An Importance-Aware Bloom Filter for Set Memebership Queries in Streaming Data

Ravi Bhoraskar

December 9, 2010

under the guidance of
Prof. Purushottam Kulkarni

Contents

1 Introduction **3**

2 Related Work **5**
- 2.1 Bloom Filter . . . 5
  - 2.1.1 Desription of Algorithm . . . 5
  - 2.1.2 Analysis of False Positives . . . 5
- 2.2 Stable Bloom Filter . . . 6
- 2.3 Dynamic and Scalable Bloom Filters . . . 6
- 2.4 Spectral Bloom Filters . . . 7
- 2.5 Aging Bloom Filters . . . 7

3 Problem Statement **8**

4 Metrics Used **9**
- 4.1 Weighted False Positives and Weighted False Negatives . . . 9
- 4.2 Precision and Recall . . . 9
- 4.3 More Granularity: FP/FN against importance . . . 9

5 The Importance Aware Bloom Filter **11**
- 5.1 Insertion Schemes . . . 11
- 5.2 Deletion Schemes . . . 12
- 5.3 Reinsertion . . . 13
- 5.4 The Heuristics . . . 14

6 Experimental Evaluation of IBF **15**
- 6.1 Datasets Used . . . 15
  - 6.1.1 Web Crawl Dataset . . . 15
  - 6.1.2 Artificial Datasets . . . 15
- 6.2 Parameters . . . 15
- 6.3 Aggregate Based Experiments . . . 16
- 6.4 Importance Variation of Error Metrics . . . 17
  - 6.4.1 Varying n . . . 17
  - 6.4.2 Varying p . . . 18
  - 6.4.3 Varying k . . . 18
  - 6.4.4 Varying M . . . 19
- 6.5 Counting the number of Empty Cells . . . 19
- 6.6 Summary of Experiments . . . 20

7 Discussion **21**

8 Conclusions **23**

1 Introduction

Data streams, unbounded sets of continuous data, pose unique challenges towards issues of indexing and membership queries. Queries over data streams of the nature, flagging stock-value fluctuations in stock update streams, detecting news headlines of interest in RSS feeds, object lookup over in a cache summary etc., need to consider large or unbounded data sets. Bloom filter, a probabilistic data structures with finite memory requirements, is a useful data structure to index and lookup stream of data items. The basic bloom filter mechanism has been extended to adapt to the continuous indexing and querying requiring of unbounded data. While all extensions consider the unbounded constraint, we argue that another characteristic, *data-importance*, is an important factor towards indexing and querying. The notion of *data importance* is related to application-specific priority of different items of a data set—some stocks are more important that others, *cost* of fetching objects from a cache are directly related to their size etc. As part of this work, we make a case for the need to develop solutions that are importance-aware and towards this we propose *Importance -aware Bloom Filter* (IBF). As part of IBF, we provide a set insertion and deletion heuristics to make the bloom filter *importance-aware*. Our comparison of IBF with other bloom filter-based mechanisms shows that IBF performs very well—it has low false positives, and false negatives that occur due to deletions decrease with data-importance. Our precision and recall measurements that reflect the metric in terms of data-importance are also encouraging, and we believe that IBF provides a practical framework to balance the application-specific requirements.

There exist numerous challenges in storing and querying a data stream. Our work focuses on indexing an unbounded stream of data items to answer *set membership* queries. Since data streams are continuous and unbounded, it is impractical to store the entire data during query processing, especially so in finite-memory systems. In such contexts, it is difficult to provide precise answers to queries. To limit the amount data to be stored, or to be processed per data item, time-based *sliding window* [1] techniques have been used for *continuous queries* (CQ) [1]. A continuous query executes over successive instances of the sliding time-window. In our work, we consider a *landmark window*, where one end of the time-window is fixed, for example start-of-day, and the other end of the window is unspecified. With this setting, the system would index all or partial data items and answer membership queries.

Another important observation that motivates our work is the notion of *importance* of data items (or tuples). The following examples introduce *data-importance* and motivates the need to be importance-aware during indexing, querying and measuring performance of query accuracy. Application usage-scenarios:

Scenario 1: Consider a portfolio of stocks and an *importance* related to each. Possible importance functions can be based on individual or combinations of stock values, number of stocks, past and predicted fluctuations etc. Over the period of a day, the value of each stock fluctuates. An user is interested in knowing which stocks changed by more than 10% of its opening value. A less accurate answer regarding *less* important stocks is tolerable, whereas an inaccurate answer on *more* important stocks can result in sub-optimal actions to maximize portfolio value.

Scenario 2: A *webcrawler* starts crawling from a particular web page, collects links on that page, and then crawls these links recursively, similar to a breadth-first-traversal of a graph. It is obvious that a some links are popular and will appear on multiple web pages, e.g., a BBC news article may appear on international news listing of several web-sites. Crawlers need to accurately identify such popular links and avoid crawling them multiple times. An importance function in such cases can be correlated with the popularity of a web page or a web-site. A crawlers expectation of accuracy regarding the question of whether a page has been previously crawled can be related to its importance— popular pages require more accurate answers.

Scenario 3: Consider a data structure associated with a cache that can be looked-up to determine if an object is present in the cache. Objects stored in the cache are of different sizes and the *cost* to fetch them from a remote server is a function of their size. Assuming the look-up service provides a probabilistic answer, a requirement that aims to minimize *cost* would be less tolerable to false negatives for larger objects as compared to objects with smaller sizes (less important objects).

These examples motivate us to look at schemes that exploit data importance during indexing

and query evaluation. We believe that the assumption that all data items have the same importance level is too simplistic and not universally applicable in practice. Our work focuses on developing an indexing scheme assuming data items with *different* importance levels.

This work assumes that available memory is limited, hence it is not possible to store an entire unbounded data set. Our goal is to store an approximate summary using a hash-based index, popularly known as the Bloom Filter (BF). 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 on Stable Bloom Filters (SBF) [4] limits false positives by clearing cells of a bloom filter over time. Although, SBF tries to limit the false positive rate, it is agnostic to data importance— both during insert and delete operations. As we show later, exploiting the importance-semantics during indexing, and deletion of items would provide better query performance.

Our approach, Importance-aware Bloom Filter (IBF), is a modification of the traditional Bloom Filter which incorporates importance-aware semantics for insertion and deletion and operates on an unbounded data set. Apart from basic modifications to the Bloom Filter operations, we also examine schemes that balance the false-positive, and false-negative requirements of an application. The key idea is to periodically delete some cells (hence making way for more important items), and simultaneously reduce impact on false-positives and false-negatives.

As part of this work we make the following contributions:

- We develop a framework to index data streams into an importance-aware Bloom Filter.
- We propose importance-aware heuristics to index and query data items
- We evaluate and compare performance of these heuristics to provide importance-sensitive performance in terms of false positives and false negatives.

2 Related Work

Since traditional techniques (for indexing and storing) used with relational databases are not directly applicable in the context of data streams, there has been a significant research in building Data Stream Management. The major challenge with Data Streams as opposed to conventional RDBMS is that the size of the input set is unbounded. Hence, in general, a fixed amount of space cannot be allocated per record. In the scenario where the input set is extremely large as compared to the space available, approximate data structures seem to be an attractive option.

2.1 Bloom Filter

The Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. False positives are possible, but false negatives are not. Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

2.1.1 Desription of Algorithm

An empty Bloom filter is a bit array of m bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the m array positions with a uniform random distribution.

To add an element, it is fed to each of the k hash functions to get k array positions. The bits at all these positions are set to 1.

To query for an element, it is fed to each of the k hash functions, and the value is checked at the k array positions given. If at least one of these values is 0, then the element is not present in the set, and a negative response is reported. If all of the values are 1, then a positive response is generated. Note that a false positive is possible, since these k positions may have been set by during the insertion of some other elements.

