Importance-aware Bloom Filter for Managing Set Membership Queries on Streaming Data

Ravi Bhoraskar*, Vijay Gabale†, Purushottam Kulkarni‡, and Dhananjay Kulkarni§

* Computer Science and Engineering, University of Washington, Seattle, bhora@cs.washington.edu
† IBM Research, India, vijay.gabale@gmail.com
‡ Department of Computer Science and Engineering., Indian Institute of Technology Bombay, puru@cse.iitb.ac.in
§ Asia Pacific Institute of Information Technology, Sri Lanka, dhananjay@apiit.lk

Abstract*—In this work[1], we consider a set of networking applications which generate or process a continuous stream of data items, for example, a web-cache which processes a stream of web-objects. These applications often require to answer membership queries for duplicate detection on an unbounded set of data items. Two key challenges to answer such membership queries are the limited space to store the entire stream and the different importance-levels associated with different data items. For instance, a web-caches are of finite sizes and the cost of fetching objects into the cache is proportional to the size of objects. Motivated by such examples, our work focuses on developing a time and space efficient indexing and membership query scheme which takes into account *importance* levels of objects in a data stream. We propose Importance-aware Bloom Filter (IBF) which provides a set of insertion and deletion algorithms to handle membership queries on a data stream. Our evaluation of IBF for a synthetic as well as a real data set of a stream of Youtube videos, shows that IBF has close to 0% error in answering membership queries on important data items, and it results in 4-fold better performance in comparison to other importance-agnostic Bloom filter-based schemes. Importantly, we also find properties of IBF analytically via a Markov model based analysis. Thus, we believe, IBF provides a practical framework to balance the application-specific requirements to index and query data streams based on the data semantics.

I. Introduction

Several networked-services such as news feeds [6], stock trades [1], [2], [4], sensors-based cyber physical systems [5], [3], and ever increasing web-based utilities etc., are continuously generating data. Such *data streams* are also being consumed by various networking and data distribution applications, e.g., web-pages are used by web-crawlers to update the page-ranks, or web-objects are used by client-applications to load a web-page or to play multimedia content etc.

Along with generation or consumption of such data streams, these applications often process the data streams to record relevant information. For instance, before delivering the web-objects, content distribution networks often store web objects in a proxy-cache (a temporary store of web-page streams closer to the users); such storage of a stream of web-objects, thus, can reduce the network bandwidth and latency costs due to local cache-hits. Web crawlers too process a stream of web-links and maintain temporary state to avoid unnecessary crawling of already visited web-links. An important factor that determines the efficacy of such intermediate processing, and thus application performance, is the effective "state" maintenance of data streams. To understand the notion of state maintenance, consider the following scenarios,

Scenario 1: Caching at Edge Proxies. Consider a content distribution network (CDN) with a set of cooperating edge proxies. The proxies "cooperate" by sharing information about cached objects, —each proxy maintains *state* about the set of cached objects at other proxies. Such a collective cache is used to resolve web requests by restricting the processing "closer" to the users. A critical constraint on proxies, while processing a stream of web-objects, is the finite cache size.

Further, the cache contents are updated frequently based on several replacement policies (LFU, LRU etc.). As a result, the updated state of a cache needs to be periodically exchanged with other caches to ensure that the cooperative-caches work efficiently. However, such periodic exchange of cache state can incur significant network overhead. To minimize such network overheads, one effective scheme to exchange state information is via a Bloom filter-based Summary-cache [9], [14]. The Bloom filter, being a fixed-sized data structure has fixed overheads for storage and exchange. However, a Bloom filter is probabilistic with respect to representation of a set of objects[2]— false positives and false negatives can result on membership queries. For a Summary-cache, the objects inserted in the Bloom filter are web-object identifiers. Before serving a web-object from a remote web-server, a proxy can look up the web-object *locally* in Bloom filter representations of other caches. If a local-hit occurs, then the object can be served from the corresponding cache.

A Bloom filter is a probabilistic data structure and queries on the cache-state may result in false positives or false negatives [3]. An initial desired property for the Summary-cache is to keep false negatives low. A more subtle requirement can arise when cache-misses result in differentiated costs, e.g., the bandwidth and latency costs of fetching a large web object is higher than a smaller object. Thus, a more suitable property is to have lower false negative rates for larger-sized frequently-

[1]This work was done when Ravi Bhoraskar and Vijay Gabale were students at IIT Bombay.

[2]The Bloom filter size is constant and hence the tradeoff.

[3]For a query, a false costive occurs when the Bloom filter answers 'yes' to a query even if the data item is not inserted in the filter. Similarly, a false negative is when the Bloom filter answers 'no' to a query even if the data item was inserted in the filter.

978-1-4673-5494-3/13/$31.00 © 2013 IEEE

accessed objects at the cost of smaller-sized seldom-accessed objects.

Scenario 2: A simple Web-crawler. Consider a simplistic web crawler which collects a set of links on a web page and then crawls those links recursively. A subset of those links, being more popular, may appear on multiple web pages. Thus, when the crawler is visiting a stream of links, it is important to identify such popular links as accurately as possible and avoid crawling them multiple times.

To meet these requirements, the crawler has to maintain a history of previously of crawled-pages. Since the stream of crawled pages is potentially very large, given a finite storage space or finite time for duplicate detection, the state of crawled pages has to be approximate or reset periodically. Similar to the web-caching scenario, a nuanced requirement for the history maintenance would be to maintain an accurate record of the popular pages, while less popular pages can be evicted from the history more frequently. Since less popular pages occur infrequently, they will cause fewer redundant crawls.

Examples in the above two scenarios fit the general category of applications that require to maximize "cost savings" while indexing and detecting duplicates in a data stream with finite storage space. For example, for a cooperative cache, the web-objects are required to be indexed in a fixed size Summary-cache such that the accuracy of serving large-sized objects is maximized. Thus, a state building mechanism cognizant to the importance of data items can prove vital to the performance such applications. In this work, we consider storing such state of a data stream in a finite memory space while processing queries on an unbounded stream of data items. There exist numerous challenges in storing and querying such unbounded data streams.

1) Since data streams are continuous and unbounded, it is impractical to store the entire data during query processing, especially so in finite-memory systems. Hence, traditional techniques (for indexing and storing) used with relational databases [7] are not directly applicable in the context of data streams.
2) Given a finite memory system, it is difficult to provide precise answers to queries. One way to achieve precision is the time-based *sliding window* [8] technique. However, such a technique is constrained by the finite size of the sliding time-window, and may not be applicable to bursty data arrival or unpredictable data distributions. Hence, given a finite memory system, one has to rely on approximating the answers to set membership queries on an unbounded data stream.
3) A subtle challenge lies in capturing the importance of data items while answering the set membership queries approximately. Since important data items have higher "cost" implications, the indexing and state maintenance scheme should reflect the differentiation in object-importance.

