A Distributed Algorithm for Finding Global Icebergs with Linked Counting Bloom Filters

Kui Wu*, Yang Xiao †, Jie Li‡, and Bo Sun§

* Computer Science Department
University of Victoria
BC, Canada V8W 3P6
† Computer Science Department
University of Alabama
Tuscaloosa, AL 35487-0209, USA
‡ Department of Computer Science
University of Tsukuba
Tsukuba Science City, Ibaraki 305-8573, Japan
§ Department of Computer Science
Lamar University,
Beaumont, TX 77710, USA

Abstract—Icebergs denote data items whose total frequency of occurrence is greater than a given threshold. When data items are scattered across a large number of network nodes, searching for global icebergs becomes a challenging task especially in bandwidth limited wireless networks. Existing solutions require a central server for ease of algorithm design and/or use random sampling to reduce bandwidth cost. In this paper, we present a new distributed algorithm to search for global icebergs without any centralized control or random sampling. A new type of Bloom filter, called linked counting Bloom filter, is designed to check the membership of a set and to store the accumulative frequency of data items. We evaluate the performance of our distributed algorithm with real data sets.

I. INTRODUCTION

With the advance of virtual communities over networks like Mobile Bazaar [4], it is often required to know popular items among mobile users, e.g., popular web sites, songs, or products. The local count of an item could be the quantity of the item or the times the user accesses the item. To know popular items in the virtual communities, we need to search for global icebergs, i.e., data items whose total counts in the network are above a given threshold value [1], [5], [8]. Generally speaking, it is unclear whether or not a local popular item is a global iceberg. The problem of searching for global iceberg also arises in other applications such as peer-to-peer networks where a popular item may be stored at different peers.

A naive solution to this problem is to allocate a central server and ask all users in the network to ship their data items to the server so that it can merge the data and find the global icebergs. It is obvious that this method incurs prohibitive bandwidth cost and thus becomes infeasible for wireless networks. To overcome this problem, Zhao et al. [9] use random sampling to reduce bandwidth cost. Since the sampling-based method is not very accurate without a large number of samples, they design a counting-sketch based scheme to reduce sampling rate. In both cases, a central server has to be selected and random sampling tends to work well only when the number of data items is huge. In distributed wireless networks, however, users may have limited processing capacity or may not be willing to serve as a central server. As such, we need to design a distributed algorithm that can effectively find global icebergs.

Many papers have been devoted to search for icebergs over streaming data in a single node [1], [5], [8] or in a distributed environment [7]. Several assumptions are generally made in previous work: (a) a central server is responsible for collecting data [9]; (b) random sampling can represent the distribution of popular items [9]; (c) a globally frequent item is also locally frequent somewhere [7]. These assumptions are either too strong or undesirable for wireless networks. It has been pointed out that assumption (c) is not effective if items are evenly distributed among network users [9]. Up to date, we have not seen any distributed algorithms suitable for searching global icebergs over resource-constrained networks, e.g., wireless networks.

In this paper, we investigate the problem from a fresh new angle. The distributed control and lightweight traffic load of our method make it a special niche for iceberg searches over wireless networks or peer-to-peer networks. This paper includes the following contributions:

1) Instead of assuming certain distribution of data items among different nodes or using sampling methods, we truthfully collect all data items from all nodes in a distributed fashion. To make this seemingly awkward solution bandwidth efficient, we design and implement a new type of Bloom filter, called linked counting Bloom filter. A two-layer linked counting Bloom filter is also implemented to further save bandwidth when the percentage of empty entries in the filter is large.

2) We do not depend on any central server. We present a
A Bloom filter [2] is a simple space-efficient randomized data structure for approximating a set, which supports membership queries. Bloom filters use a set of hashing functions and achieve space efficiency at the cost of a small probability of false positives. Due to the benefit of saving memory space and bandwidth, bloom filters have been broadly adopted in many network applications [3].