Removing an element from a Bloom Filter is impossible, since resetting any of the k positions might lead to a false negative on a subsequent query of another element, which is prohibited by the no false-negatives property of the bloom filter.

2.1.2 Analysis of False Positives

Note that as we add more and more elements to the Bloom Filter, the probability of any of the m bits being set increases, thus increasing the probablity of false positives. Formally, after the insertion of n elements, the probability of a cell being 1 is

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \tag{1}$$

Thus, if an element is not present, the probability of all k cells given by the hash functions being 1, and thus the probability of a false negative, is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{2}$$

This can be approximated to

$$\left(1 - e^{\frac{-kn}{m}}\right)^{k} \tag{3}$$

Hence, as n increases, this term approaches 1, and the probability of getting a false positive increases. In the worst case, all m bits of the bloom filter are set to 1, and the bloom filter represents the universal set. In other words, the bloom filter gives a positive response for all values. This is clearly a problem for unbounded data streams, since potentially an infinite number

of elements may get inserted. One way to overcome this problem is to periodically *flush* the bloom filter, i.e. make all cells 0. This however, would compromise the no false-negatives propery.

The Bloom Filter is agnostic to *Data Importance*. As more and more elements are inserted, the BF remembers *all* elements inserted so far. This compromises with the false positive rates of important elements. The scheme that we propose evicts some elements periodically. Though this introduces false negatives, it ensures that the false positive and false negative rates for important elements remain low.

2.2 Stable Bloom Filter

Stable Bloom Filter [4] is a variant of the Bloom Filter for streaming data. The idea is that since there is no way to store the entire history of a stream (which can be infinite), Stable Bloom continuously evicts the stale information to make room for those more recent elements.

A Stable bloom is an array of 'm' integer $SBF[1], ..., SBF[m]$ whose minimum value is 0 and maximum value is MAX, where MAX is dependent on number of bits used to represent an integer. Each integer in the array is called as cell.

Thus, for inserting an incoming element into the Stable Bloom Filter, the following steps are performed.

- Probe k cells of the SBF to find whether the element is already present.
- Select p cells randomly from the m cells present.
- Decrement each cell by 1, if its value is positive.
- Now, for each of the k cells probed by the hash functions, set the value to MAX.

Thus, p decrements occur before the insertion of every element. This 'creates space' in the SBF for further incoming elements. [4] proves the *Stable Property of the SBF*, which states that.

After large number of insertion of items in the Stable BF, the probability that a cell in the Stable BF is 0 is fixed.

In other words,

$$lim_{N\to\infty} Pr\left(SBF_N = 0\right) \tag{4}$$

exists. A corollary of this property is that the expected fraction of zeros after N insertion is constant i.e. when large enough insertion of data item is made in stable BF, then the fraction of zeros constant. This ensures that whenever an element is inserted into the SBF, there is always 'space' for it.

However, the Stable Bloom Filter Algorithm is agnostic to *data importance*, since all cells are equally likely to be decremented. We borrow the idea of decrementing cells before an insertion from the SBF, and create an importance-aware version of it. Similarly, we modify the insertion algorithm of the SBF in order to incorporate data importance semantics in it.

2.3 Dynamic and Scalable Bloom Filters

The *Dynamic Bloom Filter* [6] and *Scalable Bloom Filter* [5] are variants of the Bloom Filter which is useful for storing sets of increasing cardinality.

For example, in a social networking application we would like to represent set of friends using bloom filter. Different users may have different number of friends in their profile. Thus, it becomes difficult to allocate single uniform filter size to each users since in some cases it may lead to increase in false positive while in other cases the memory is not utilized completely.

The Dynamic Bloom Filter uses the Standard Bloom Filter. However, during insertion when the algorithm senses that a BF is 'full', some new space is allocated for another bloom filter. Searching then involves applying the Bloom Filter Search algorithm on an array of Bloom Filters. The advantage of the Dynamic Bloom Filter is that since space is allocated dynamically, neither wastage of space, nor an excess of false positives occur.

The Scalable Bloom Filter uses a similar idea, but instead of appending a Bloom Filter of the same size, it appends a filter of varying size. It starts of with bloom filter of false positive P_0, once the bloom filter is filled to the capacity, a new filter is added to scalable bloom filter such that the false positive of new filter is $P_0 * r$, where r is called a tightening ratio $(0 < r < 1)$ .Thus, scalable bloom filter is series of bloom filters with false positive $P_0, P_0r, P_0r^2, ...$

Scalable bloom filter is useful, when we do not know how much space to allocate to bloom filter as the set size grows. We can start with small amount of memory for bloom filter and as the set size keeps growing we can add more memory. Thus, space utilization improves.

Although these ideas are interesting, they are not applicable in the given problem statement, since we assume that the space available for a single application is fixed and bounded. Thus, dynamically increasing the size of the bloom filter is not possible.

2.4 Spectral Bloom Filters

Spectral bloom filters [3] are useful for measuring multiplicity of data. Instead of using bits as the bloom filter does, they use integers. Whenever an element is added, the cell can either be set, or increased in value. Thus, multiplicity of an element in the bloom filter can be counted by measuring the minimum of the k values refered by the hash functions. Deletion also becomes possible, by decrementing the values of the appropriate.

However, for realistic prediction of the multiplicity of elements, the spectral bloom filter, and the Counting Bloom Filter [5] — that uses similar ideas — must assume that the set cardinality is known. This is not possible in our case, and thus these ideas are not directly applicable.

Further, given limited space, the only way in which we can store 'information' about the data elements is through the values of the cells in the bloom filter. We choose to use the cells to store information regarding the importance value of the elements rather than their multiplicity.

2.5 Aging Bloom Filters

These ideas are useful when data loses its validity, or importance as time progresses.

Consider a firewall application, where every packet hast to be tested against set of rules before being cleared to enter the network. The firewall would like to maintain a list of few IP address which have been denied access to the network. It may also not want that once the IP address is added to the list it is always denied access, since the IP address in the list of malicious user may be assigned to non malicious user by the ISP.

The aging bloom filter [8] uses a double-buffering technique. It uses two bloom filters, known as the Active Cache and the Warm-Up Cache. It fills the Active Cache till it is half-full. The subsequent elements are added to both the active and warm-up caches. When the active cache is full, the caches are swapped, and the warm-up cache is flushed. Thus, only the recent data is remembered by the Aging Bloom Filter.

Time-decaying Bloom Filter [2] is another work that maintains the frequency count for each item in a data stream, and the value of each counter decays with time.

The Aging Bloom Filter, and the similar time-decaying bloom [2] filter assume that the importance of data elements decrease with time. This may not be true in the general case. Our work makes no assumptions on the importance of the data, and leaves it to the application programmer to give relevant importance, as per the application, thus providing a greater flexibility.

3 Problem Statement

Let F (d_i) be an importance-function that maps some attribute of the data item d_i to a value imp_i, which represents the importance d_i. For simplicity we assume that items arrive in the following order $d_1, d_2, \cdots, d_n$, where n is the number of items that have arrived, so far. Assuming that all queries have a landmark window that starts at time before d_1 arrives, query Q is to be evaluated over all elements d_i, where $1 \leq i \leq n$ and value of n is unbounded.. We address the following problem.

Given a set of data items $D = \{d_1, \cdots, d_n\}$, an importance function F, and finite memory M, we index the items in D using F, so that membership of an item $d \in D$ can be determined in constant time.