In this respect, our work focuses on developing an indexing and membership query scheme for a stream of data items with different *importance* levels. Our goal is to store an approximate summary of each data item using a hash-based index, the Bloom Filter (BF) [9]. However, since the volume of data insertions is high the traditional Bloom Filter would quickly "fill up", and lead to a large number of false positives (FPs). Recent work, Stable Bloom Filters (SBF) [12], limits the false positives by clearing cells of the filter over time, however, it is agnostic to data importance, both during insert and delete operations. Thus, a key challenge lies in exploiting the data-importance semantics during insertion and deletion of items to maximize performance of query processing. Specifically, in this work, we make following contributions.

- We propose Importance-aware Bloom filter (IBF), a time and space efficient data structure for indexing and querying data items on an unbounded set based on data-semantics.
- Via Markov chain based analysis, we show the IBF is stable, i.e., the Bloom filter has a constant upper-bound on the false-positive and false-negative rates irrespective of the stream size.
- Our evaluation of IBF for a synthetic as well as a real data set shows that IBF has close to 0% error in answering membership queries on important data items, and results in up to 4-fold better accuracy (in terms of false positive and false negative rates) compared to other Bloom filter-based schemes.

The rest of the paper is organized as follows. In next section, Sec. II, we describe preliminaries of our problem, and give a brief sketch of IBF. In Sec. III, we describe IBF in detail and analyze its properties theoretically. Next, in Sec. IV, we show effectiveness of IBF by experimentally comparing it with Bloom filter-based mechanisms. In Sec. V, we describe prior work in indexing and querying streaming data. Finally, in Sec. VI, we discuss future work, and conclude the paper.

II. Background and Problem Formulation

Our work is partially inspired by the work in [12] which proposes a data structure to approximately answer membership queries on a stream of data items. In this section, first, we briefly describe work in [12]. Then, we explain why work in [12] is not suitable to our problem. Subsequently, we formally formulate the problem under consideration.

A. Bloom filter

Traditionally, the Bloom filter [9] (BF) data structure has been used to store data items to approximately answer the membership query. A membership query asks whether *a data item d_i is present in the memory* or not. Note that, a Bloom filter only stores a hash for each data item, whereas the data items are physically stored in a separate memory (typically an external disk-memory), distinct from the Bloom filter. To answer a membership query for an item, a Bloom filter takes the hash of the data item, and returns the answer as 'Yes' or 'No' in a constant time, but with some probability. The entire Bloom filter can reside in main memory, and it takes constant number of memory accesses to know whether an item is present or not. Thus, it is time-efficient than indexing data structures like B-Tree or B^+-Tree, which require several memory accesses to search for an item. However, the constant time search of Bloom filter comes at the cost of the

probabilistic behavior in answering the membership query. We now define Bloom filter formally.

A Bloom filter is essentially an array of m bits, initially set to 0. When a new item d_j is to be stored in the memory, BF uses K uniformly distributed and independent hash functions, and computes K bit locations: $\{h_1(d_j), h_2(d_j), \ldots, h_k(d_j)\}$, to be set to 1. To answer a membership query on data item d_j, BF again applies K hash functions, and checks whether all of the corresponding K bit locations are set to 1 or not. If so, BF replies 'true' to the query, indicating the presence of a duplicate for data item d_j. However, it is possible that the K bit locations are set due to insertion of some other data items previously stored in the filter (which were mapped to a subset of these K bit locations). This gives rise to a false positive; the case where BF can incorrectly answer 'true' to a given query. To explain the false positives formally, consider a query $Q(d)$, which tests for the membership of item d. A false positive occurs when d is not present in BF, but $Q(d)$ returns true (d is present in BF). A false negative occurs when $Q(d)$ returns false, although d was inserted in the BF.

B. Stable Bloom Filter

The basic BF explained above, with a fixed number of bits, may not be applicable to store the data items in a stream. This is because, as more and more elements arrive, the fraction of zeros in the BF decreases continuously, and the false positive rate increases accordingly. Eventually, the false positive rate reaches the limit, 1, where every distinct item is reported as a duplicate. To avoid this problem, the work in [12] extends the basic Bloom Filter data structure, and proposes Stable Bloom Filter (SBF), which (1) adds an "eviction" operation to BF, and (2) supports querying on a stream of data items. This way, by deleting a set of items, SBF avoids the error rate from exceeding a predefined threshold.

We now give a brief description of SBF. A SBF is defined as an array of integers SBF[1],...,SBF[m]. The size of the array is m elements, and each element can take a minimum value of 0, and a maximum value M. Each element of the SBF is called as a cell. Each cell of the array is allocated d bits; the relation between M and d is $M = 2^d - 1$. The initial value of the cells is zero. Each newly arrived item in the stream is mapped to K cells by some uniform and independent K hash functions. As in a regular Bloom filter, SBF checks if a new item is duplicate or not by checking whether all the cells the item is hashed to are non-zero. After the duplicate detection process, before inserting new item into the SBF, it randomly decrements P cells by 1 so as to make room for fresh items, and then sets the K cells, used during duplicate detection process, to M.

In [12], the authors prove that after a number of iterations, the fraction of zeros in the SBF will be non-zero irrespective of the values of M, K or P. This is termed as the stable property, and hence the name Stable Bloom Filter. Thus, because of the presence of a definite set of cells set to '0', SBF proves that, if the stream elements are uniformly distributed, the false positive rate can be upper bounded.

However, we observe that, the work on SBF in [12] is not suitable to our problem since SBF does not differentiate between items based on the importance values while during indexing and membership query operations. Thus, the processing of a data stream by SBF may result in a high cost if a large fraction of data items with high importance suffer from high false positive rate.

C. Importance-aware Bloom Filter (IBF) overview

To ensure, lower false positive and false negative rates for important data items, we design Importance-aware Bloom Filter (IBF). In IBF, as a first step, we modify the insert operation in SBF [12] to come up with an importance-aware data structure. The overall result of such modification is that, items having high importance are stored for longer time. Similarly, we evict elements from IBF such that, the items having high importance have less probability of eviction. The importance-aware insert and delete operation thus result in lower false positives (and lower false negatives) for items with high importance. At the same time, as a desirable effect, (which we confirm through theory and experiments) the false positive rate and false negative rate for items with lower importance remains reasonable low.

D. Problem Formulation

Following is a formal statement of the problem to index and query an unbounded data stream. Let $\mathbb{I} : d_i \rightarrow \mathbf{IMP}$ be an importance-function that maps some attribute of the data item d_i to corresponding importance value imp_i, We assume that items arrive in a stream denoted by $D = d_1, d_2, \ldots, d_n$, where n is the number of items arrived so far. The value of n is assumed to be unbounded. We assume that a fixed hash function is used each data item to convert it into a number, and henceforth, when we refer to an item d_i, it is the hash or fingerprint of the original data item. Now, given D, I and a finite amount of memory S, our *overall objective* is to store items in D so that membership of an item $d_i \in D$ can be determined in constant time. Further, the confidence of answers to membership queries corresponds to imp_i (importance values) of data items—low false negative rates and low false positive rates for data items with data items with higher importance.

