Bloofi: A Hierarchical Bloom Filter Index with Applications to Distributed Data Provenance

Adina Crainiceanu
United States Naval Academy
adina@usna.edu

ABSTRACT

Bloom filters are probabilistic data structures that have been successfully used for approximate membership problems in many areas of Computer Science (networking, distributed systems, databases, etc.). With the huge increase in data size and distribution of data, problems arise where a large number of Bloom filters are available, and all the Bloom filters need to be searched for potential matches. As an example, in a federated cloud environment, with hundreds of geographically distributed clouds participating in the federation, information needs to be shared by the semi-autonomous cloud providers. Each cloud provider could encode the information using Bloom filters and share the Bloom filters with a central coordinator. The problem of interest is not only whether a given object is in any of the sets represented by the Bloom filters, but which of the existing sets contain the given object. This problem cannot be solved by just constructing a Bloom filter on the union of all the sets. We propose Bloofi, a hierarchical index structure for Bloom filters that speeds-up the search process and can be efficiently constructed and maintained. We apply our index structure to the problem of determining the complete data provenance graph in a geographically distributed setting. Our theoretical and experimental results show that Bloofi provides a scalable and efficient solution for searching through a large number of Bloom filters.

Categories and Subject Descriptors: E.1 Data Structures - Trees, H.3.3 Information Search and Retrieval, H.3.5 Online Information Services - Data sharing

General Terms: Algorithms, Management, Performance.

Keywords: Bloom filter index, federated cloud, data provenance.

1. INTRODUCTION

Bloom filters [1] are used to efficiently check whether an object is likely to be in the set (match) or whether the object is definitely not in the set (no match). False positives are possible, but false negatives are not. Due to their efficiency, compact representation, and flexibility in allowing a trade-off between space and false positive probability, Bloom filters are increasingly popular in representing diverse sets of data. They are used in databases [11], distributed systems [4], web caching [9], and other network applications [2].

Cloud-I '13 August 26, 2013, Riva del Garda, Trento, Italy

As digital data increases in both size and distribution, applications generate a large number of Bloom filters, and these filters need to be searched to find the sets containing particular objects.

Our work was motivated by a highly distributed data provenance application, in which data is tracked as it is created, modified, or sent/received between the multiple sites participating in the application, each site maintaining the data in a cloud environment. For security and technical reasons, it is impractical to maintain a centralized index of all the data in the geographically distributed system. However, information needs to be correlated across the semi-independent, geographically distributed clouds. Bloom filters can be maintained by each individual site, in the "local" cloud, and shared with a central location. For each piece of data, we need to find the sites holding the data. Thus, we need to search through a large number of Bloom filters stored at the central location.

Indexing Bloom filters is different than indexing generic objects to improve search time. There is one level of indirection between the elements searched for, and the objects directly indexed by the index structure. In particular, each Bloom filter is a compact representation of an underlying set of elements. The question of interest is an *all-membership* query: given a particular element (not a Bloom filter), which underlying sets contain that element? The query subject is an element, but the objects we are indexing and searching through are Bloom filters, so what we are creating is a meta-index. The traditional index structures, such as hash indexes, B+trees, R trees etc. do not directly apply in this case.

There has been significant work in using Bloom filters in various applications, and developing variations of Bloom filters. Counting filters [9] support deletions from the Bloom filter; compressed Bloom filters [10] are used with web caching; stable Bloom filters [6] eliminate duplicates in streams, spectral Bloom filters [5] extend the applicability of Bloom filters to multi-sets. Not much work has been done when a very large number of Bloom filters need to be checked for membership.

We propose *Bloofi* (**Bloo**m **F**ilter **I**ndex), a hierarchical index structure for Bloom filters. Bloofi provides probabilistic answers to *all-membership* queries and scales to tens of thousands of Bloom filters. When the probability of false positives is low, Bloofi of order d (a tunable parameter) provides $O(d * log_d N)$ search cost, where N is the number of Bloom filters indexed. Bloofi also provides support for inserts, deletes, and updates with $O(d * log_d N)$ cost and requires $O(N)$ storage cost. Bloofi could be used whenever a large number of Bloom filters that use the same hash functions need to be checked for matches. Bloofi can also be used to support deletions from a Bloom filter when only current data from a discretely moving window needs to be indexed.

The rest of this paper is structured as follows: Section 2 introduces Bloofi, a new hierarchical index structure for Bloom filters.

Section 3 introduces the maintenance algorithms and a theoretical performance analysis. Section 4 describes the distributed data provenance application for Bloofi and Section 5 shows the experimental results. We discuss related work in Section 6 and conclude in Section 7.

2. INDEXING BLOOM FILTERS

Each Bloom filter [1] is a bit array of length m constructed by using a set of k hash functions. The empty Bloom filter has all bits 0. To add an element to the filter, each of the k hash functions maps the new element to a position in the bit array. The bit in that position is turned to 1. To check whether an element is a member of the set represented by the Bloom filter, the k hash functions are applied to the test element. If any of the resulting k positions is 0, the test element is not in the set, with probability 1. If all k positions are 1, the Bloom filter $matches$ the test element, and the test element might be in the set (it might be a true positive or a false positive). There is a trade-off between the size of the Bloom filter and the probability of false positives, $probf$, returned by it. $probf$ can be lowered by increasing the size of the Bloom filter. In the rest of the paper we assume that all the Bloom filters indexed have the same length and use the same set of hash functions.