Since our approach is to store the n items using a hash-based index structure of size M, the metrics of interest are number of false positives (FPs) and false negatives (FNs). Consider a query $Q(d)$, which tests for the membership of item $d \in D$. A false positive results when $d \notin D$ and $Q(d)$ returns true. A false negative occurs when a membership query returns $false$, although $d \in D$. Ideally, we would like to have minimum false positive and false negative rates. With IBF, our expectations of performance metrics are related to the importance functions—false positive and false negative rate of more important data items should be lower than less important ones. A relaxation, low false positive rate for all items but a importance-related false negative rate, avoids false "actions" (or cost) concerned with more important data items. This cost could be application-specific, for example, in terms of computational overhead, storage etc.

The metrics used, keeping in mind the performance requirements of the IBF as explained above, are explained in detail in section 4. But here is a brief description of the intuition behind the metrics.

- ***Aggregate Quantities*** The weighted false positive and weighted false negative are measures to capture the performace of an algorithm in a single number. We weigh the FPs/FNs with the importance of the elements they occur for. Thus a FP for an important element would affect the WFP more, and we would expect the WFP/WFN rates for IBF to be lower than SBF, even if the absolute FN/FP rates are similar.

- ***FP/FN against importance*** If we calculate the FN/FP rates with items having the same importance grouped together, and plot them againt importance, we would expect a downward sloping graph, since FPs and FNs for more important quantities are expected to be smaller. If the FPs are similar to the SBF, and FNs are sloping downwards, then we would expect a ***Crossover Point***, which is an importance value, below which SBF has lower FNs, but above which IBF does. We would like the crossover point to be as low as possible.

4 Metrics Used

4.1 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 *weigh* the occurence of the false positives and false negatives with the importance of the element for which the FP/FN occurs.

Given N queries over the set D, and F (d_i)=imp_i, the weighted-false positive rate (wFP) and weighted-false negative rate are defined as shown below:

$$wFP = \sum_{k=1}^{n} \beta \times imp_k \tag{5}$$

where β=1 if Q_k generates a FP, otherwise β=0.

$$wFN = \sum_{k=1}^{n} \beta \times imp_k \tag{6}$$

where β=1 if Q_k generates a FN, otherwise β=0.

Since, false positive and false negative rates are expected to be inversely proportional to the relative importance of data items, the weighted rates should be approximately similar.

4.2 Precision and Recall

Recall is the fraction of correct instances among all instances that actually belong to the relevant subset, while precision is the fraction of correct instances among those that the algorithm believes to belong to the relevant subset.

Since our heuristics provide approximate results, we also would like to quantify the measure using *precision* and *recall* [7]. Let $wTotal = \sum_{\forall i \in R} imp_i$, where R is the set of all data items i that generated a "hit" in the Bloom filter. The weighted versions of precision and recall are defined as below:

$$Precision = \frac{wTotal - wFP}{wTotal} \tag{7}$$

$$Recall = \frac{wTotal - wFP}{wTotal - wFP + wFN} \tag{8}$$

4.3 More Granularity: FP/FN against importance

All the metrics used above use a form of aggregate over the entire data. Although this provides a good way to view the effect of varying a parameter over the entire dataset, some loss of information is observed, since the effect of the parameter cannot be seen over individual data elements. We wish to observe the performance of the IBF against the importance of the data elements. Suppose the importace values of the elements are integers in the range 1.....Z. Then we define an array, the *Importance Rates Array*.

$$I_{fp}[i] = \frac{\sum_{imp=i} \beta}{\sum_{imp=i} 1} \tag{9}$$

where β=1 if Q_k generates a FP, otherwise β=0.

$$I_{fn}[i] = \frac{\sum_{imp=i} \beta}{\sum_{imp=i} 1} \tag{10}$$

where β=1 if Q_k generates a FN, otherwise β=0.

The i^{th} entry of the array I_{fp} thus gives the false positive rate among elements having importance i. Similarly, the i^{th} entry of the array I_{fn} gives the false negative rate among elements having importance i.

We plot a graph of I_{fp} and I_{fn} against their index numbers. Since the SBF is blind to importance of elements, we expect the graph to be close to a straight line parallel to the x axis. For the IBF, a desirable behaviour would be for the false positive and false negative to decrease as the importance increases. Thus I_{fp} and I_{fn} both sloping downwards, with the mean value close to the SBF is desirable. A relaxed condition would be for I_{fn} to be sloping downwards, and I_{fp} to be constant at a value close to the SBF line.

We formalize this intuitive notion by defining a quantity known as the ***Crossover Point***. The intuition behind it was explained in section 3. When the FPs of SBF and IBF are close to each other, we define the crossover point to be

$$min_{i=0}^{maxImp} \left(IBF.I_{fn}[i] < SBF.I_{fn}[i]\right) \tag{11}$$

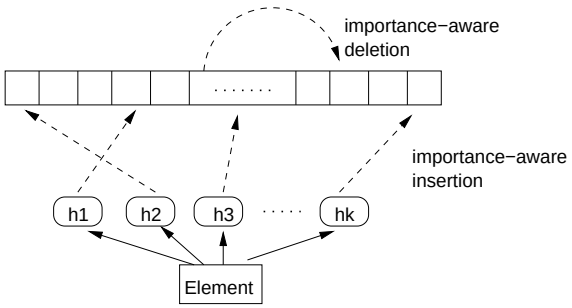


Figure 1: Importance-aware Bloom Filter.

5 The Importance Aware Bloom Filter

As shown in Figure 1, basic changes in the Importance-aware Bloom filter are modifications to the insert and delete procedures. As with traditional data item insertion, each data item is hashed to k cells using k hash functions and the value in each updated. As part of IBF, we update the value during insertion based on an importance function. Further, during periodic or insert-triggered deletion, which cells to be considered for deletion is in cognizance with the importance function. Since, importance-aware actions can be of different variety, we propose a set of heuristics for insertion and deletion operations as part of IBF.

Our basic IBF insertion policy as is follows, as part of the insertion operation, values of a subset of cells are first decremented (or reset to zero) and then the new element is inserted into the bloom filter. The deletion of elements before the insert operation is in line with the strategy followed by Stable Bloomfilters [4], to maintain false positive rates below a certain value. Either one or both, the insert and delete operations, are $importance - aware$. We present a set of possible insert and delete operations and then present combinations of these, which form importance-aware heuristics for IBF.

5.1 Insertion Schemes

Algorithm 1 M-insert(element)

```
1: delete()
2: for x ← 0 to k do
3:     hash = createHash(element,x)
4:     bitset[hash] ← M
5: end for
```

The $M - insert$ procedure uses a constant value M for insertion in the k hashed cells during insertion. The idea being inserts are of same value, but delete operations by importance will automatically delete less important elements.

Algorithm 2 Imp-insert(element)

```
1: delete()
2: for x ← 0 to k do
3:     hash = createHash(element,x)
4:     bitset[hash] ← Importance[element]
5: end for
```

The $Imp - insert$ procedure uses the importance of the element from a importance function $F(element)$ as the value to be inserted in the k hashed cells. Higher values of more important elements, will remain in the bloomfilter for longer, ensuring lower false negative rates.

5.2 Deletion Schemes

Algorithm 3 Random-delete()

```
1: for i ← 0 to p do
2:    index ← randomInt() % n
3:    if bitset[index] ≥ 1 then
4:       bitset[index] ← bitset[index] −1
5:    end if
6: end for
```

Random − delete chooses p cells uniformaly at random and then decrements value in the cell by one (if cell already not at zero). With uniform deletions and importance-aware insertions, the false negative ratios of more important elements can be kept low.

Algorithm 4 Probabilistic-delete(impArray, nDeletes)

```
1: for i ← 0 to nDeletes do
2:    index ← Cell chosen with probability proportional to 1/impArray
3:    if bitset[index] ≥ 1 then
4:       bitset[index] ← bitset[index] −1
5:    end if
6: end for
```

Probabilistic-delete is an importance-aware delete procedure, that associates the probability of choosing a cell for deletion to be inversely proportional to an importance value attached to each cell. An example usage of this function is, *Probabilistic-delete(totalImp, p)*, where $totalImp$ refers to the summation of importance values mapped to each cell till time of deletion. p denotes the number of cells to delete from the bloom filter. This algorithm deletes cells that belong to low importance data items more frequenctly than data items with higher importance.