To represent a set \( S \), a Bloom filter uses an array \( B \) of \( m \) bits, initialized to all 0’s, and \( k \) independent hash functions \( h_1, h_2, \ldots, h_k \) with integer output ranging from 1 to \( m \). Each hash function maps an input data item to a random number uniformly selected over the range \( 1, 2, \ldots, m \). To insert a data item \( x \) into the set \( S \), the bits \( B[h_i(x)], i = 1, 2, \ldots, k \), are set to 1. Figure 1 shows an example. A bit can be set to 1 multiple times when a hash collision happens or the same data item has already been inserted into \( S \), but only the first change takes an effect. To check if a data item \( x \) is in set \( S \), all bits \( B[h_i(x)], i = 1, 2, \ldots, k \), should be 1. If the condition is true, we assume that \( x \) is in \( S \). Obviously, a Bloom filter guarantees not to have any false negative, i.e., returning “no” even though \( S \) actually contains \( x \), but may have a false positive, i.e., suggesting that \( x \) is in \( S \) even if it is not.

The above introduction only includes the operations of inserting an element into a set and checking the membership of a set. If we want to delete an element from a set, unfortunately, we cannot hash the element to be deleted and reverse the corresponding bits from 1 to 0. This is because different elements may be hashed to the same location and thus setting the location to 0 may create a false negative.

To avoid the problem, Fan et al. [6] design a different Bloom filter, called counting Bloom filter, where each entry is not a single bit but instead a counter. When an element is added, the corresponding counter is increased by 1; when an element is removed, the corresponding counter is decreased by 1. When a counter decreases from 1 to 0, it indicates that the element is removed. The counting Bloom filter solves the problem of removing an element from a set at the cost of using more bits each entry. It has been analyzed that 4 bits each entry should be enough for the counting Bloom filter in realistic application settings [6].

B. Linked Counting Bloom Filter

The idea of counting Bloom filter is helpful in searching for global icebergs if we include the frequency of an element in the filter. To do this, each entry of the Bloom filter is not a single bit but instead a counter to record frequency. However, counting Bloom filter with this revision cannot be used directly to find global icebergs. Figure 2 shows an example where the modified counting Bloom filter cannot correctly find the global icebergs. In the example, the counter vector is used to record frequency of items. When an entry is set multiple times, the sum of the frequencies (i.e., \( f_a + f_b \) in the example) does not correctly reflect the frequency of any element.

We solve the above problem based on the observation that the probability that \( k \) hash functions hash two different elements into exactly same locations is extremely small and can be safely ignored (The analysis is in Section IV.). We introduce a linked list to each entry of the Bloom filter and once a hash collision occurs, we use the linked list to record each individual frequency value. Frequency values are added only when the same element has been found in the Bloom filter. We call this type of Bloom filter linked counting Bloom filter.

Figure 3 illustrates the operations of linked counting Bloom filters with three hash functions \( h_1, h_2, h_3 \). Assume that node 1 has two items \( a \) and \( b \) with frequency \( f_a \) and \( f_b \), respectively. Assume that node 2 has one item \( b \) with frequency \( f'_b \). Initially, node 1 inserts item \( a \) into the Bloom filter by hashing \( a \) with \( h_1, h_2, h_3 \) and setting \( B[h_1(a)], B[h_2(a)], B[h_3(a)] \) to \( f_a \). When node 1 inserts item \( b \) into the Bloom filter, it first checks if entries \( B[h_3(b)], B[h_2(b)], B[h_3(b)] \) have been set. If any one of the entries is empty, item \( b \) is a new item and \( f_b \) should be stored into the entries \( B[h_1(b)], B[h_2(b)], B[h_3(b)] \), respectively. Note that in the example, \( h_2(a) = h_3(b) \) and thus the entry \( B[h_2(a)] \) uses a linked list to include both \( f_a \) and \( f_b \).
Assume that the linked counting Bloom filter is sent from node 1 to node 2 and node 2 inserts item b into the Bloom filter. By checking $B[h_1(b)], B[h_2(b)], B[h_3(b)]$, node 2 finds out that all these entries are non-empty and assumes that item b may be an item already included in the filter. Nevertheless, since it is not allowed to store the identities in the Bloom filter, it is still unclear what should be the correct frequency of item b set by node 1 if all these entries include multiple linked values.