2.1 Bloofi: A Hierarchical Bloom Filter Index

Bloofi, the Bloom filter index, is based on the following idea. We construct a tree: the leaves of the tree are the Bloom filters to be indexed, and the parent nodes are Bloom filters obtained by applying a bitwise *or* on the child nodes. This process continues until the root is reached. The index has the property that each non-leaf Bloom filter in the tree represents the union of the sets represented by the Bloom filters in the sub-tree rooted at that node. As a consequence, if an object matches a leaf-level Bloom filter, it matches all the Bloom filters in the path from that leaf to the root. Conversely, if a particular Bloom filter in Bloofi does not match an object, there is no match in the entire sub-tree rooted at that node.

Using Bloofi, a membership query starts by first querying the root Bloom filter: if it does not match the queried object, then none of the indexed sets contains the object and a negative answer is returned. If the root does match the object, the query proceeds by checking which of the child Bloom filters matches the object. The query continues down the path of Bloom filters matching the object, until the leaf level is reached. In a balanced tree, the height of Bloofi is logarithmic in the number of Bloom filters indexed, and each step in the query process goes down one level in the tree. In the best case, a query with a negative answer is answered in constant time (check the root only), and a query with a positive answer is answered in logarithmic time. However, if multiple paths in the index are followed during the query process, the query time increases. Section 3.1 introduces our heuristics for Bloofi construction such that the number of "misleading" paths in the Bloofi is reduced (similar Bloom filters are in the same sub-tree).

There are many possible implementations for Bloofi: as a tree similar with binary search trees, AVL trees, B+ trees. etc. We implement Bloofi similarly with B+ trees. We define a parameter d, $order$, and each non-leaf node maintains l child pointers. $d \leq l \leq 2d$ for all non-root nodes, and $2 \leq l \leq 2d$ for the root. Each node in Bloofi stores only one value, which is different than general search trees. For the leaves, the value is the Bloom filter to be indexed. For all non-leaf nodes, the value is obtained by applying bitwise *or* on the values of its child nodes. Throughout the paper we use the usual definitions for tree, node in a tree, root, leaf, depth of a node (the number of edges from root to the node), height of a tree (maximum depth of a node in the tree), sibling, and parent.

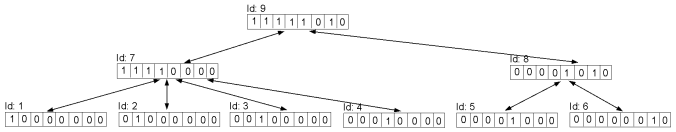

Figure 1: Bloofi Tree of Order 2

Fig 1 shows an example of a Bloofi index of order 2. Each internal node has between 2 and 4 child pointers. The leaf level of the tree contains the original Bloom filters indexed by the Bloofi index, with identifiers 1, 2, 3, 4, 5, and 6. The node identifiers are shown here for ease of presentation, but they are also used in practice during updates and deletes to identify the node that needs to be updated or deleted (see Section 3.2 and Section 3.3 for details). In the rest of the paper, we often use "node X" to refer to the node with identifier X. At the next higher level, the values are obtained by applying bitwise *or* on the values of the children nodes, so the value of the node 7 is the bitwise *or* between values of nodes 1, 2, 3, and 4. The process continues until the root is reached.

More formally, the data structure for a Bloofi index of order d is a tree such that:

- $node.nbDesc = sizeOf(node.children)\ \forall node$
- $node.nbDesc = 0$, $\forall node$ such that $node$ is leaf
- $d \leq node.nbDesc \leq 2 * d$, $\forall node$ such that ($node \neq root$ and $node$ not leaf)
- $2 \leq root.nbDesc \leq 2 * d$
- If | is bitwise OR then $node.val$ = $node.children[0].val$ | $node.children[1].val$ | ... | $node.children[node.nbDesc-1].val$, $\forall node$ such that $node$ is not leaf
- $node1.parent = node2 \Leftrightarrow node1 \in node2.children$
- $node.level$ = $height$, $\forall node$ such that $node$ is leaf

Bloofi height: Since Bloofi is a balanced tree, with each internal node having at least d child nodes, where d is the order of the tree, the height of a Bloofi index of order d is at most $\lfloor log_d N \rfloor$, where N is the number of Bloom filters to index.

2.2 Search

The search algorithm (Algorithm 1) returns the identifiers of all the leaf-level Bloom filters that match a given object in the subtree rooted at the given node. It first checks whether the current node value matches the object (line 3). If not, then none of the Bloom filters in that sub-tree match the object, so the empty set is returned (line 4). If the current node does match the object, then either it is a leaf, in which case it returns the identifier (line 8), or it is an inner node, in which case the `findMatches` function is called recursively for all of its child nodes (lines 11-14).

Example: Consider a query for object value 4 in the Bloofi tree in Figure 1. The `findMatches` function in Algorithm 1 is invoked with arguments $root$ and 4. In line 3 of Algorithm 1, the algorithm checks whether the value of the root matches 4. For simplicity of presentation, assume that there is only one hash function used by the Bloom filters, the function is *modulo 8*, and the elements in the underlying set are integers. Since $root.val[4]$ is 1, the root matches the queried object 4 and the search proceeds by invoking the `findMatches` function for each of its child nodes. The first child node, node 7, does not match the queried object, so `findMatches` for that sub-tree returns $null$. The second child node of the root, node 8, matches 4, so the search continues at the next lower level. Node 5 matches the queried object, and is a leaf,

Algorithm 1 : `findMatches`($node$,o)

```
1: //RETURN VALUE: the identifiers of leaves in the subtree
   rooted at node with Bloom filters matching the object o
2: //if node does not matches the object, return empty set, else
   check the descendants
3: if !match(node.val,o) then
4:    return ∅;
5: else
6:    //if this node is a leaf, just return the identifier
7:    if node.nbDesc == 0 then
8:       return node.id;
9:    else
10:      //if not leaf, check the descendants
11:      returnList = ∅;
12:      for i = 0; i < node.nbDesc; i++ do
13:         returnList.add(findMatches(node.children[i],o));
14:      end for
15:   end if
16: end if
17: return returnList;
```

so the `findMatches` function returns the identifier 5. Leaf 6 does not match the queried object, so that `findMatches` call returns $null$. Now, the recursive call on node 8 returns with the value $\{5\}$, and finally the call on the root returns $\{5\}$, which is the result of the query.