Algorithm 5 Probabilistic-delete-p(impArray, nDeletes)

```
1: for i ← 0 to nDeletes do
2:    index ← Cell chosen with probability proportional to 1/impArray
3:    if bitset[index] ≥ 1 then
4:       bitset[index] ← bitset[index] −1
5:    else
6:       i = i + 1
7:    end if
8: end for
```

Probabilitic Delete - p is a variant of the probabilistic delete algorithm in which exactly p cells are always deleted — unless less than p cells are full (not reflected in the algorithm above to keep it simple).

Probabilistic Delete Val is a deletion scheme that is very similar to Probabilistic Delete - p. The crucial difference is that instead of referring the *impArray*, the probability of choosing a cell is inversely proportional to the *value of the cell itself*. Note that this algorithm implicitly assumes that the cell itself stores some information about the importance. Hence it is meaningless to use it with *M-insert*. It can only be used with *Imp-Insert*.

The *MinImp-delete* function takes as input an array of importance values (current or cummulative) corresponding to each cell and decrements the p least important cells. If a cell can not be decremented — if it is already zero — then another cell is chosen. Hence, at least p cells are always decremented — unless less than p cells are full (not reflected in the algorithm above to keep it simple).

Algorithm 6 Probabilistic-delete-val(impArray, nDeletes)

1: **for** $i \leftarrow 0$ to $nDeletes$ **do**
2: $index \leftarrow$ Cell chosen with probability proportional to $\frac{1}{cellValue}$
3: **if** bitset[$index$] ≥ 1 **then**
4: bitset[$index$] $\leftarrow$ bitset[$index$] -1
5: **else**
6: i = i + 1
7: **end if**
8: **end for**

Algorithm 7 MinImp-delete(impArray, nDeletes)

1: sort $impArray$ in descending order
2: **for** $i \leftarrow 0$ to $nDeletes$ **do**
3: index $\leftarrow$ cell correspoding to $impArray[n-i]$
4: **if** bitset[index] ≥ 1 **then**
5: bitset[index] $\leftarrow$ bitset[index] -1
6: **else**
7: i = i + 1
8: **end if**
9: **end for**

Algorithm 8 MinImp-delete-withPriority (impArray, nDeletes)

1: **for** $i \leftarrow 0$ to $nDeletes$ **do**
2: index $\leftarrow argmin_{i=0}^{n-1}$ $(impArray[i] + cellPriority[i])$
3: **if** bitset[index] ≥ 1 **then**
4: bitset[index] $\leftarrow$ bitset[index] -1
5: **end if**
6: cellPriority[index] $\leftarrow$ cellPriority[index] + M
7: **end for**

The *MinImp-delete-withPriority* extends MinImp-delete to increase the probability of non-deleted cells to increase slowly, so that they can be selected for deletion over often deleted cells.

5.3 Reinsertion

Another possibility that we considered was reinsertion of delted tuples back into the Bloom Filter after some interval. The rationale behind this is that if important elements are removed from the bloom filter prematurely, they should be inserted back, in order to avoid false negatives. A reinsertion cache is maintained, that contains elements with importance above a certain threshold. Thus, when an element arrives, it is inserted into both the bloom filter and the reinsertion cache. Elements from the reinsertion cache are inserted into the bloom filter according to some conditions.

The following strategies are considered for reinsertion

- ***FIFO Reinsertion*** The elements removed first would be reinserted first into the bloom filter.
- ***Random Reinsertion*** Out of all the elements in the reinsertion cache, one is chosen randomly to be reinserted into the bloom filter.

The condition for rehashing is also an important parameter to consider. The following conditions could be used

- ***Periodic*** Reinsertion is done periodically, i.e. after every fixed number of elements have been inserted into the Bloom Filter.

- ***Threshold*** Reinsertion is done when the fraction of zeroes in the bloom filter falls before a certain threshold.

Some experiments were conducted using reinsertion. However, the results were not encouraging, since the reinsertion cache required substantial space, and adding that space to the bloom filter itself provided better results. Hence the idea of reinsertion of elements into the bloom filter was abandoned.

5.4 The Heuristics

We considered several combinations of the Insertion and Deletion schemes mentioned above, and ran experiments on them. Table 1 provides a subset of the heuristics that we tried, that provides a good representation of the traits shown by different algorithms we tried. Section **??** presents the experiments carried out on these heuristics.

	Combination
BF	*M-insert with M=1*
SBF	*M-insert + Random-delete*
IBF_1	*Imp-insert + Random-delete*
IBF_2	*Imp-insert + MinImp-delete(totalImp, p)*
IBF_3	*M-insert + Probabilistic-delete-p(totalImp, p)*
IBF_4	*M-insert + MinImp-delete(totalImp, p)*
IBF_5	*Imp-insert + Probabilistic-delete-p(totalImp, p)*
IBF_6	*Imp-insert + Probablistic-delete-val(totalImp,p)*

Table 1: Combinations of inert and delete procedures for the Importance-aware Bloomfilter.

The bloom filter uses **M-Insert policy with** $M = 1$, and **no deletion** policy. Thus, as time progresses, it keeps getting more and more full, as explained in section 2.1. This algorithm is included in the experiments in order to compare the behaviour against the IBF. The FP rates are expected to be higher than SBF and IBF, and the FN rates to be zero. The stable bloom filter, as described in section 2.2, is a modification of the Bloom Filter agorithm for Data Streams. It uses the **M-Insert** and the **Random-Delete** policies. False positives are expected to be lower than Bloom Filter, and False negatives will exist. Since the algorithm is agnostic to data importance, the FP and FN rates will be uniform across importance value. IBF_1, IBF_3 and IBF_4 use a minor modification to the SBF. IBF_1 uses the **Imp-Insert** and the **Random-Delete** policies. This algorithm shall show the difference that is made by the Imp-Insert policy as compared to the M-insert policy. IBF_3 uses **M-insert** and **Probabilistic-Delete**. The insertion policy is same as the SBF, and it will show the effect of the Probabilistic-Delete policy.Similarly, IBF_4 uses the **M-Insert** and **MinImp-Delete** policies. IBF_2 uses the **Imp-Insert** and the **MinImp-Delete** policies. Since it uses both Importance-Aware insertion and Importance-Aware deletion, its performance is expected to be superior to any of the algorithms that use only Importance-Aware Insertion or Importance-Aware Deletion. IBF_5 uses **Imp-Insert** and **Probabilistic-Delete**. It too is expected to show the benefits of both of these Importance-Aware techniques on the result. IBF_6 uses **Imp-insert** and **Probabilistic-delete-val**. The major advantage of this algorithm is that it does not require additional state to store the cumulative importance of the cells. However, the importance information is conveyed only during insertion, and this may get distorted and even lost as cell deletion takes place.

6 Experimental Evaluation of IBF

The motivation behind conducting the experiments was three fold. First, to make use of the IBF on some data, and verify that its behaviour is as we expect it to be. Second, to find the set of parameters that would lead to the optimum behaviour of the IBF. Third, to compare the IBF with the Bloom Filter and the Stable Bloom Filter, and demonstrate that it indeed works well in applications where data importance semantics are important. We conducted the experiments on both real world and artificially generated datasets, measuring the metrics described in section 3.