To solve this problem, we use a method called frequency matching. The basic idea is that if item b is an existing item in the Bloom filter, there must be a common value stored in $B[h_1(b)], B[h_2(b)], B[h_3(b)]$. For instance, $f_b$ can be found in all the three entries in the above example. If a match is found, item b is assumed to be in the Bloom filter and its frequency, $f'_b$, is added up with the existing count $f_b$. If such a match cannot be found, item b must be a new item not in the filter. Its frequency should not be added up to any existing values but instead should be stored separately in the linked lists.

To remove an item, say item a, from the linked counting Bloom filter, search for a frequency match in the entries of $B[h_1(a)], B[h_2(a)], B[h_3(a)]$ and remove the values in the match. In the example, the value $f_a$ will be removed from the entries of $B[h_1(a)], B[h_2(a)], B[h_3(a)]$. If multiple frequency matches could be found, randomly delete one match. In this case, we may delete the incorrect item. But the probability of this type of errors is very small and can be ignored as shown later.

The linked counting Bloom filter is very flexible and supports useful count queries that are impossible with traditional Bloom filters. The insertion and deletion of elements are also easy. On the downside, the linked counting Bloom filter introduces extra bits on recording the links and the counts. But the extra overhead could be well justified by the facts that (1) no identities of elements are stored, (2) links are introduced only when hash collisions happen, (3) a link can be actually implemented with 1 bit only because we only need to decide whether or not the current value is the end of the linked list, and (4) the linked counting Bloom filter can be compressed effectively with simple compression algorithms like arithmetic coding.

**Fig. 3. An example of linked counting Bloom filters**

### III. A Distributed Algorithm for Searching for Global Icebergs

Our algorithm is based on linked counting Bloom filters and random gossiping. To simplify description, a filter is by default a linked counting Bloom filter in the following. Given a threshold frequency value, the source node can find global icebergs with the following steps:

1) **Step 1:** The source node builds a linked counting Bloom filter using its own data set. To save resource, if a local data item is already a global iceberg, this item is returned as one answer and it is not recorded in the filter.

2) **Step 2:** The source node randomly selects one of its neighbors and sends the filter together with the frequency threshold value to the neighbor. The filter can be uniquely identified by the ID of the source node and the timestamp when the source node first sends the filter.

3) **Step 3:** Each node, once receiving the filter, checks the filter and its own data items. If a data item turns out to be an iceberg, the result is returned to the source node using any underlying routing protocol. To save space, the item is removed from the filter. For a local data item that is not currently an iceberg, the node checks the membership of the data item in the filter and inserts (if the data item is new to the filter) or adds (if the data item already exists in the filter) the frequency of the data item to the filter. After that, the node randomly selects one of its neighbors and sends the filter to that neighbor. In the mean time, the node needs to record the neighbor ID so that it will not forward the filter twice to the same neighbor.

4) **Step 4:** If a node already receives the filter before, it should not process any filters received later but instead simply forward the filters to a random neighbor to which it has not sent the query before.

5) **Step 5:** Repeat steps 3 and 4.

Since a node will not forward the filter if it has received the filter d times where d is the number of neighbors of the node, the algorithm will finally terminate after at most $d \times N$ steps where d is the average node degree and N is the number of nodes.

### IV. Probability of False Positives and Multiple Frequency Matches

In this section, we analyze the probability that k independent hash functions, $h_1, h_2, \ldots, h_k$, hash two different items, a and b, into the same locations. This probability is equivalent to the probability that two entry vectors $(x_1, x_2, \ldots, x_k)$ and $(y_1, y_2, \ldots, y_k)$ weakly match where $x_i = h_i(a) \in [1, m]$ and $y_i = h_i(b) \in [1, m]$. Two vectors are considered to weakly match if the elements of one vector can be obtained by the permutation of the elements of the other vector. We use the balls-and-bins model to analyze the above probability: Randomly throw $k$ balls into $m$ bins and record the bins that have at least one ball. Repeat the same experiment. The probability that the two experiments have the same set of bins with each bin having at least one ball is:
Bloom filter can be calculated as $m \ast (l + 1) \ast q - m$ (We need 1 bit to store the link to next element.). Obviously, if $(l+1)\ast q < 1$, there is no benefit of using the two-layer filter, but in most cases, $(l+1)\ast q$ should be much larger than 1, since it is required that the probability of false positives is controlled to be small, i.e., the value $q$ is usually large.