Search cost: The complexity of the search process is given by the number of `findMatches` invocations. In the best case, if there are no leaf-level Bloom filters matching a given object, the number of Bloom filters to be checked for matches is 1 (the root). To find a leaf-level Bloom filter that matches a query, the number of `findMatches` invocations is $O(d * log_d N)$ in the best case (one path is followed, and at each node, all children are checked to find the one that matches) and $O(N)$ in the worst case (if all the nodes in the tree are checked for matches).

3. BLOOFI MAINTENANCE

We introduce now the algorithms for inserting, deleting, and updating Bloom filters.

3.1 Insert

The algorithm for insert (Algorithm 2) finds a leaf which is "close" to the input Bloom filter in a defined metric space, and inserts the new Bloom filter next to that leaf. The intuition is that similar Bloom filters should be in the same sub-tree to improve search performance. As distance metric we use the Hamming distance. The new Bloom filter is inserted by first updating the value of the current node to *or*-it with the value of the filter to be inserted (since that node will be in the sub-tree), and then recursively calling the insert function on the child node most similar with the new value (lines 9-10). Once the most similar leaf $node$ is located (lines 33-39), a new leaf is created for the new Bloom filter (lines 35-36) and is inserted as a sibling of the $node$ by calling the `insertIntoParent` function (Algorithm 3). This function takes as parameters the new node $newEntry$, and the most similar node to it, $node$. $newEntry$ is inserted as sibling of $node$. If the number of children in the parent is still at most $2d$, the insert is complete. If an overflow occurs, the node splits (lines 9-16) and the newly created node is returned by the function. The splits could occasionally propagate up to the root level. In that case, a new root is created, and the height of the Bloofi tree increases (lines 17-26 in Algorithm 2).

Algorithm 2 : `insert`($newBloomFilter$, $node$)

```
1: //insert into the sub-tree rooted at the given node
2: //RETURN: null or pointer to new child if split occurred
3: //if node is not leaf, direct the search for the new filter place
4: if node.nbDesc > 0 then
5:    //update the value of the node to contain the new filter
6:    node.val = node.val OR newBloomFilter;
7:    //find the most similar child and insert there
8:    find child C with the closest value to newBloomFilter
9:    newSibling = insert(newBloomFilter, C);
10:   //if there was no split, just return null
11:   if newSibling == null then
12:      return null;
13:   else
14:      //there was a split; check whether a new root is needed
15:      if node.parent == null then
16:         //root was split; create a new root
17:         newRoot = new BFINode();
18:         newRoot.val = node.val OR newSibling.val;
19:         newRoot.parent = null;
20:         newRoot.children.add(node);
21:         newRoot.children.add(newSibling);
22:         root = newRoot;
23:         node.parent = newRoot;
24:         newSibling.parent = newRoot;
25:         return null;
26:      else
27:         newSibling = insertIntoParent(newSibling,
            node)
28:         return newSibling;
29:      end if//current node is root or not
30:   end if//there was a split or not
31: else
32:   //if node is leaf, need to insert into the parent
33:   //create a BFI node for newBloomFilter
34:   newLeaf = new BFINode();
35:   newLeaf.val = newBloomFilter;
36:   //insert the new leaf into the parent node
37:   newSibling = insertIntoParent(newLeaf, node);
38:   return newSibling;
39: end if//current node is leaf or not
```

Algorithm 3 : `insertIntoParent`($newEntry$, $node$)

```
1: //insert into the node's parent, after the node pointer
2: //RETURN: null or pointer to new child if split occurred
3: node.parent.children.addAfter(newEntry, node);
4: newEntry.parent = node.parent;
5: //check for overflow
6: if !node.parent.overflow then
7:    return null;
8: else
9:    //overflow, so split
10:   P = node.parent;
11:   P' = new BFINode();
12:   move last d children from P to P';
13:   update parent information for all children of P';
14:   re-compute P.val as the OR between its children values;
15:   compute P'.val as the OR between its children values;
16:   return P';
17: end if
```

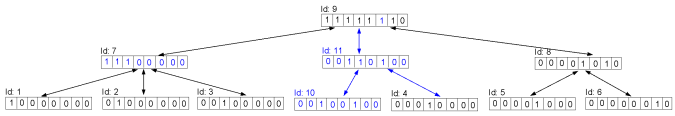

Figure 2: Bloofi Tree After Insert and Split

Example: Consider the Bloofi tree in Fig 1 and assume that we insert the Bloom filter with value "00100100". The new node is inserted as a child of node 7 (Hamming distance between the new node and nodes 7 and 8 is 4, so let's assume that node 7 is chosen as the closest node), which needs to split. The resulting Bloofi tree is shown in Fig 2.

Theorem 1 gives the cost of the insert algorithm for Bloofi. The cost metric we use to measure the performance of an operation in Bloofi is the number of Bloofi nodes accessed by that operation: either the Bloom filter value is read/modified by that operation, or the parent or children pointers in the node are read/modified.