6.1 Datasets Used

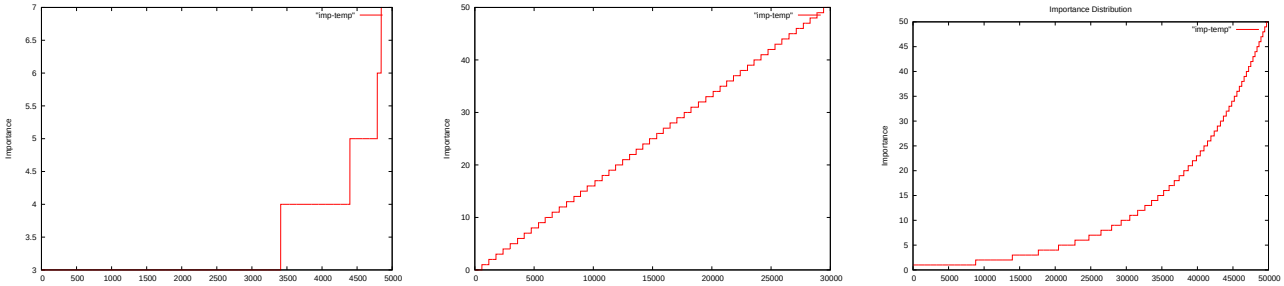
We generated and used several kinds of datasets for the experiments conducted. Both real-world as well as artificially generated datasets were used. The artificial datasets were constructed so as to mimic the real world data. Using artificial datasets gave us greater flexibility and freedom in choosing parameters such as dataset size, number of duplicates, etc.

6.1.1 Web Crawl Dataset

Our real world dataset was constructed by performing a web crawl. The crawl was started on `http://business.yahoo.com` and performed upto a depth of 7, spanning upto 6 URLs from every node. This gave us approximately 100000 URLs. The number of duplicates in this dataset was approximately 73 percent. The importance of an element was inversely related to the depth of the URL in the web-crawl. This is a reasonable assumption, since the relevance of a URL can be expected to go down as the distance from the source increases. Figure 2(a) shows the distribution of this dataset against the importance. Notice that there are a large number of tuples with low importance. This is expected since the web crawl grows as a tree.

6.1.2 Artificial Datasets

We used two kinds of artificially generated datasets. The linear dataset has a uniform distribution of elements across all importance values (Figure 2(b). The exponential dataset has more values of lower importance, and less values of more importance. When the index of the elements is plotted against their importance, the figure resembles an exponential curve(Figure 2(c). This mimics the behaviour of the Web Crawl dataset. We fixed the duplicates rate at 70*percent* in order to remain close to the real life dataset. A dataset containing $100,000$ elements was generated for each of the two kinds of artificial datasets.



(a) Web Crawl Dataset (b) Artificial Linear Dataset (c) Artificial Exponential Dataset

Figure 2: Distribution of Datasets

6.2 Parameters

Table 2: Parameters in BF experiments

k	Number of Hash Functions
n	Size of the Bloom filter
N	Size of the Dataset
p	Number of cells deleted
M	The MAX inserted in the BF

Table 2 summarizes the parameters that can be varied to conduct the IBF experiments. Increasing k may increase the FP rates, since now the bloom filter gets 'filled' sooner. On the other other hand, it may also decrease FP rates, since now testing for an element needs more cells to be checked, thus making a positive harder to be achieved. The trend of false negative would be opposite to that of the false positives. Increasing n would almost definitely decrease both FP and FN rates, since the probability of collision of cells of two different elements is now lower. Similarly, increasing N would almost definitely increase the FN and FP rates, since probability of collision is higher. However, this would also depend on the nature of the dataset, the percentage of duplicates present etc. However, keeping these constant FP and FN can be expected to increase with N. Increasing p would increase the FN rates, since elements are more rapidly evicted from the bloom filter. FPs would go down, since the likelihood of a cell being 'empty' is now more. Increasing M does the exact opposite, and FPs are expected to go up, and FNs are expected to go down when M is increased.

6.3 Aggregate Based Experiments

The Aggregate Based experiments are useful to see the overall effect of varying the parameters on the IBF results. We ran experiments for varying k, n, N, p, and M. We noted the values of WFP, WFN, precision and recall. The experiments were run on all datasets. For brevity, here we present the results of the linear(uniform) artificial dataset of size 100000.

Table 3: Aggregate Based Experiments: Weighted False Positives

	$n = 16384$		$n = 65536$		$n = 262144$	
Algorithm	$k = 4$	$k = 8$	$k = 4$	$k = 8$	$k = 4$	$k = 8$
BF	46.93	5.87	23.52	33.31	1.58	2.34
SBF	23.10	33.47	13.49	21.93	1.45	2.04
IBF_1	3.83	4.30	2.12	2.25	0.27	0.15
IBF_2	0.0	0.02	0.0	0.004	0.01	7.56E-4
IBF_3	7.64	6.02	4.77	5.22	0.49	0.70
IBF_4	3.48	5.40	1.45	2.75	0.13	0.22
IBF_5	7.74	49.90	4.74	5.15	0.48	0.68

Tables 3 and 4 show the Weighted False Positives and Weighted False Negatives values for several experiments we conducted. We notice an inverse correlation between the WFP and WFN values. For example, for IBF_2, the WFP is very low at close to 0.2 percent, but the WFN are high at close to 44 percent. This is expected, since FPs are higher when the array is more 'full', whereas FNs are higher when the array is more 'empty'. Thus, some of the factors that would cause FPs to increase would cause FNs to decrease, and vice versa.

The algorithms using IBF-delete policies (Probabilistic delete and MinImp delete) show lower FPs(3-7 percent) than those using random delete(around 20 percent). Since these algorithms

Table 4: Aggregate Based Experiments: Weighted False Negatives

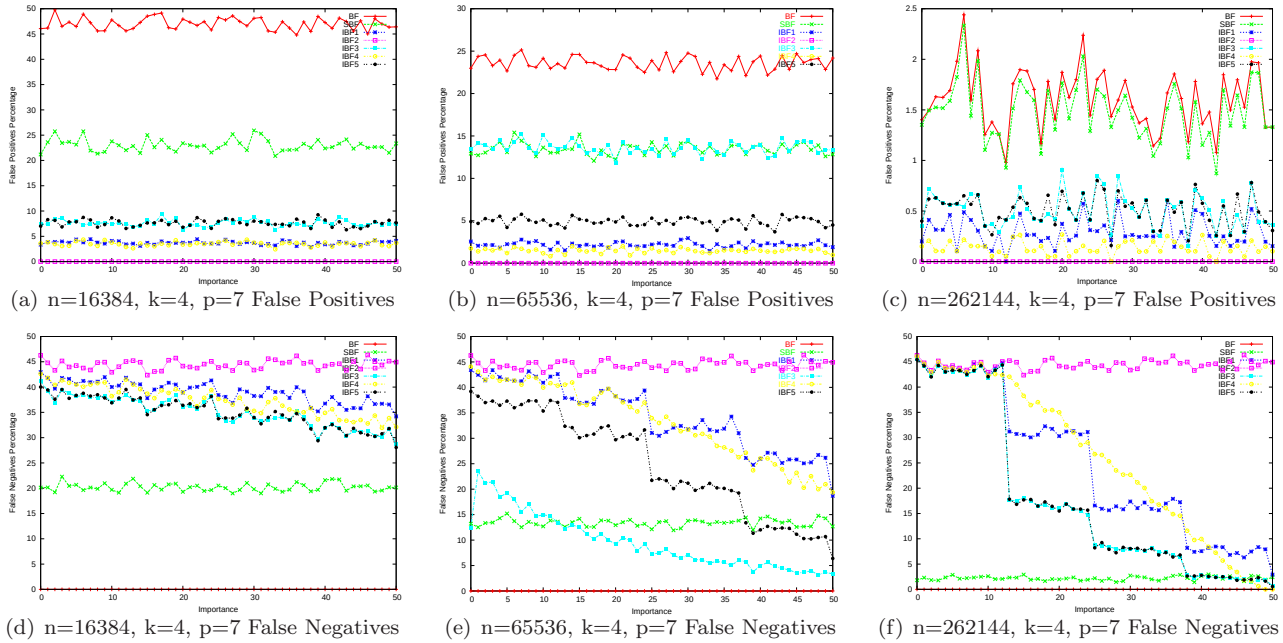
	$n = 16384$		$n = 65536$		$n = 262144$	
Algorithm	$k = 4$	$k = 8$	$k = 4$	$k = 8$	$k = 4$	$k = 8$
BF	0.00	0.0	0.0	0.0	0.0	0.0
SBF	20.04	13.42	13.56	22.93	2.24	2.67
IBF_1	37.89	38.15	30.69	10.43	16.43	19.99
IBF_2	44.53	44.27	44.53	32.55	44.53	38.14
IBF_3	33.45	36.22	19.11	43.26	8.90	4.40
IBF_4	35.72	33.92	28.91	22.87	16.99	9.50
IBF_5	33.50	36.30	19.20	26.09	8.77	4.48