III. The IBF Algorithm and Analysis

TABLE I
IBF Parameters

Parameter	Symbol
Storage space	SS
Total number of cells	m
Maximum value of a cell	M
Importance map or function	$\mathbb{I}$
No. of hash functions	K
No. of cells to delete	P

As we mentioned in last section, we insert or delete elements in IBF such that, important data items have lower probability of false positives or false negatives. In this section, we first describe a simple extension of SBF, IBF-2C (2C stands for two classes). In comparison to SBF, which sets the values of cells to M while inserting a new item, in IBF-2C, we either

set the cell values to $\frac{M}{2}$ or to M, depending on importance of the data item. In a generic case, we assume that a function f is given to us, which maps the importance value to a number between 1 and M, i.e., $f : imp_i \rightarrow (1, M)$. In case of IBF-2C, this function maps the importance value to either $\frac{M}{2}$ or M. Next, we describe a generalization of IBF, IBF-MC, where while inserting an item, we set the cell values to a number $\in (1, M)$ based on the importance value of the data item. For IBF-2C as well as IBF-MC, we present analysis of false positive and false negative rates, and prove the stability property. Our proofs are inspired by the lemmas in SBF [12], however, our approach is more generic in terms of modeling IBF using a Discrete Time Markov Chain (DTMC).

A. IBF-2 Class (2C)

We now introduce IBF-2C, an Importance-aware Bloom filter which has importance-aware insert operation, and random delete operation.

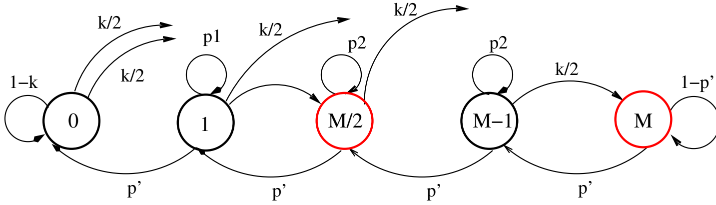

Fig. 1. IBF-2C: Important Insert Random Delete

Importance-aware Bloom Filter-Two Class (IBF-2C) An IBF-2C is defined as an array of integers $IBF[1]$, ... , $IBF[m]$. The value of each cell of the array is between 0 (minimum) or M (maximum). The update process follows Algorithm 1 with K independently and uniformly distributed hash functions. Each cell of the array is allocated d bits; the relation between M and d is $M = 2^d - 1$.

We now explain Algorithm 1. The parameters used in the algorithm are shown in Tab. I. In this algorithm, each incoming element is mapped to K cells by K uniform and independent hash functions. We then check, by probing the K cells, whether all the cells are non-zero, and see if the element is a duplicate. After this step, we update IBF-2C as follows. We first pick P cells randomly, and decrement the cell value by 1 (unless it is already zero). This step is to make room for future elements in the stream. We then set the same K cells to either $\frac{M}{2}$, if the data item is not important or M, if the data item is important.

Analysis: We analyze IBF-2C, and prove the stable property, which states that when the number of items in stream are sufficiently large, the fraction of zeros in IBF-2C will converge to a fixed value irrespective of the values of M, K, and P. This is an important property from both theory as well as practical point of view as it bounds the false positive rate.

To analyze IBF-2C, we assume that the data items as well as the importance values are uniformly distributed. We then model the cell value of IBF-2C as a Discrete Time Markov Chain as shown in Fig. 1. Note that, each iteration of the algorithm is a time step in the Markov chain. Thus, in each time step, the cell value is either decremented by 1 or is set to $\frac{M}{2}$ or M, or remains the same. These conditions are captured in transition probabilities in the Markov chain as shown in fig. 1. If a cell value is T, after one iteration it goes to state $T - 1$ with probability $p' = p \times (1 - k)$, where p is the probability of decrementing a cell, and k is the probability setting the cell. Note that $p = \frac{P}{m}$ and $k = \frac{K}{m}$. If the cell is set, it goes to state $\frac{M}{2}$, with probability $\frac{k}{2}$ or to state M with probability $\frac{k}{2}$. Otherwise, the cell remains in the same state. It is easy to see that this Markov chain is indeed a valid discrete time Markov chain with irreducibility (every state is reachable from every other state) and aperiodicity (period of recurrence for each state is 1). As shown in [22], for any irreducible, aperiodic Markov chain, the limiting probabilities V_i for each state exist and are unique. We now prove the stable property of IBF which states that, after a large number of iterations, the expected fraction of zeros in IBF are constant. This theorem would show that there is always enough 'room' in IBF. Further, it would be useful to show the bound on false positive and false negative rates of IBF-2C. To prove this property, we first calculate the limiting probability V_0.

Input: A sequence of numbers
Output: A sequence yes or no corresponding to each input number
Initialize IBF[1],...,IBF[M] to 0.
foreach *number* x_i **do**
  Probe K cells IBF[$h_1(x_i)$],...,IBF[$h_K(x_i)$].
  If(none of the above K cells is 0) DupFlag = yes
  Else DupFlag = no
  Select P cells uniformly at random.
  foreach *each cell IBF[j]* $\in$ { *IBF[j_1],..., IBF[j_p]* } **do**
    If (IBF[j] $\geq$ 1) IBF[j] = IBF[j] - 1
  end
  foreach *each cell* $\in$ *IBF[$h_1(x_i)$],...,IBF[$h_K(x_i)$]* } **do**
    If($f(imp(x_i)) < \frac{M}{2}$ and IBF[$h(x_i)$] $< \frac{M}{2}$) IBF[$h(x_i)$] = $\frac{M}{2}$
    Else IBF[$h(x_i)$] = M
  end
  Output DupFlag.
end

Algorithm 1: Duplicate detection with IBF-2C.

Theorem III.1. *Given an IBF-2C of m cells, if in each iteration, a cell is decremented by 1 with a probability p and set to importance value imp* $\in \{\frac{M}{2}, M\}$ *with a probability k, the limiting probability of a cell becoming zero exists.*

Proof: The existence of probability is straightforward from the Markov model of IBF-2C. Since the Markov chain is irreducible, aperiodic and stable, each state i will have a non-zero limiting probability V_i [22]. Thus, there is non-zero probability that a cell will have its value zero in limiting cases. We now compute the limiting probability of a cell becoming zero, V_0. When limiting probabilities exist, the probability of the system entering a state is same as the

probability of leaving the state, and the sum over probabilities of all states is 1 [22].
Let $P_{i,j}$ denote the probability of transition from state i to state j. Then, for every state i, we have

$$\sum_j P_{j,i}V_j = (\sum_k P_{i,k})V_i \quad (1)$$

$$\sum V_i = 1 \quad (2)$$

We compute steady state probabilities for each state as follows. Refer to Fig. 1 to see the transition probabilities.