THEOREM 1 (INSERT COST). *The number of Bloofi nodes accessed during the insert operation in Bloofi is $O(d*log_dN)$, where d is the order of the Bloofi index and N is the number of Bloom filters that are indexed.*

PROOF. The values of all nodes in the path from the root to the new leaf are updated to reflect the newly inserted Bloom filter. The height of Bloofi tree is at most $\lfloor log_dN \rfloor$ and the cost of update is constant, therefore the total cost for that update is $O(log_dN)$ (1). At each level in the tree, a search for the most similar child node is performed (line 9), and the cost of that search is $O(d)$, so total cost due to the search for the placement of the new leaf is $O(d*log_dN)$ (2). The cost of a split is $O(d)$ since there are at most $2d+1$ children for a node. In the worst case, the split propagates up to the root, and the height of the tree is $O(log_dN)$, so the worst case cost for the split operations is $O(d*log_dN)$ (3). From (1), (2), and (3), the cost of the insert operation is $O(d*log_dN)$. □

3.2 Delete

The delete algorithm (Algorithm 4) deletes the given node from the Bloofi index. When the procedure is first invoked with the leaf to be deleted as argument, the pointer to that leaf in the parent is deleted. If the parent node is not underflowing (at least d children are left), the Bloom filter values of the nodes in the path from the parent node to the root are re-computed to be the bitwise *or* of their remaining children (line 35) and the delete procedure terminates. If there is an underflow (lines 13-31), the parent node tries to redistribute its entries with a sibling. If redistribution is possible, the entries are re-distributed, parent information in the moving nodes is updated, and the Bloom filter values in the parent node, sibling node, and all the way up to the root are updated (lines 16-22). If redistribution is not possible, the parent node merges with a sibling, by giving all its entries to the sibling (lines 24-30). The Bloom filter value in the sibling node is updated, and the delete procedure is called recursively for the parent node. Occasionally, the delete propagates up to the root. If only one child remains in the root, the root is deleted and the height of the tree decreases (lines 6-10).

Example: Assume that the node with id 5 and value "00001000" is deleted from Fig 1. The resulting tree, after deletion of node 5 and redistribution between 8 and 7 is shown in Fig 3.

THEOREM 2 (DELETE COST). *The number of Bloofi nodes accessed during the delete operation in Bloofi is $O(d*log_dN)$, where d is the order of the Bloofi index and N is the number of Bloom filters that are indexed.*

PROOF. Once the reference to a node is deleted from its parent in the Bloofi tree (constant cost operation), the values of the nodes from the deleted node to the root need to be recomputed, so the total cost is $O(d*log_dN)$ (1). Occasionally, a redistribute or merge is needed, with the cost of $O(d)$. In the worst case, the merge and delete propagates up to the root, so the worst case cost for merge is $O(d*log_dN)$ (2). From (1) and (2) it follows that the cost of the delete operation is $O(d*log_dN)$. □

Algorithm 4 : `delete`($childNode$)

```
1: //find the parent node
2: parentNode = childNode.parent;
3: //remove the reference to the node from its parent
4: parentNode.children.remove(childNode);
5: //check whether the tree height needs to be reduced
6: if parentNode == root AND parentNode.nbDesc == 1
   then
7:    root = parentNode.children.get(0);
8:    root.parent = null;
9:    return null
10: end if
11: //if not, check if underflow at the parent
12: if parentNode.underflow then
13:    //underflow, so try to redistribute first
14:    sibling = sibling of parentNode;
15:    if sibling.canRedistribute then
16:       //redistribute with sibling
17:       remove some children from sibling to even out the num-
          ber of children
18:       insert new children into parentNode
19:       update .parent information for all nodes moved
20:       //update value of all nodes involved, up to the root
21:       sibling.val = OR of all children value;
22:       recomputeValueToTheRoot(parentNode);
23:    else
24:       //merge with sibling
25:       move all children from parentNode to sibling;
26:       update .parent information for all nodes moved
27:       //recompute sibling value
28:       sibling.val = OR of all childrenValue
29:       //delete the parentNode
30:       delete(parentNode);
31:    end if//merge or redistribute
32: else
33:    //no underflow
34:    //re-compute the value of all Bloom filters up to the root
35:    recomputeValueToTheRoot(parentNode);
36: end if
```

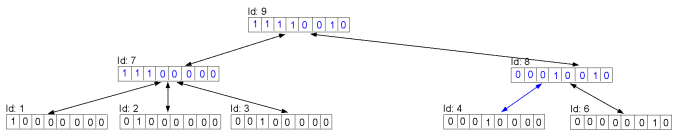

Figure 3: Bloofi Tree After Delete and Redistribute

Algorithm 5 : update($leaf$,$newBloomFilter$)

1: //update all values on the path from leaf to the root
2: $node$ = $leaf$;
3: **repeat**
4: $node.val$ = $node.val$ OR $newBloomFilter$;
5: $node$ = $node.parent$;
6: **until** $node == null$

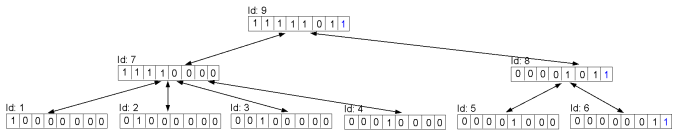

Figure 4: Bloofi Tree After Update

3.3 Update

Object insertions in the underlying set lead to updates of the Bloom filters, so we expect the update operation for Bloofi to be quite frequent. Instead of treating a Bloom filter update as a delete followed by insert, we use an "in-place" update. If the Bloofi tree becomes inefficient in routing due to updates (too many false positives during search) the Bloofi tree can be reconstructed from scratch in batch mode. Algorithm 5 shows the pseudo-code for the update algorithm. The algorithm takes as parameters the leaf node corresponding to the updated value and the new Bloom filter value for that node. All the Bloom filters in the path from the leaf to the root are updated by *or*-ing with the new value.