compulsorily delete p cells, the bloom filter is more empty. Also, those using M-insert show greater false positives than those using Imp-Insert. This is probably due to the fact that M-insert always sets the cells to the MAX value, thus making it harder to decrement than Imp-Insert, which sets it to a value betwen 1 and MAX.

6.4 Importance Variation of Error Metrics

In these experiments, for a fixed set of paramters, we measure the false positive and false negative rates for different importance values. Note that this is a more fine-grained result than the Aggregate Based experiments, since more information is captured. However, the aggregate based experiments are also important, since they show us the effects variation of individual parameters on the IBF, and thus help to find the optimum set of parameters to be used for this experiment. Once again, the experiments were conducted for various datasets, but for brevity, we present the results of only the linear(uniform) artificial dataset of size 100000.

6.4.1 Varying n



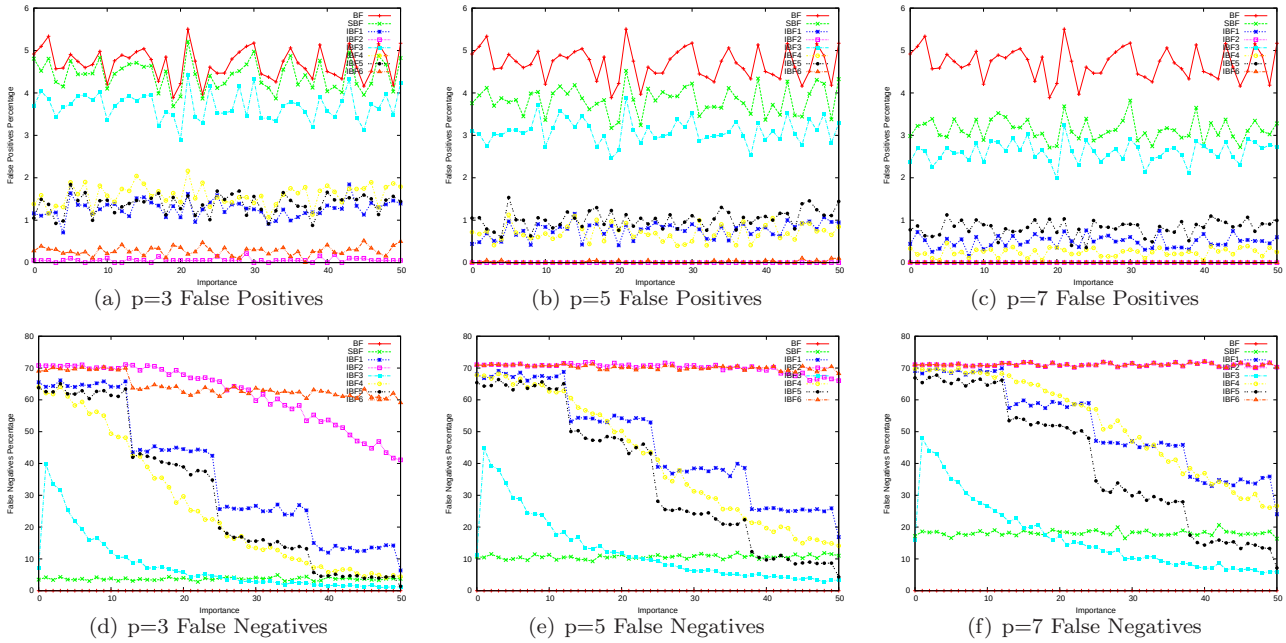
(a) n=16384, k=4, p=7 False Positives (b) n=65536, k=4, p=7 False Positives (c) n=262144, k=4, p=7 False Positives

(d) n=16384, k=4, p=7 False Negatives (e) n=65536, k=4, p=7 False Negatives (f) n=262144, k=4, p=7 False Negatives

Figure 3: Importance Rates Experiments (varying n)

Figure 3 shows the importance rates for some values of n,k and p for a dataset of size 100000. Notice that the false positive rates of the Bloom Filter is 3-5 times that of the IBF. This is easily explained, since the Bloom Filter does not perform any deletions on the array. The FP for the IBF algorithms do not vary much with importance, staying around 5 percent for $n = 65536$ whereas the false negatives fall sharply as importance increases, from 40 percent to 5 percent. This fall is more prominent when n is larger. For $n = 16384$, the FNs fall to 30 percent, whereas they fall to around 2 percent when $n = 262144$. This is probably because increasing n reduces the conflicts in the cells, and thus ensures that the IBF insertion and deletion algorithms do not inardvertantly benefit the non-important elements. This result is similar to what we expected, as described in section 3.

6.4.2 Varying p



(a) p=3 False Positives (b) p=5 False Positives (c) p=7 False Positives

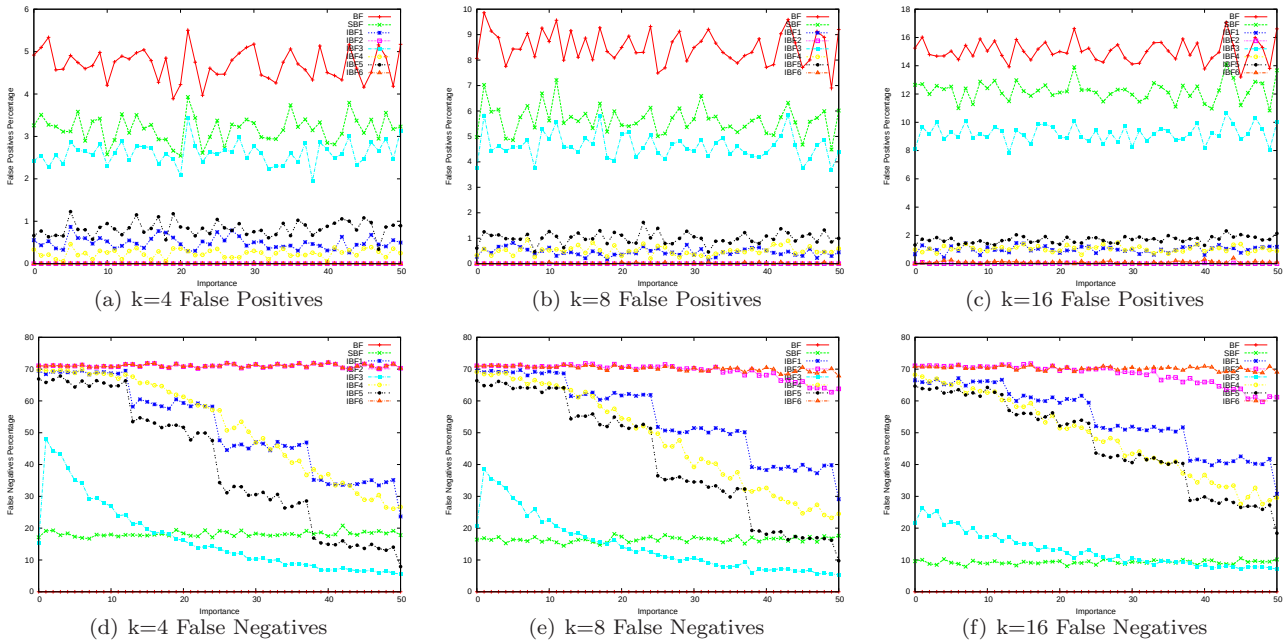
(d) p=3 False Negatives (e) p=5 False Negatives (f) p=7 False Negatives