For state M, using eq. 1, $\frac{k}{2}\sum_{i=0}^{M-1} V_i = p'V_M$.
But by Eq. 2, $\sum_{i=0}^{M} V_i = 1 \rightarrow \sum_{i=0}^{M-1} V_i = (1 - V_M)$.
Thus, $V_M(p' + \frac{k}{2}) = \frac{k}{2}$, which implies, $V_M = \frac{\frac{k}{2}}{p'+\frac{k}{2}}$.
For state (M-1), using eq. 1, $(\frac{k}{2} + p')V_{M-1} = p'V_M$.
$\rightarrow V_{M-1} = \frac{p'}{p'+\frac{k}{2}}V_M, \rightarrow V_{M-1} = \frac{p'}{p'+\frac{k}{2}}\frac{\frac{k}{2}}{p'+\frac{k}{2}}$.
For state i, $\frac{M}{2} < i < M, V_i = \left(\frac{p'}{p'+\frac{k}{2}}\right)^{(M-i)} \frac{\frac{k}{2}}{p'+\frac{k}{2}}$
For state $\frac{M}{2}$, using eq. 1,
$\frac{k}{2}\sum_{i=0}^{\frac{M}{2}-1} V_i + p'V_{(\frac{M}{2}+1)} = \left(p' + \frac{k}{2}\right) V_{\frac{M}{2}}$.
Replacing $V_{(\frac{M}{2}+1)}$ and
$\sum_{i=0}^{\frac{M}{2}-1} V_i$ by $\left(1 - V_{\frac{M}{2}} - \sum_{i=\frac{M}{2}+1}^{M} V_i\right)$, we get
$\frac{k}{2}\left(1 - V_{\frac{M}{2}} - \sum_{i=(\frac{M}{2}+1)}^{M} V_i + \left(\frac{p'}{p'+\frac{k}{2}}\right)^{\frac{M}{2}}\right) = \left(p' + \frac{k}{2}\right) V_{\frac{M}{2}}$
If $A = \sum_{i=(\frac{M}{2}+1)}^{M} V_i$, and $B = \left(\frac{p'}{p'+\frac{k}{2}}\right)^{\frac{M}{2}}$,
we get $V_{\frac{M}{2}} = \left(\frac{\frac{k}{2}}{p'+k}\right)(1 - A - B)$.

For state $\left(\frac{M}{2} - 1\right)$,
$p'V_{\frac{M}{2}} = (p' + k)V_{\frac{M}{2}-1} \rightarrow V_{\frac{M}{2}-1} = \left(\frac{p'}{p'+k}\right) V_{\frac{M}{2}}$
Similarly, for state i, $\frac{M}{2} > i > 0, V_i = \frac{p'}{p'+k}V_{\frac{M}{2}}$

Thus, we can compute each of V_i $i \in (1, M)$. Now, using Eq.(2) gives $V_0 = (1 - \sum_{i=1}^{M} V_i)$, which shows that V_0 exists and can be determined by computing V_1 to V_M recursively. ■

Corollary III.2. (Stable Property) *After large iterations, the expected fraction of zeros in IBF-2C is a constant.*

Proof: In each iteration, each cell of the SBF has a certain probability of being set to M or to M/2 by the item hashed to that cell. For a fixed distribution of input data, the probability that a particular cell is processed in each iteration is fixed. Therefore, the probability of each cell being set is fixed. Also, the probability that an arbitrary cell is decremented by 1 is also a constant. Now, by theorem III.1, the probabilities of all cells in the SBF becoming 0 after N iterations are constants, for sufficiently large value of n. Therefore, the expected fraction of 0 in an SBF after n iterations is a constant. ■

Theorem III.3. (Stable Point (Average Case)) *When IBF is stable, the expected fraction of 0s in the IBF is* $m \times V_0$.

Proof: Since we assume uniform distribution for data and importance values, each cell has the same limiting probability of being set zero. Thus, on an average, when IBF-2C has m cells, the fraction of cells having zero values is $\sum_{i=0}^{m} Pr(\text{cell i is zero})$, which is $m \times V_0$. ■

In IBF, there could be two kinds of errors: false positives (FP) and false negatives (FN). A false positive happens when a distinct element is wrongly reported as duplicate. A false negative happens when a duplicate element is wrongly reported as distinct. We call the respective probabilities as the false positive rate (FPR) and false negative rate (FNR).

Corollary III.4. (FP Bound When Stable) *When IBF is stable, the FP rate is constant and no greater than FPR,* $FPR = (1 - V_0)^K$

Proof: If $V_{j,0}$ denotes the probability that the cell $IBF[j] = 0$ when IBF is stable, the upper bound on FP rate is $FPR = (\frac{1}{M}(1-V_{1,0}) + ... + \frac{1}{M}(1-V_{M,0}))^K$, i.e., $FPR = (1 - \frac{1}{M}(V_{1,0} + ... + V_{M,0}))^K$ Note that $\frac{1}{M}(V_{1,0} + ... + V_{M,0})$ is the expected fraction of 0s which is V_0 in average case. ■

False negative rate: False negative rate is related to input data distribution. As in [12], we define a gap to be the number of elements between a duplicate and its nearest predecessor. Suppose a duplicate element x_i whose nearest predecessor is $x_{i-\delta_i}$ is hashed to K cells. A FN happens if any of those K cells is decremented to 0 during the δ_i iterations when x_i arrives. From the Markov chain, we can calculate the probability, $Pr_0(\delta_i)$ that a cell is decremented to 0 from $M/2$ or M within the δ_i iterations. With this, the probability that FN occurs is given by $Pr(FN_i) = 1 - \prod_{j=1}^{K}(1 - Pr_0(\delta_i))$.

B. IBF-Multi Class (MC)

We now briefly describe generalization of IBF-2C, IBF-MC. In IBF-MC, as per the importance of the data item, we set the value of a cell to a value between 1 and M. For items with highest importance, the cell is set to M, whereas for items with lowest importance, the value will be set to 1.

Importance-aware Bloom Filter-M-Class (IBF -MC) An IBF-MC is defined as an array of integers $IBF[1]$, ... , $IBF[m]$. The minimum value of each cell is 0, and the maximum value is M. The update process follows Algorithm 2 with K independently and uniformly distributed hash functions. Each cell of the array is allocated d bits; the relation between M and d is $M = 2^d - 1$.

The steps in Algorithm 2 are similar to Algorithm 1 except when we set the cells. In Algorithm 2, the K cells are set to value between 1 and M based on the importance of the item.