Example: In the Bloofi tree in Fig 1, assume that we update the value of node 6 to be "00000011". The values of all the nodes in the path from node 6 to the root are *or*-ed with "00000011" and the resulting tree is shown in Fig 4.

THEOREM 3 (UPDATE COST). *The number of Bloofi nodes accessed during the update operation in Bloofi is $O(log_d N)$, where d is the order of the Bloofi index and N is the number of Bloom filters that are indexed.*

PROOF. The number of nodes to be updated is equal to the height of the Bloofi tree + 1, and each update accesses only the current Bloom filter for that node and the new Bloom filter. The height of Bloofi is at most $\lfloor log_d N \rfloor$, therefore the update cost is $O(log_d N)$. □

3.4 Improving Pruning Efficiency

In Bloofi, each non-leaf node value is the bitwise *or* of its children values. As the total number of objects in the underlying sets indexed by Bloofi increases, the probability of false positive results returned by the Bloom filters at the higher levels in the tree increases. In the worst case, all bits in the Bloofi nodes at higher levels in the tree could be one. This leads to decreased pruning efficiency of the higher levels in the tree, as more false positive paths are followed during search. To improve upon the number of Bloom filters that need to be checked for matches during a query, we propose the following heuristic.

During the insert procedure, we do not split a node that has all the bits set to one, even if the node is full. This could stop the splitting a little too early, but it avoids creating multiple levels in the tree with all the bits set to one. Our experimental results in Section 5.2 show that the search cost is indeed improved by using this heuristic when the root level value has all bits set to one.

Alternatively, we could dynamically change the size of the Bloom filters when the false positive probability at the root reaches 1. In such a case, if the application allows, we could reconstruct the base Bloom filters to have a lower false positive probability, and reconstruct the Bloofi tree from bottom-up. The index construction time for 100,000 Bloom filters was about 15 minutes in our experiments, so periodic reconstruction of the index is a viable solution.

4. APPLICATION TO DISTRIBUTED DATA PROVENANCE

In this section we describe the distributed data provenance application that motivated our work on Bloofi.

Let us assume that a multinational corporation with hundreds of offices in geographically distributed locations (sites) around the world is interested in tracking the documents produced and used within the corporation. Each document is given a universally unique identifier (uuid) and is stored in the local repository, in a cloud environment. Documents can be sent to another location (site) or received from other locations, multiple documents can be bundled together to create new documents, which therefore are identified by new uuids, documents can be decomposed in smaller parts that become documents themselves, and so on. All these "events" that are important to the provenance of a document are recorded in the repository at the site generating the event. In our application, all events are stored as RDF triples in Rya [12], a scalable cloud triple store. The data can be modeled as a Directed Acyclic Graph, with labeled edges (event names) and nodes (document uuids). As documents travel between sites, the DAG is in fact distributed not only over the machines in the cloud environment at each site, but also over hundreds of geographically distributed locations. The data provenance problem we are interested in solving is finding all the events and document uuids that form the "provenance" path of a given uuid (all "ancestors" of a given node in the distributed graph).

Storing all the data, or even all the uuids and their location, in a centralized place is not feasible, due to the volume and speed of the documents generated globally. Moreover, local regulations might impose restrictions on where the data can be stored. However, since all the global locations belong to the same corporation, data exchange and data tracking must be made possible.

Without any information on the location of a uuid, each provenance query for a uuid must be sent to all sites. Each site can then determine the local part of the provenance path, and return it. However, the provenance might contain new uuids, so a new query needs to be sent to each site for each new uuid connected to the original uuid, until no new uuids are found. This recursive process could consume significant bandwidth and latency in a geographically distributed system.

To minimize the number of unnecessary messages sent to determine the full provenance of an object, each local site maintains a Bloom filter of all the uuids in the local system. Updates to the Bloom filter are periodically propagated to a centralized location (the headquarters for example). At the central location, a Bloofi index is constructed from the Bloom filters, and every time a provenance query for a uuid is made, the Bloofi index is used to quickly determine the sites that might store provenance information for the given uuid.

5. EXPERIMENTAL EVALUATION

We evaluate Bloofi's search performance and maintenance cost for different number and size of Bloom filters, different similarity metrics used for Bloofi construction and different underlying data distributions. We also compare Bloofi's performance with the 'baseline' case where the Bloom filters are searched linearly, without any index. We show that in most cases, Bloofi achieves loga-

rithmic search performance, with low maintenance cost, regardless of the underlying data distribution.

5.1 Experiments Setup

We implemented Bloofi in Java. For Bloom filters, we use the implementation from [13]. The experiments were run on a Dell Latitude E6510 with 2.67 GHz Intel Core i7 CPU and 4GB RAM.

As performance metrics we use: *search cost*: the number of Bloom filters checked to find the one(s) matching a queried object, averaged over 1000 searches; *search time*: the average time, in milliseconds, to find all the Bloom filters matching a queried object; *storage cost*: space required for the index, as estimated by the number of bytes for a Bloom filter, multiplied by the number of nodes in the Bloofi tree; *maintenance cost*: the average number of Bloofi nodes accessed during an insert, delete, or update operation.

We vary the following parameters: N number of Bloom filters indexed by Bloofi; d Bloofi order; *Bloom filter size*; *index construction method*: $iterative$ (insert Bloom filters one by one using the algorithm in Section 3.1) or $bulk$ (first sort all Bloom filters such that the first Bloom filter is the one closest to the empty Bloom filter, the second is the filter closest to the first Bloom filter, etc., and then construct the Bloofi tree by always inserting next to the right-most leaf); *similarity measure* used to define "closeness" during insert; and *data distribution*: $nonrandom$ (non-overlapping ranges for each Bloom filter: each Bloom filter i contains $nbEl$ integers in the range $[i * nbEl, (i+1) * nbEl)$) and $random$ (overlapping ranges for data in the Bloom filters: each Bloom filter i contains $nbEl$ random integers in a randomly assigned range).