Figure 4: Importance Rates Experiments (varying p) n=65536, k=4, M=4

Figure 4 shows similar experiments conducted for varying values of p. Increasing p causes the *False Negatives* to increase. For IBF_1, the average FN rate is around 40 percent for $p = 3$ and around 50 percent when $p = 7$ This is probably because with a greater p, more elements get decremented, thus making the probability of an element getting evicted larger. The *crossover point* for IBF_3 decreases with increasing p. For $p = 3$ it is 30, and for $p = 7$ is 15. Also the distance between IBF_3 and SBF increases with increasing p. It is only about 2% for $p = 3$ but increases to 12% for $p = 7$. Hence, the $importance - aware$ characteristics increase with a greater p. Please note, however, that the nature of the graphs do not change with p, and the shapes of the graphs look very similar for the three cases.

6.4.3 Varying k

Increasing k causes an increase in false positive rates — for IBF_5, it is around 0.8 percent for $k = 4$, and rises to 2 percent for $k = 8$. A decrease in false negative rates is also observed — for IBF_1, from 60 percent to 40 percent. We discussed in section 6.2 that increasing k has two opposite effects: one increasing FP and once decreasing it. From the experiment, we note that the effect of the bloom filter getting 'full' dominates over the more stringent checking condition



(a) k=4 False Positives (b) k=8 False Positives (c) k=16 False Positives

(d) k=4 False Negatives (e) k=8 False Negatives (f) k=16 False Negatives

Figure 5: Importance Rates Experiments (varying k) n=65536, p=7, M=4

due to increase in k, and thus an increase in k causes an increase in FP and decrease in FN. Note, however that the effect of varying k is not as dramatic as the effect of varying n.

Although the FN rates decrease on increasing k, it has an adverse effect on the *crossover point*. It drops from 18 for $k = 4$ to 27 for $k = 16$.

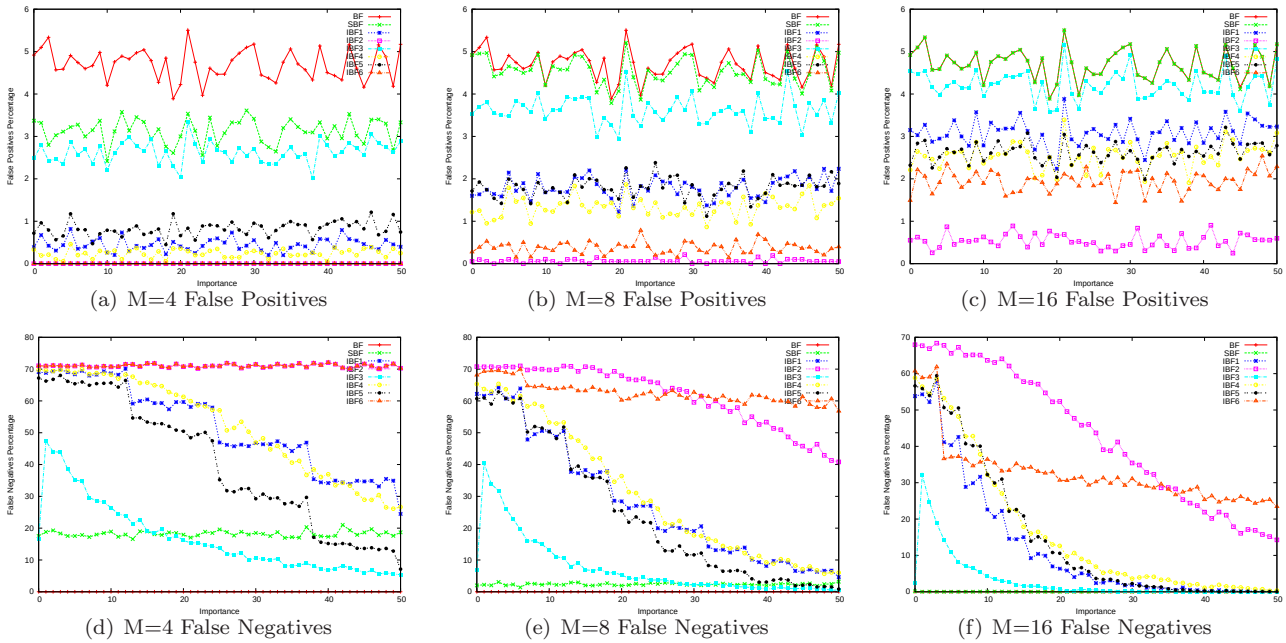
6.4.4 Varying M

Figure 6 depicts the variation that changing M causes in the behaviour of the various Bloom Filter algorithms. Notice that increasing M increases the false positive rates for all algorithms except the Bloom Filter (for which M is irrelevant). For SBF it goes up from 3% for $M = 4$to 5% for $M = 16$. The false negatives also fall accordingly — from 20% for $M = 4$ to almost *zero* for $M = 16$. In fact the SBF behaves very close to the Bloom Filter for $M = 16$. This behaviour is expected, since cells now take more iterations to go down to zero, and thus eviction of elements is harder. What is interesting to note is that the behaviour of IBF in general, and IBF_3 in particular, is less spectacular than SBF with respect to FNs with increase in M. For IBF_3, it drops from 50% to 30%. However, this causes the *Crossover Point* to go up significantly. Let us consider it for IBF_3. It is around 18 for $M = 4$, increases to around 30 for $M = 8$ and shoots up to almost 40 for $M = 16$. This is caused by the increasingly low FNs of the SBF. However, the difference in FPs for IBF_3 and SBF are quite significant (nearly 1%), and hence the crossover point may not be a good metric.

6.5 Counting the number of Empty Cells

We hypothesized in the above experiments that the reason for the difference in the results for the different algorithms and different parameters is the varying degree of 'Emptiness' of the Bloom Filter. In order to check whether this is indeed true, we ran a set of experiments, in which we measure the number of zeroes, i.e. the number of empty cells in the bloom filter periodically. The experiments were run with parameters $n = 512$, $k = 4$, $p = 5$ and $M = 5$, for a dataset of size 100000.