Analysis: Similar to analysis of IBF-2C, we now analyze IBF-MC. We assume that the data items as well as the importance values are uniformly distributed. We then model the cell value of IBF-MC as a discrete time Markov chain as shown in Fig. 2. In each iteration or time step, the cell

```
Input: A sequence of numbers
Output: A sequence yes or no corresponding to each
        input number
Initialize IBF[1],...,IBF[m] to 0.
foreach number x_i do
  | Probe K cells IBF[h_1(x_i)],...,IBF[h_K(x_i)].
  | if none of the above K cells is 0 DupFlag = yes
  | else  DupFlag = no
  | Select P cells uniformly at random.
  | foreach each cell IBF[j] ∈ { IBF[j_1],..., IBF[j_p] }
  | do
  |   | if IBF[j] ≥ 1 then
  |   |  | IBF[j] = IBF[j] - 1
  |   | end
  | end
  | foreach each cell ∈ IBF[h_1(x_i)],...,IBF[h_K(x_i)] }
  | do
  |   | if IBF[h(x_i)] < f(imp(x_i)) then
  |   |  | IBF[h(x_i)] = f(imp(x_i))
  |   | end
  | end
  | Output DupFlag.
end
```

Algorithm 2: Approximately Detect Duplicates using IBF.

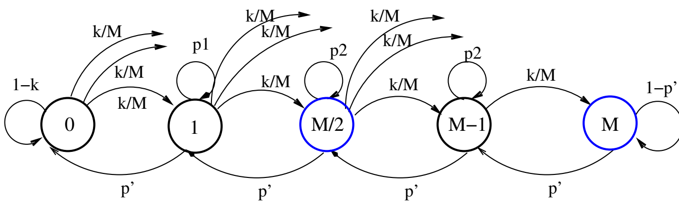

Fig. 2. IBF-MC: Importance-aware Insert, Random Delete

value is either decremented by 1, or set to a value $\in (1, M)$, or it remains the same. These conditions are captured in transition probabilities in the Markov chain shown in fig. 2. This Markov chain is indeed a valid discrete time Markov chain with irreducibility (every state is reachable from every other state) and aperiodicity (period of recurrence for each state is 1). If the cell is set, it goes to state i with probability k/M. Otherwise, the cell remains in the same state.

Theorem III.5. *Given an IBF-MC of m cells, if in each iteration, a cell is decremented by 1 with a probability p and set to importance value $imp \in \{1, M\}$ with a probability k, the limiting probability of a cell becoming zero exists.*

Proof: The existence of probability is straightforward from the Markov model of IBF-MC. Since Markov chain is stable, each state will have a non-zero limiting probability. Thus, there is non-zero probability that a cell will have its value zero in limiting cases. We omit the details of the probability calculation due to lack of space. The probabilities can be calculated in same fashion as described in proof of Theorem III.5. ■

Corollary III.6. (Stable Property) *After sufficiently large iterations, the expected fraction of zeros in IBF-MC is a constant.*

Proof: Similar to argument in corollary III.2. ■

Theorem III.7. (Stable Point (Average Case)) *When IBF is stable, the expected fraction of 0s in the IBF is $m \times V_0$.*

Proof: Similar to argument in Theorem III.3. ■

Corollary III.8. (FP Bound When Stable) *When IBF is stable, the FP rate is constant and no greater than FPR, $FPR = (1 - V_0)^K$*

Proof: Similar to argument in III.4. ■

The false negative rate can be calculated in similar way as in IBF-2C.

C. Time complexity

Theorem III.9. Time complexity *Given that K, P, and M are constants (set in IBF a-priori), the processing of each data element in the input stream take $O(1)$ time, independent of the size of IBF and the length of the stream.*

Proof: The time cost of our algorithm for handling each element is dominated by K, P and M. Within each iteration, we firstly probe K cells to detect duplicates, and then pick P cells to decrement by 1. Then we set K cells to a value between 1 and M. Since K, P, and M are constants, the time complexity is $O(1)$. ■

IV. Experimental Evaluation

In this section, we compare IBF-2C, IBF-MC with SBF and BF. Our overall goal is to see how these variants behave when different data items have different importance. Thus, for importance aware comparison, we define two new metrics– weighted false positive and weighted false negatives. We also track FP and FN rates for a fixed importance value, across all importance values. For evaluation, we consider two different data sets, one taken from a real-world trace of accessing video objects from Youtube, and other generated synthetically. To make the results independent of the sequence of data items, we repeat the experiments by taking different permutations of the data streams.

A. Experiment parameters

Setting M and m. Given fixed storage space SS, we have $SS = \log_2(M) \times m$. M is also function of the range of $\mathbb{I}$. Further, higher value of M entails higher values of K and P to maintain the same false positive and false negative rates, which may increase the computational cost per iteration. To set the appropriate value of M, we set a fixed value of SS, K and P and experiment with $M = \{3, 7, 15\}$. We observe that $M = 7$ trades the false positive and false negatives rates well to keep the sume of the rates lower as compared to the other two options, $M = \{3, 15\}$. We thus choose $M = 7$. With $M = 7$, the importance values are logically partitioned into 8 classes, and an importance value requires only 3-bits per cell. However, we wish to note that $M = 7$ may not result in best performance for IBF for different data streams.

TABLE II
AGGREGATE BASED EXPERIMENTS: YOUTUBE DATA

	Dataset 1				Dataset 2			
Algorithm	FP	FN	wFP	wFN	FP	FN	wFP	wFN
BF	34.81	0	35.29	0	34.81	0	34.17	0
SBF	25.83	0.84	26.23	0.87	26	0.9	25.55	0.81
IBF_{2C}	16.14	2.49	16.34	1.99	10.16	3.97	9.95	2.91
IBF_{MC}	5.19	8.91	5.23	4.76	0.84	14.01	0.87	7.52
IBF_{H}	9.08	13.25	9.39	10.43	0.12	16.07	0.07	10.34