For each experiment we vary one parameter and use the default values shown in Table 1 for the rest. The numbers reported are the averages over 5 runs.

N - Number of Bloom filters indexed	1000
d - Bloofi order	2
Construction method	iterative
Distance metric	Hamming
Data distribution	nonrandom
Bloom filter size (bits)	100,992
Number of hash functions used	7
Number of elements in each indexed Bloom filter	100

Table 1: Default Values for Parameters

5.2 Performance Results

Varying Number of Bloom Filters Indexed N: Fig 5 shows the increase in the search cost as N increases from 100 to 100,000. Note the logarithmic scale on both X and Y axes. In the "ideal" case, when exactly one path from root to the leaf is followed during search, the search cost is approximately $l * log_l N + 1$ if each non-leaf node has l children. In our experiments, the increase in search cost is logarithmic as long as the false positive probability ($probf$) at the root is less than one ($N \leq 1000$), but the the increase is higher than logarithmic after that, due to high value for $probf$ at the high levels of the tree. However, even for 100,000 filters, Bloofi still performs two orders of magnitude better than the baseline case. If the size of the Bloom filters increases, the search cost decreases to the "ideal" cost, as shown in Fig 11.

Bulk construction performs slightly better than incremental (indistinguishable from *After Updates* curve), since a global sort is performed before the bulk insert, while the incremental construction is greedy and might not lead to optimal placement. However, the cost of sorting is $O(N^2)$ in our implementation, which leads to high cost of bulk construction (that is why no numbers are shown for N=100000 for $Bulk$).

We run the same experiment by using the heuristic in Section 3.4. When there are Bloofi nodes with all bits 1, the search cost when the heuristic is used is lower than when the heuristic is not used (103.16 vs. 111.27 for $N = 10000$ and 885.66 vs. 969.80 for $N = 100000$) which shows the effectiveness of the heuristic.

To evaluate the effect of the in-place update, the Bloofi tree is built incrementally using only half of the elements in each Bloom filter. The rest of the elements are inserted then in each Bloom filter and Bloofi is updated in-place. The *After Updates* curve in Fig 5 is the search cost in the final tree. The *After Updates* and *Incremental* curves are almost identical, which shows that the in-place update maintains the search performance of the Bloofi tree.

Fig 6 shows similar trends for the search time, since the time depends on the number of Bloom filters checked. The search time also depends on the locality of nodes in memory (due to the memory hierarchy), so the difference between Bloofi and baseline case is not as big when only a few Bloom filters are checked. The difference increases as N increases.

The storage cost increases linearly with N (not shown), as the number of nodes in the Bloofi tree is proportional to N.

The maintenance cost increases logarithmically with N (Fig 7), as expected from Theorems 1, 2, and 3. The cost of insert is higher than the delete, as the place for the new node needs to be found during insert. The update cost is the lowest, as we use in-place updates, and no splits, merges, or redistributions are needed.

Varying Bloofi Order d: The search cost increases as d increases (Fig 8), since the search cost is proportional to $d * log_d N$. The search cost obtained from experiments is very close to the ideal search cost for full trees ($2d$ children per non-leaf node), which shows that our algorithms for tree construction perform very well, and not many nodes outside of the path from the root to the answer leaf are checked during search. Bulk construction performs better than incremental construction, due to the global sort. The *After Updates* and *incremental* lines are almost identical, which shows that our in-place update maintains the performance of the Bloofi tree. The same trends were obtained for the search time.

The storage cost decreases with order (Fig 9), as the number of non-leaf nodes in a Bloofi tree of order d is between $\frac{N-1}{2*d-1}$ and $\frac{N-1}{d-1}$, so higher the order, the lower the storage cost. While the storage cost is lower for higher orders, the search cost is higher (Fig 8), so there is a trade-off between search and storage cost. The storage cost for bulk construction is slightly higher than for incremental construction, because always inserting into the right-most leaf leads to skinnier, taller trees, with more nodes. The additional storage cost for Bloofi versus the baseline, is at most equal to the baseline storage cost.

The insert and delete costs ($O(d*log_d N)$) increase with d (Fig 10), while the update cost ($O(log_d N)$) decreases with d.

Varying Bloom Filter Size: Fig 11 shows the search cost variation with the Bloom filter size in a system with 100,000 total elements (1000 filters with 100 elements each). The search cost for Bloofi is always below baseline cost, and it decreases to the ideal cost as the size of the Bloom filters increases and the false positive probability of the high level nodes in Bloofi decreases. In fact, $O(d*log_d N)$ search cost is achieved as soon as $probf < 1$ for the root, even if $probf$ is very close to 1 (0.993 in our experiments for size 100992).

