}^3$ of items can be approximated as $\lambda_{b_i}^3 = E[Y_{b_i}^3] = \frac{Y_b^2 k_2}{m_2} = \frac{Y_b^2 2}{4} = \frac{Y_b^2}{2}$. As the expectation $E[Y_b^2]$ of the number of items per filter in the second level amounts to λ_2, we see that the expectation $\lambda_{b_i}^3$ of the number of items per filter in the third level amounts to $\frac{\lambda_2}{2} = 2$, which matches Figure 8(a) quite well.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their numerous and constructive comments that helped improve the article significantly.

REFERENCES

Karolina Alexiou, Donald Kossmann, and Per-Ake Larson. 2013. Adaptive range filters for cold data: Avoiding trips to Siberia. In *Proceedings of the VLDB Endowment*. 1714–1725.

Mayank Bawa, Tyson Condie, and Prasanna Ganesan. 2005. LSH forest: Self-tuning indexes for similarity search. In *Proceedings of WWW*. 651–660.

Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of IMC*. 267–280.

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7, 422–426.

Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. 2008. On the false-positive rate of bloom filters. *Information Processing Letters* 108, 4, 210–213.

Andrei Z. Broder and Michael Mitzenmacher. 2003. Network applications of bloom filters: A survey. *Internet Mathematics* 1, 4.

Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. 2010. SplitScreen: Enabling efficient, distributed malware detection. In *Proceedings of NSDI*. 377–390.

Xu Cheng and Jiangchuan Liu. 2009. NetTube: Exploring social networks for peer-to-peer short video sharing. In *IEEE INFOCOM*. 1152–1160.

Ken Christensen, Allen Roginsky, and Miguel Jimeno. 2010. A new analysis of the false positive rate of a Bloom filter. *Information Processing Letters* 110, 21, 944–949.

Adina Crainiceanu and Daniel Lemire. 2015. Bloofi: Multidimensional Bloom filters. *Information Systems* 54, 311–324.

Dubhashi and D. Ranjan. 1998. Balls and bins: A study in negative dependence. *Random Structures and Algorithms* 13, 2, 99–124.

David Eppstein, Michael T. Goodrich, Frank Uyeda, and George Varghese. 2011. What's the difference? Efficient set reconciliation without prior context. In *Proceedings of SIGCOMM*, Vol. 41. 218–229.

Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. 1998. Computing Iceberg queries efficiently. In *Proceedings of VLDB*. 299–310.

Domenico Ficara, Stefano Giordano, Gregorio Procissi, and Fabio Vitucci. 2008. Blooming trees: Space-efficient structures for data representation. In *Proceedings of ICC*. 5828–5832.

Yongquan Fu and Yijie Wang. 2012. BCE: A privacy-preserving common-friend estimation method for distributed online social networks without cryptography. In *7th International ICST Conference on Communications and Networking in China (CHINACOM'12)*. 212–217.

Yongquan Fu, Yijie Wang, and Ernst Biersack. 2013. A general scalable and accurate decentralized level monitoring method for large-scale dynamic service provision in hybrid clouds. *Future Generation Computer Systems* 29, 5, 1235–1253.

Yongquan Fu, Yijie Wang, and Wei Peng. 2014. CommonFinder: A decentralized and privacy-preserving common-friend measurement method for the distributed online social networks. *Computer Networks* 64, 369–389.

David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Fang Hao, Murali Kodialam, and T. V. Lakshman. 2007. Building high accuracy Bloom filters using partitioned hashing. In *Proceedings of SIGMETRICS*. 277–288.

Mark C. Jeffrey and J. Gregory Steffan. 2011. Understanding Bloom filter intersection for lazy address-set disambiguation. In *Proceedings of SPAA*. 345–354.

Adam Kirsch and Michael Mitzenmacher. 2008. Less hashing, same performance: Building a better Bloom filter. *Random Structures and Algorithms* 33, 2, 187–218.

Georgia Koloniari, Nikos Ntarmos, Evaggelia Pitoura, and Dimitris Souravlias. 2011. One is enough: Distributed filtering for duplicate elimination. In *Proceedings of ACM CIKM*. 433–442.

Dan Li, Henggang Cui, Yan Hu, Yong Xia, and Xin Wang. 2011. Scalable data center multicast using multi-class Bloom filter. In *Proceedings of IEEE ICNP*. 266–275.

Steven S. Lumetta and Michael Mitzenmacher. 2007. Using the power of two choices to improve Bloom filters. *Internet Mathematics* 4, 1, 17–33.

Bruce M. Maggs and Ramesh K. Sitaraman. 2015. Algorithmic nuggets in content delivery. *SIGCOMM Computer Communication Review* 45, 3, 52–66.

Michael Mitzenmacher. 2002. Compressed Bloom filters. *IEEE/ACM Transactions on Networking* 10, 5, 604–612.

Michael Mitzenmacher and Salil Vadhan. 2008. Why simple hash functions work: Exploiting the entropy in a data stream. In *Proceedings of SODA*. 746–755.

Felix Putze, Peter Sanders, and Johannes Singler. 2009. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics* 14, 4.4 (2009).

Brad Solomon and Carl Kingsford. 2015. *Large-Scale Search of Transcriptomic Read Sets with Sequence Bloom Trees*. Technical Report. Retrieved August 25, 2016 from http://repository.cmu.edu/cgi/viewcontent.cgi?article=1001&context=cbd.

S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. 2012. Theory and practice of Bloom filters for distributed systems. *IEEE Communications Surveys Tutorials* 14, 1, 131–155.

wikipedia.org. 2016a. Integer Programming. Retrieved August 25, 2016 from https://en.wikipedia.org/wiki/Integer_programming.

wikipedia.org. 2016b. Multi-objective optimization. Retrieved August 25, 2016 from https://en.wikipedia.org/wiki/Multi-objective_optimization.

Tong Yang, Alex X. Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. 2016. A shifting Bloom filter framework for set queries. *Proceedings of the VLDB Endowment* 9, 5, 408–419.

MyungKeun Yoon, JinWoo Son, and Seon-Ho Shin. 2014. Bloom tree: A search tree based on Bloom filters for multiple-set membership testing. In *Proc. of INFOCOM*. 1429–1437.

Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proceedings of ACM CoNEXT*. 313–324.

Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. 2015. Scaling up clustered network appliances with ScaleBricks. In *Proceedings of SIGCOMM*. 241–254.

Received October 2015; revised May 2016; accepted May 2016