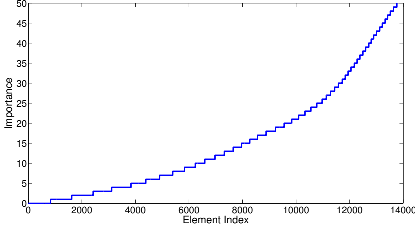

Fig. 3. Data Distribution: Youtube Data 1

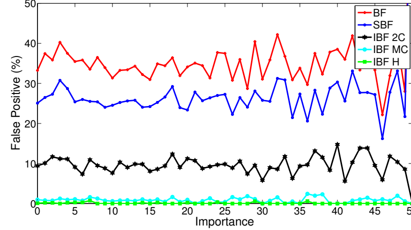

Fig. 4. False Positives: Youtube Data 1

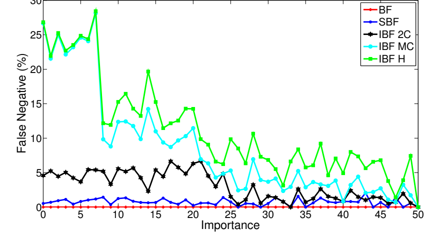

Fig. 5. False Negatives: Youtube Data 1

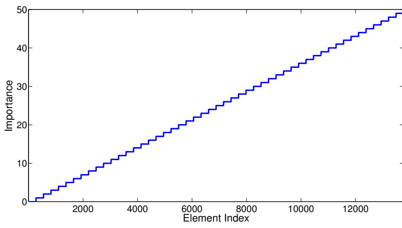

Fig. 6. Data Distribution: Youtube Data 2

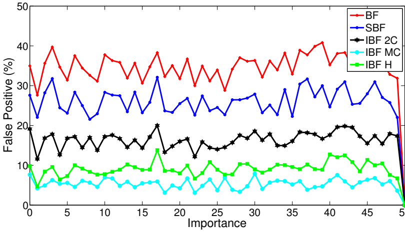

Fig. 7. False Positives: Youtube Data 2

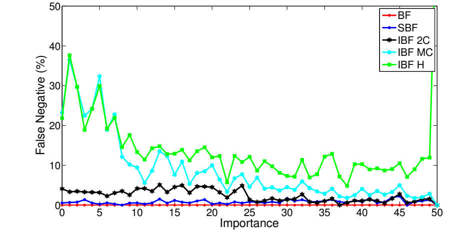

Fig. 8. False Negatives: Youtube Data 2

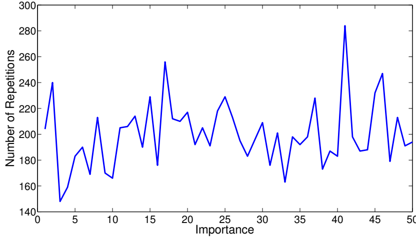

Fig. 9. Data Distribution: Synthetic Data 1

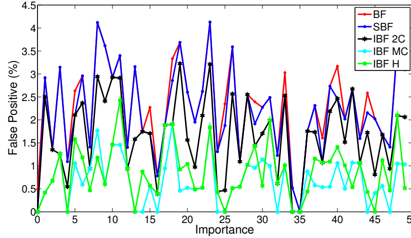

Fig. 10. False Positives: Synthetic Data 1

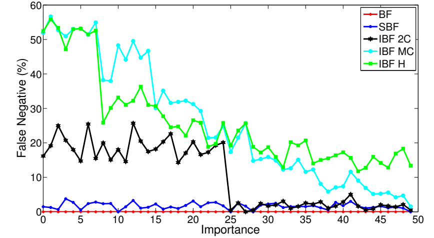

Fig. 11. False Negatives: Synthetic Data 1

Our primary objective, in setting $M = 7$ which gives $\frac{SS}{\log(M)}$ cells, is to compare IBF with SBF and BF for a fixed storage space SS. We consider it advisable to experimentally verify the appropriate value of M.

Setting K and P. Our theoretical analysis gives a handle to observe the behavior of IBF-2C and IBF-MC by different values of input parameters. By using theoretical equations, we varied the values of K and P, and observed the false positive and false negative for a set of bench marks. For example, we select $K = 5$, and $P = 10$ based on the result of theoretical equations which upper bounds the the false positive rate to our bench mark of 20%.

We consider total space available (SS) to be 16KB. Thus, with 3 bits per cell, we have about 42.6K cells in IBF. Tab. III summarizes other experimental parameters.

TABLE III
EXPERIMENTAL PARAMETERS

Storage space SS	16KB
Maximum value of a cell	7
No. of hash functions K	5
No. of cells to delete P	10

B. Performance metrics

Weighted False Positives and Weighted False Negatives: The “impact” of false positives and false negatives for important items is more than that of a less important item. Thus, if a false positive or false negative occurs for an *important* element, a greater penalty is paid by the application. This is well captured if we *weight* the occurrence of the false positives and false negatives with the importance of the element for which the FP/FN occurs. Thus, given n queries, we define weighted-false positive rate (wFP) and weighted-false

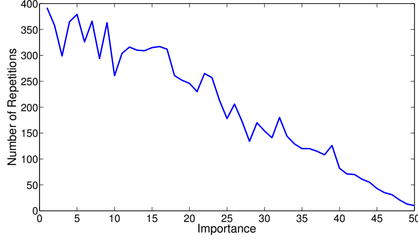

Fig. 12. Data Distribution: Synthetic Data 3

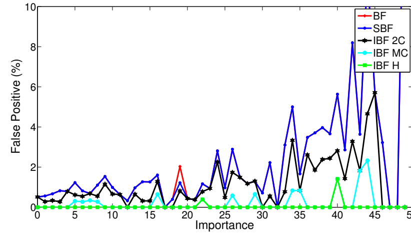

Fig. 13. False Positives: Synthetic Data 3

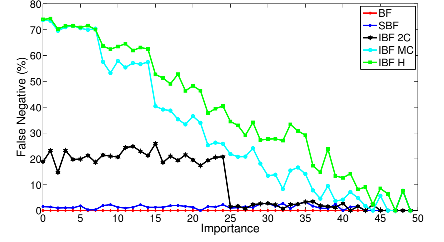

Fig. 14. False Negatives: Synthetic Data 3

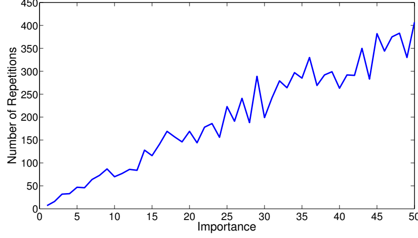

Fig. 15. Data Distribution: Synthetic Data 2

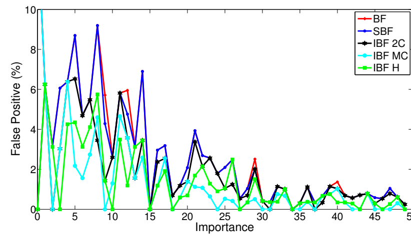

Fig. 16. False Positives: Synthetic Data 2

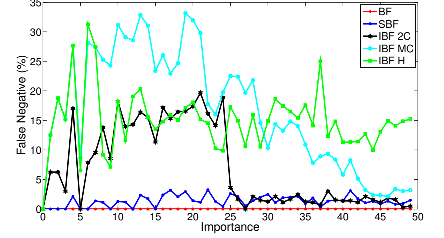

Fig. 17. False Negatives: Synthetic Data 2

negative rate (wFN) as shown below: $wFP = (\sum_{k=1}^{n} \beta \times imp_k)/(\sum_{k=1}^{n} \times imp_k)$ where β=1 if Q_k generates a FP, otherwise β=0. $wFN = (\sum_{k=1}^{n} \beta \times imp_k)/(\sum_{k=1}^{n} \times imp_k)$ where β=1 if Q_k generates a FN, otherwise β=0.