The search time (Fig 12) also decreases for Bloofi as $probf$ at the higher levels is reduced, while the search time for baseline case is constant. For very small Bloom filter size, the search time for

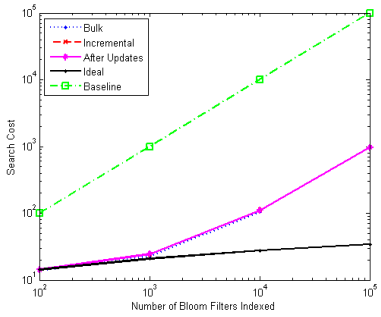

Fig 5. Search Cost vs N

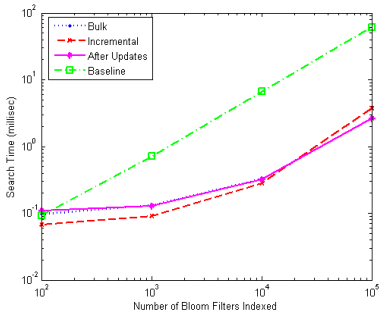

Fig 6. Search Time vs N

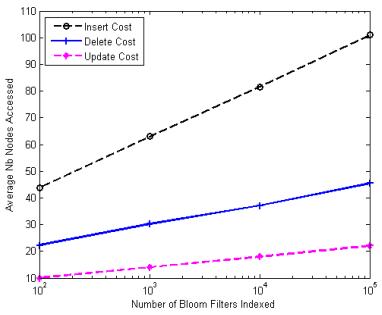

Fig 7. Maintenance Cost vs N

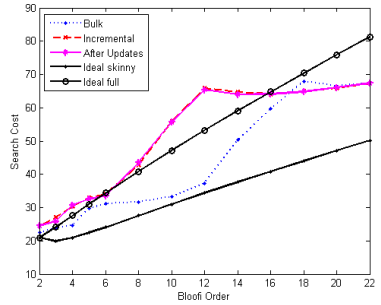

Fig 8. Search Cost vs d

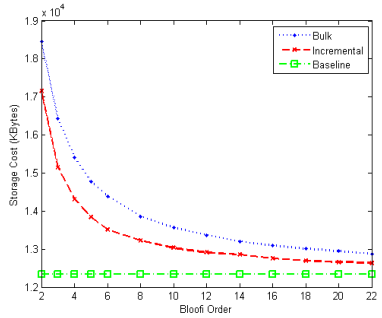

Fig 9. Storage Cost vs d

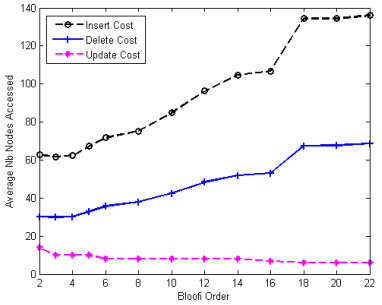

Fig 10. Maintenance Cost vs d

the baseline case is slightly lower than using Bloofi, likely due to memory locality.

Varying Similarity Metric: We use the following metrics: $Hamming(A,B) = |A\ xor\ B|$, $Jaccard(A,B) = 1 - |A\ and\ B|/|A\ or\ B|$, and $Cosine(A,B) = 1 - |A\ and\ B|/\|A\|_2 * \|B\|_2$, where $|A|$ is the number of 1s in the bitvector A. The search cost is very similar for all the metrics (Fig 13). When Jaccard is used, the search cost is a little lower, but the differences are very small, and they might be just due to chance. Fig 14 shows the search time is the lowest when Hamming distance is used because the trees constructed using Hamming distance were shorter than the others, and therefore fewer levels in the tree were searched for matches. Likely due to the memory hierarchy and physical layout of the data, the search time does not depend only on the number of Bloom filters search, but also on their location in memory.

Varying Underlying Data Distribution: We use the following distributions for the underlying data in the Bloom filters: $nonrandom$, in which we insert in each Bloom filter $0 \leq i \leq N$ the $nbEl$ integers in the range [i*nbEl, i*nbEl+nbEl), and $random$, in which we insert in each Bloom filter $nbEl$ integers randomly chosen from a random range assigned to that Bloom filter. $nbEl$ is the actual number of elements in the Bloon filters, which is set to 100 for these experiments. In the $nonrandom$ case, there is no overlap between the sets represented by different Bloom filters, while in the $random$ case, we could have overlap between the sets. We expect that in a real scenario, the $random$ case is more common. The search cost is shown in Fig 15 and the search time is shown in Fig 16. We expected the performance in the $random$ case to be a little worse than in the $nonrandom$ case, since multiple Bloom filters might match a queried object. However, the overlap of ranges in the $nonrandom$ case was not enough to lead to a substantial increase in the search cost or search time for the $nonrandom$ case, even if the number of results was indeed larger for $nonrandom$ case. The search performance is very similar for both distributions used, which shows that Bloofi is robust to the underlying data distribution.

5.3 Experimental Results Conclusion

Our experimental evaluation shows that:

- Search cost increases logarithmic with N, but gets worse if the false positive probability at the root is 1. For optimal performance, the size of the Bloom filters should be based on the estimate for the total number of elements in the entire system. If the false positive probability at the root is below 1, even if very close to 1, the search cost is $O(d * log_d N)$.
- Search costs increases with order, since search cost is $O(d * log_d N)$, so low order is preferred for Bloofi.
- Storage cost is $O(N/d)$
- Insert cost and delete cost are $O(d * log_d N)$, and update cost is $O(log_d N)$
- Bulk construction gives slightly better search results, since the trees constructed with bulk construction are skinnier. However, differences in performance between bulk and incremental construction are small. The cost of sorting, which is the first step in bulk construction, is $O(N^2)$ in our implementation, so the operation is expensive. Our experimental results show that the Bloofi algorithm for insertion produces a tree very close to the tree obtained by using a global ordering of Bloom filters.
- The distance metric used to compare the "closeness" between Bloom filters does not have a big effect on the search cost of the resulting Bloofi tree, but the Hamming distance seems to lead to best search time.

6. RELATED WORK

Part of the existing work related to Bloom filters [5, 6, 8, 9, 10] is concerned with extending or improving the Bloom filters themselves and does not deal with the problem of searching through a large set of Bloom filters.