VI. EVALUATION

We evaluate the performance of our algorithm in terms of hit rate of icebergs, error rate, and average bandwidth cost per query:

1) Average bandwidth cost per query: it is defined as the total number of bits transmitted in the network averaged by the total number of queries sent from source nodes.

2) Hit ratio of icebergs: it is defined as the ratio of the total number of returned icebergs over the total number of actual icebergs.

3) Error rate: it is defined as the total number of errors divided by the total number of icebergs. An error occurs if an item that is not an iceberg is returned as an iceberg.

We use the data set collected by Ziegler [10] in a 4-week crawl from the Book-Crossing community (www.bookcrossing.com), which has 278858 members and 1,149,780 ratings (numbered from 0 to 10) about 271,379 books. Since the data is sparse (i.e., each user has only several ratings and many books have been mentioned very few times), we truncate the data set and artificially combine every 1000 users as one virtual user so that the total number of users becomes smaller but each user includes more data items. We then remove the books that have less than 25 mentions. In this application, an iceberg is defined as a book whose total threshold rating is above a threshold value specified by the source nodes in the virtual community. The threshold value is set to 200 in the experiment.

We assume that the users are randomly distributed within a rectangular area of size 1000 meters by 500 meters. The radio range of all users is assumed to be the same and is set to 150 meters so that the average network node density is moderate (around 10). We randomly select 5 users to send out queries for global icebergs. We change the number of entries in the linked counting Bloom filter, $m$, to investigate the tradeoff between bandwidth cost of Bloom filter and error probability. Three different values of $m$, 12460, 24920, 37380 are evaluated, corresponding to $m/n = 5$, $m/n = 10$, and $m/n = 15$ respectively, where $n$ is the number of books. We study the two-layer linked counting Bloom filter since it costs less bandwidth when the percentage of empty entries in the linked counting Bloom filter is not small. For each non-empty entry, 15 bits are used to store the count value and 1 bit for the link. The number of hash functions $k$ is a variable in the simulation.

Regarding average bandwidth cost per query, the naive method requires all nodes to send data items to a source node so that the source node can locally search for icebergs. This method needs the transmission of at least $5MB$ for each query, estimated for sending all 224,999 ratings with...
random hash functions, which are hard to guarantee in reality. The results mainly because the theoretical model assumes perfectly experiment, the error probability is larger than the theoretical bounds of error probability in Section IV tends to 0 when \( m/n \) is large. But in our experiment, the error probability is larger than the theoretical results mainly because the theoretical model assumes perfectly random hash functions, which are hard to guarantee in reality.

![Fig. 5. Average bandwidth cost per query](image)

Figure 6 and Figure 7 show the results of hit ratio of global icebergs and error rate, respectively. It could be seen that the hit ratio increases with the increase of the number of hash functions \( k \) and the increase of \( m/n \), i.e., the ratio of number of entries in the filter over the number of stored data items. When \( m/n \) is not less than 10 and \( k \) is larger than 2, the hit ratio of global icebergs is above 99% and the error rate is smaller than 0.5%. The accuracy is obtained at the cost of a slightly large bandwidth overhead since a larger \( m/n \) value usually requires larger communication overhead. It is worth noting that the theoretical bounds of error probability in Section IV tends to 0 when \( m/n \) is large. But in our experiment, the error probability is larger than the theoretical results mainly because the theoretical model assumes perfectly random hash functions, which are hard to guarantee in reality.

![Fig. 6. Hit ratio of global icebergs](image)

![Fig. 7. Error rate](image)

VII. Conclusion

Searching for global icebergs is an essential operation in online virtual communities like Mobile Bazaar [4] or peer-to-peer networks. In this paper, we truthfully collect all data items from all nodes in a distributed fashion and design a new type of Bloom filter, called linked counting Bloom filter, to reduce bandwidth cost. A distributed algorithm is proposed to search for global icebergs. We provide the upper and lower error bounds of the new algorithm and evaluate its performance with real data sets.

Currently two linked counting Bloom filters cannot merge together correctly. Designing more powerful Bloom filters to support the merging and counting operations is left as our future work. We have observed that it is hard to compare our distributed method to existing methods [7], [9] that use centralized control. Our further work is to investigate the possibility of integrating different methods.

REFERENCES