FP/FN against importance: While wFP and wFN capture the overall effectiveness of a probabilistic data structure, they do not explain the effect of the parameters on individual data elements. We wish to observe the performance of the IBF against the importance of the data elements. Suppose the importance values of the elements are integers in the range $1.....M$. Then we define *Importance rate arrays(I)* for FP (I_{fp}) and FN (I_{fn}). $I_{fp}[i] = \frac{\sum_{imp=1}^{M} \beta}{\sum_{imp=i}^{M} 1}$ where β=1 if Q_k generates a FP, otherwise β=0. $I_{fn}[i] = \frac{\sum_{imp=1}^{M} \beta}{\sum_{imp=i}^{M} 1}$ where β=1 if Q_k generates a FN, otherwise β=0. The ith entry of the array I_{fp} thus gives the false positive rate among elements having importance i. Similarly, the ith entry of the array I_{fn} gives the false negative rate among elements having importance i. This is our primary metric of comparison.

C. Comparison among Bloom filters

Tab. IV-C shows different Bloom filter data structures we consider for comparison. IBF_H is another variant of IBF, where insertion is done similar to IBF_{MC}, but the deletion is performed with respect importance values mapped to a cell, instead of random delete. We call the deletion scheme as *Probabilistic Delete Val* where the probability of choosing a cell is inversely proportional to the *value of the cell itself*, except cells having 0 value. We use this algorithm, so that elements with high importance are not evicted, thus improving the false negative rates.

D. Data sets

Real-world data set: We use the traces for the video accessed from Youtube, a popular video-sharing website, for a campus network over several days from [17]. We use the IDs of the videos as our data elements, and a function of the video length as the importance. Such a scenario is conceivable for a cooperative proxy that buffers the videos. Thus, we represent the state of a proxy by a summary-cache in terms of IBF. The proxies exchange the IBF to create a combined state of videos cached in different proxies. Since the overhead for fetching a longer video from the network is higher, the penalty for false negatives is high. Hence, we assign higher importance to longer length videos. For evaluation of IBF, we assume two cooperative proxies C_1 and C_2 and view the performance of IBF shared by C_1 with C_2. Thus, C_1 constantly updates and sends its IBF to C_2 which, for a given query, first searches its own cache and then the IBF of C_2.

	Combination
BF	*M-insert with M=1*
SBF	*M-insert + Random-delete*
IBF_{2C}	*M/2-insert + Random-delete*
IBF_{MC}	*Imp-insert + Random-delete*
IBF_H	*Imp-insert + Probabilistic-delete-val*

TABLE IV
COMBINATIONS OF INERT AND DELETE PROCEDURES FOR THE IMPORTANCE-AWARE BLOOM FILTER.

Synthetically generated data set: We generate synthetic data, with {10%,30%,50%} of the elements repeating, and the importance of the elements randomly generated between 1 and 50. In order to determine what kinds of data distributions IBF will be most useful on, we generate three kinds of data distributions. In the first distribution, each element occurs in the data stream with equal probability, irrespective of the importance (See Fig. 9). In the second and third distribution, the probability of occurrence of an element is directly (Fig. 15) and inversely (Fig. 12) proportional to its importance respectively. Thus, in the second dataset, important elements occur more frequently, and in the third one they occur less frequently.

E. Results

We now explain the behavior of Bloom filters in Tab. IV-C by giving stream of elements from synthetic data set, as well

as youtube video data set as input. We calculate weighted false positive and weighted false negative, along with the FP and FN for elements with a particular importance values, for all importance values.

Youtube data set: In our first set of experiments, we assign the importance value for videos directly proportional to its length, scaling the importance values to the range 0-49. However, due to the presence of a few outliers of extremely long length, this distribution is skewed. Thus we assign the importance value proportional to importance for the bottom 90 percentile in the range 0-49, and capped the importance value of the very long videos at an importance value of 50. This leads to the distribution of importance amongst elements as shown in Figure 3. Figures 4 and 5 show the false positive and false negatives for this dataset.

Summary. Note that, IBF has up to 30% less false positives for almost all data items in comparison to SBF. At the same time, IBF has the same or lower false negatives for items with higher importance. That is IBF has lower false positives and lower false negatives for items with higher importance. Notice that in both Imp-Insert and $\frac{M}{2}$-Insert, a trend of decreasing FN rates with increasing importance can be observed. This is consistent with what we expect from the theory developed in Section III where items with lower importance may be evicted from IBF due to deletion operation although such items are present in main storage. Overall, the result on real data set confirms our belief that IBF is indeed a practical solution to handle indexing and membership queries while taking into account importance values of the data items.

In our second set of experiments, we assign the importance value based on the percentile of the video length. Thus, there are an equal number of videos for all importance values. The distribution of importance values, and false positives and false negatives are shown in Figures 6, 7 and 8 respectively. The results are similar to those observed in first set of experiments, with both M/2 Insert and Imp-Insert distinctly performing better than SBF to index data items with higher importance.

In all of the above experiments with IBF-H, a variant of IBF-MC, behaves almost similar to IBF-MC. Our objective was to evaluate a heuristic IBF deletion scheme where the deletion is proportional to importance values mapped to cells. However, this deletion scheme does not seem to be effective in comparison to random delete operation. We wish to investigate this behavior in the future.

Synthetic data set: Fig. 10 and Fig. 11 show false positive and false negative for different importance values with respect to distribution of items in Fig. 9. We can observe that with respect SBF, IBF has lower false positives for almost all the importance values, and at the same time, it has about the same or lower false negatives for items with more importance. This matches with the overall goal we had set initially. When important elements occur less frequently, however, as we observe in Figures 12, 13 and 14, BF and SBF are not suitable to keep low FP and low FN for more important elements in the data stream. As seen in Fig. 13 and Fig. 14, FP rates go up as importance increases. This is undesirable behavior for elements with higher importance. Notice, however, that IBF-MC, results in low false positives, and low false negatives

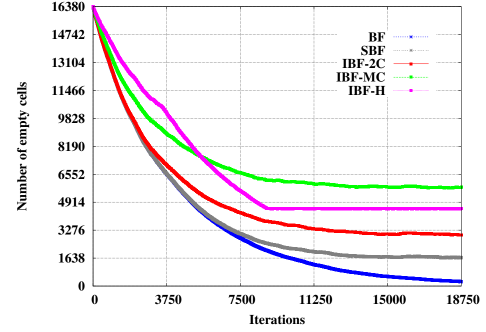

Fig. 18. Number of cells with value zero

for elements with high importance. IBF-H behaves similarly, but with higher false negatives than IBF-MC. IBF-2C shows clear distinction between 2 classes - those of higher and lower importance in false negatives. For importance values greater than 35, IBF has about 4% lower false negatives, and about 8% lower false positives than SBF.

Now, as observed in real data sets, it is expected that there are comparatively less number of items with higher importance. However, it is interesting to see how IBF behaves when there are more number of important items with higher importance. *Does IBF match performance of importance-agnostic SBF?* As can be seen in Fig 15, Fig. 16 and Fig. 17, IBF performs as good as SBF in maintaining false positives and false negatives for higher importance values. Thus, IBF can handle different data distributions, and still achieve our goal of capturing data semantics.