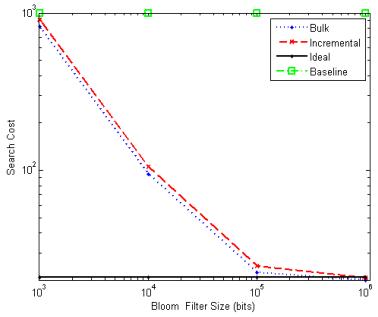

Fig 11. Search Cost vs Filter Size

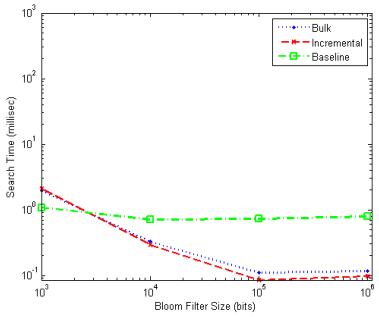

Fig 12. Search Time vs Filter Size

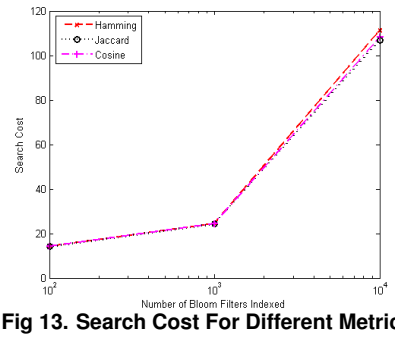

Fig 13. Search Cost For Different Metrics vs N

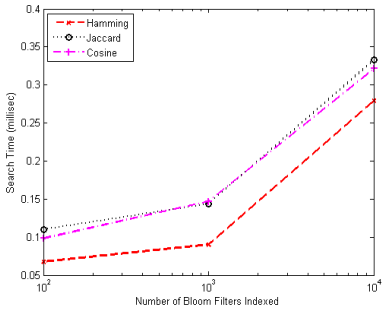

Fig 14. Search Time For Different Metrics vs N

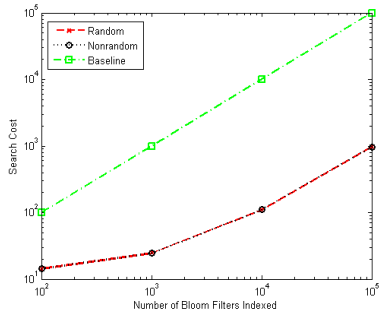

Fig 15. Search Cost For Different Distributions vs N

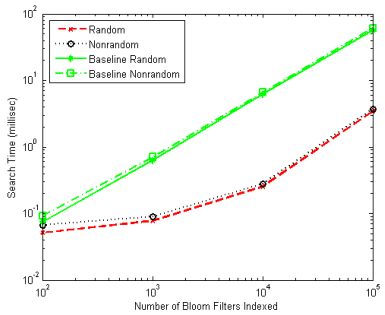

Fig 16. Search Time For Different Distributions vs N

Applications of the Bloom filters to web caching [9] use multiple Bloom filters, but their number is in general small and a linear search through the Bloom filters is performed. [11] uses Bloom filters to reduce the cost of semijoins in distributed databases. Most applications of Bloom filters[2, 4], use the Bloom filters directly and do not search through a large number of Bloom filters.

B+ trees inspired the implementation of the Bloofi tree. However, each node in Bloofi has only one value, and the children of a node do not represent completely disjoined sets, so multiple paths might be followed during search. Bitmap indexes [3] are usually used for columns with low number of distinct values, while the Bloom filters can have thousands of bits. S-trees [7] are used for set-valued attributes, but they were not designed specifically for Bloom filters.

7. CONCLUSIONS AND FUTURE WORK

We introduced Bloofi, a hierarchical index structure for Bloom filters. By taking advantage of intrinsic properties of Bloom filters, Bloofi reduces the search cost of membership queries over thousands of Bloom filters and efficiently supports updates of the existing Bloom filters as well as insertion and deletion of filters. Our experimental results show that Bloofi scales to tens of thousands of Bloom filters, with low storage and maintenance cost. In the extreme worst case, Bloofi's performance is similar with not using any index ($O(N)$ search cost), and in the vast majority of scenarios, Bloofi delivers close to logarithmic performance even if the false positive probability at the root is close to (but less than) one. For future work, we plan on exploring different ways of improving pruning efficiency when $probf = 1$ for the root.

8. ACKNOWLEDGEMENTS

Special thanks to Ciprian Crainiceanu for insightful discussions while developing Bloofi, and David Rapp, Andrew Skene, and Bobbe Cooper for providing motivation and applications for this work.

9. REFERENCES

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM*, 13(7), July 1970.

[2] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, 2002.

[3] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *SIGMOD*, 1998.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, and D. A. Wallach. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[5] S. Cohen and Y. Matias. Spectral bloom filters. In *SIGMOD*, 2003.

[6] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *SIGMOD*, 2006.

[7] U. Deppisch. S-tree: a dynamic balanced signature index for office retrieval. In *SIGIR*, 1986.

[8] S. Dutta, S. Bhattacherjee, and A. Narang. Towards "intelligent compression" in streams: a biased reservoir sampling based bloom filter approach. In *EDBT*, 2012.

[9] J. A. L. Fan, P. Cao and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. In *SIGCOMM*, 1998.

[10] M. Mitzenmacher. Compressed bloom filters. In *PODC*, 2001.

[11] J. Mullin. Optimal semijoins for distributed database systems. In *TSE*, 1990.

[12] R. Punnoose, A. Crainiceanu, and D. Rapp. Rya: a scalable rdf triple store for the clouds. In *Cloud-I*, 2012.

[13] M. Skjegstad. Bloom filter implementation. *https://github.com/magnuss/java-bloomfilter*.