In Tab. II, we show wFP and wFN values for both sets of experiments. For dataset 1, IBF-MC (for example) has several fold lower FP and wFP values than SBF. SBF has wFP of 26% whereas IBF has wFP of only 6%, thus resulting in 4-fold better performance. At the same time, the increase in false negatives is marginal, SBF has wFN of 1% whereas IBF has wFN of 5%. Similar behavior can be observed for dataset 2. This shows that IBF captures data semantics well, and results in low wFP and wFN.

F. Theory verification: expected number of '0' cells

To verify the results of our experiments with that of theory, in Fig. 18, we plot the number of zeros observed in each variant of Bloom filter with respect to number of iterations of inserting data items. We feed the same parameters (for example, in this case $M = 7$, $K = 5$, $P = 10$) to our theoretical models which then give the expected fraction of empty cells analytically, and then confirm that indeed the two values are similar. Further, from Fig. 18, it can be seen that BF eventually gets 'full' with increasing number of iterations, whereas SBF and IBF stabilize to a particular value. This confirms the stable property of IBF through experiments.

V. Related Work

The Load shedding [20] technique is used in scenarios where limited memory is available to store streaming data. In such techniques [21], [18], [19], older tuples are evicted based on some load shedding scheme to make way for new data items, and then the entire data item is stored in memory. Our work is different because we do not store the entire data

in memory, instead we use hash functions to represent it in a Bloom filter. It is easy to see that large data items will quickly fill up the memory, hence a load shedding scheme will not be useful, especially when window sizes are also large. An advantage, however, over our approach is that load shedding does not introduce false negatives.

Data-importance is studied for join queries in [16], but tuple eviction is based on load shedding. The Stable Bloom filter (SBF) discussed in [12] indexes streaming data, but the scheme does not consider data-importance. Moreover, SBF deletes cells at random, which we believe is too simplistic. Our work provides a more complete solution, motivated by the fact that streaming data items have importance semantics.

Counting Bloom filters [13] and Spectral Bloom filters [11] are not directly applicable to our problem, because they assume that the set cardinality is known (and hence the FP an FN rates can be optimized by tuning other parameters). We make no such assumption, and provide a solution that can deal with unbounded sets, and fluctuating data distributions. Aging Bloom filter [23] is an interesting idea to remove stale data when indexing a static set in the filter, however, this work also does not address data-importance, and assumes that older data ages (or becomes less important over time). We do not make any such assumption; in fact, all data items in the landmark window are relevant, and the data-importance is the factor that guides the insertion and deletion of items in the filter. Time-decaying Bloom Filter [10] is another work that maintains the frequency count for each item in a data stream, and the value of each counter decays with time. Again, this work also assumes that items in a data stream is time-sensitive, and does not address data-importance while maintaining the time-decaying counters. It is also worth noting that, as compared to all previous work, we measure an importance-centric metric: the weighted FP and FN rates, which is a more realistic way of assessing quality of the query result.

Our work is closest to L-priorities bloom filter (LPBF) [15], which introduces priorities into bloom filters. The LPBF approach divides the available storage space into a multi-dimensional bit space. Priority of an object decides the number of vectors to be used for insertion, deletion and query. The technique is shown to yield differentiated false positive rates based on priority. While we share similar goals, our approach, we use a common vector space for all objects. Nevertheless, a comparison of LPBF and IBF would be interesting to improve cognizance of priority based bloom filter usage.

VI. Conclusion

In this work, we presented IBF, a probabilistic data structure for indexing and querying a stream of data items with different importance values. IBF differs from most of the prior work in its consideration of importance values while answering set membership queries. IBF has several desirable properties, for instance, the number of empty cells in IBF remain constant, there exists an upper bound on the false positive rate irrespective of the size of the stream, and it has $O(1)$ time complexity to index an item and answer a query. Importantly, IBF produces lower false positives and lower false negatives for important data items. We evaluated IBF on synthetic as well as a real-world data set. Our results show that, indeed IBF is effective in taking data semantics into account, and considering the importance values of the data items, it results in 4-fold reduction in weighted false positives in comparison to SBF. Thus, we believe IBF is an efficient and promising solution to index a stream of data items. As part of future work, it is interesting to capture the relationship between m, K and P analytically. Also, it would be interesting to deploy IBF in an online environment (for example, as a fast cache in routers) and analyze its behavior.

References

[1] Etrade financial, http://etrade.com, 2007.

[2] Nyse euronext, http://www.nyse.com/, 2007.

[3] Pacific tsunami warning center, http://www.prh.noaa.gov/pr/ptwc/, 2007.

[4] Scottrade homepage, http://scottrade.com, 2007.

[5] Tao project, http://www.pmel.noaa.gov/tao/, 2007.

[6] Yahoo news, http://news.yahoo.com/rss, 2007.

[7] M. M. Astrahan and et al. System R: Relational Approach to Database Management. *TODS*, 1(2):97–137, 1976.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of Principles of database systems*, 2002.

[9] B. H. Bloom. Space/time Trade-offs in Hash Coding With Allowable Errors. *Commun. ACM*, 13, July 1970.

[10] K. Cheng, L. Xiang, M. Iwaihara, H. Xu, and M. M. Mohania. Time-decaying bloom filters for data streams with skewed distributions. *International Workshop on Research Issues in Data Engineering*, 2005.

[11] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2003.

[12] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the ACM SIGMOD international Conference on Management of Data*, 2006.

[13] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *IEEE/ACM Transactions on Networking*, pages 254–265, 1998.

[14] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3), June 2000.

[15] H.-S. Hu, H.-W. Zhao, and F. Mi. L-priorities bloom filter: A new member of the bloom filter family. *International Journal of Automation and Computing*, 9(2), April 2012.

[16] D. Kulkarni and C. V. Ravishankar. iJoin: Importance-Aware Join Approximation over Data Streams. In *Proceedings of the 20th international conference on Scientific and Statistical Database Management*, 2008.

[17] U. T. Repository. http://skuld.cs.umass.edu/traces/network/readme-youtube.

[18] N. Tatbul. Qos-driven load shedding on data streams. In *Proceedings of the Young Researchers Workshop, International Conference on Extending Database Technology (EDBT)*, 2002.

[19] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *Proceedings of VLDB*, September 2007.

[20] N. Tatbul, U. Cetintemel, S. Zdonik, M. Chemiack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proceedings of VLDB*, 2003.

[21] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *Procedings of VLDB*, 2006.

[22] K. Trivedi. Probability and Statistics with Reliability, Queuing, and Computer Science Applications. *John Wiley and Sons, New York, 2001*.

[23] M. Yoon. Aging bloom filter with two active buffers for dynamic sets. *IEEE Transactions on Knowledge and Data Engineering*, 2010.