We notice that the IBF algorithms also satisfy the *Stable Property* that the SBF shows (Recall



(a) M=4 False Positives (b) M=8 False Positives (c) M=16 False Positives

(d) M=4 False Negatives (e) M=8 False Negatives (f) M=16 False Negatives

Figure 6: Importance Rates Experiments (varying M) n=65536, k=4, p=7

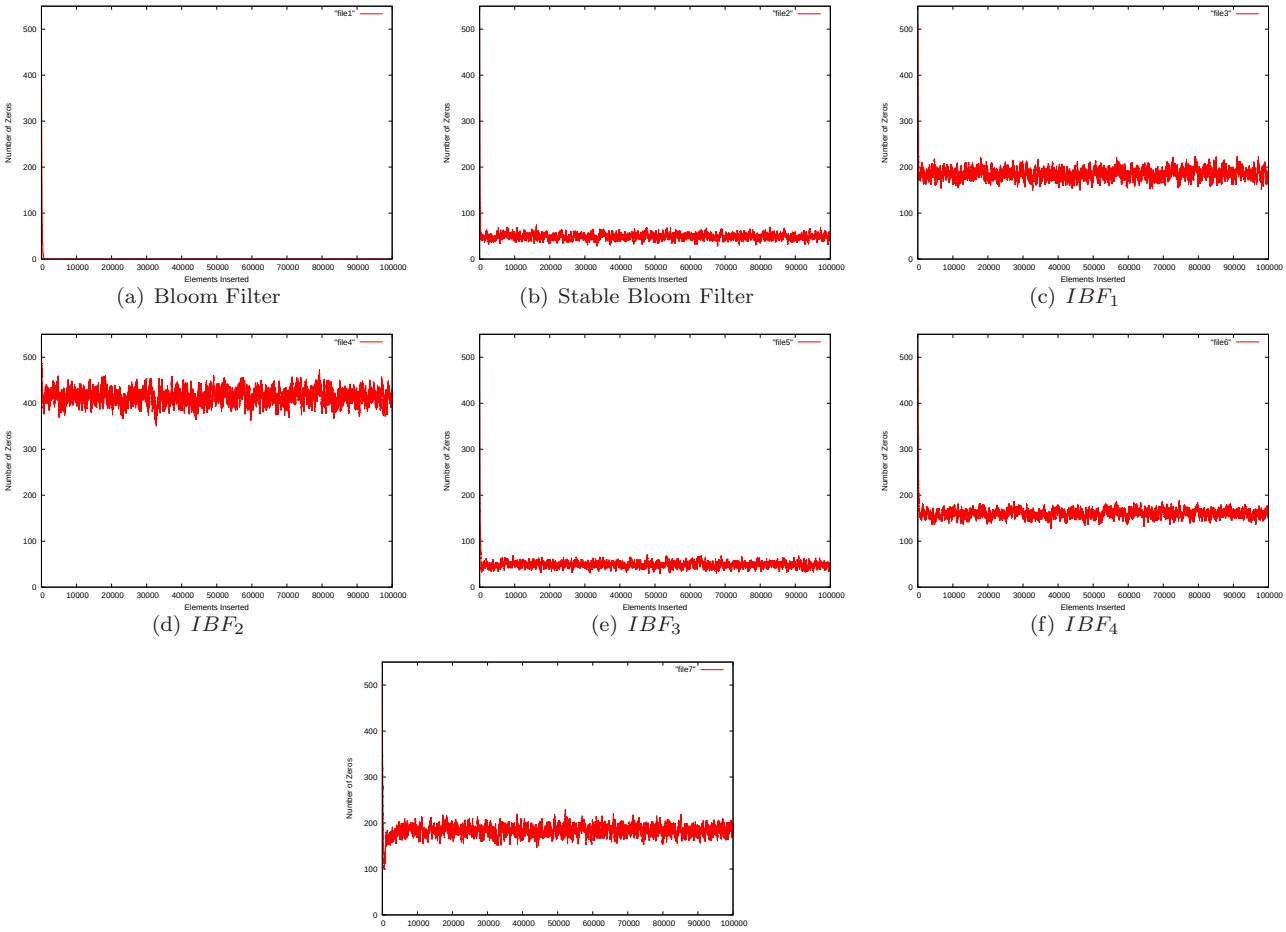
Section 2.2). The stable points, however, differ across algorithms. For SBF, it is around 50, and varies from 50 to 400 for the IBF algorithms. The bloom filter gets saturated very quickly, as expected. IBF_2 shows maximum deviation about the stable point, since it is the algorithm that most differs from the SBF algorithm. Notice that the number of zeros is highest in IBF_2, nearly 400. This validates the hypothesis we made regarding the low false positives in IBF_2 in section 6.3. When the Bloom Filter is largely empty (contains many zeroes), the false positive rates are low, and the false negative rates are higher. This also conclusively resolves the question raised in section 6.2 regarding which effect of varying k dominates. Increasing k increases the FP rates, and decreases FN rates, since it makes the array more 'full'.

6.6 Summary of Experiments

As stated above, the purpose of the experiments was threefold: to see whether the IBF behaves as it should, to compare it with older bloom filter algorithms, and to find the effect that varying the parameters has on the Bloom Filter results.

Figure 3 and Tables 3 and 4 show that the parameter that dominates most on the Error Metrics (False Positives and False Negatives) is n. Increasing the size of the bloom filter greatly reduces both false positives and false negatives of the bloom filter. However, n can in general not be increased arbitrarily since space available is limited. Therefore, k and p are also varied to see the effect. Out of the two, k dominates, and this agrees with [4], which states that k has M times as much effect as p on the false positive rates.

Comparing IBF algorithms with Bloom Filter and SBF, we find in figure 4 that for Importance based semantics, IBF indeed behaves much better than SBF and Bloom Filter. The False Positive rates are comparable to SBF (IBF_3, for example), and the False Negatives show a clear decreasing trend as the importance increases. From this we conclude that for applications where the importance semantics of data are important, IBF should be preferred over SBF and Bloom Filter. Among the IBF algorithms, IBF_5 and IBF_3 show consistently better performance than the others, with false negative rates upto 20 percent lower than IBF_2 and 10 percent lower than IBF_1. Hence, these algorithms are recommended.



(a) Bloom Filter (b) Stable Bloom Filter (c) IBF_1

(d) IBF_2 (e) IBF_3 (f) IBF_4

(g) IBF_5

Figure 7: Number of Zeroes in Bloom Filter

7 Discussion

This section will tie up a few loose ends regarding the IBF, involving the theory, the heuristics and the experiments. We shall also mention the work that can be done in the future to extend the IBF, and to use it in some applications.

- We conducted several experiments, using several other importance and deletion heuristics before finalizing these six. One of them was *MinImp-Delete* without priority, and without compulsarily deleting p cells. The problem with this strategy was that the same cells kept getting picked for deletion again and again (since $k << n$), even after they had gone to zero. Thus, deletion hardly happened and the observed behaviour was very close to BF. The problem with *MinImp-Delete with Priority* was that we had to set the priority to very large value to get a significant number of deletions. But with such large values of priority, the priority values began to dominate the importance, and the behaviour observed was not importance aware at all! Thus we discarded it too.

- The value of M used is rather small, generally being 4. Hence, the cells in the bloom filter can only have the values 0,1,2,3 and 4. The importance of the elements, however, varies between 1 and 50. Thus, the $imp - insert$ algorithm, doesn't quite capture enough importance detail. If you notice the graphs in Figure 4, the algorithms using imp-insert

(IBF_5 and IBF_1 in particular) show the variation of FN in 4 steps. These steps correspond to the four importance values to which the elements lying in each range are mapped.

- The deletion schemes *Probabilistic-Delete* and *MinImp-Delete* require extra state to be implemented. They must maintain the cumulative importance mapped to each cell. Thus, a space at least equal to the number of cells, is thus required. IBF_6 and IBF_1 do not suffer these drawbacks, as all the information they require is stored within the value of the cell itself. However, experimentally, the perfomance of the algorithms using extra space seems better. Experiments may be conducted to see if they perform better even when given equal total state. This can be done by comparing the behaviour of IBF_3 and IBF_5 of size n against an SBF of size $2n$.

- We mentioned in Section 3 that we would like the *Crossover Point* to be as low as possible. However, this may not always be the case. For example, we may have a situation where there is a trade-off, and we must choose between either a low crossover point, or less FNs for more important elements (much lower than SBF). Which one to choose out of these depends upon the application, and ideally we would like to give the application developer a 'knob', using which this behaviour could be changed. A similar 'knob' should exist for the FP vs FN tradeoff too. An application may exist where FPs are acceptable, but FNs are not. However, the converse may also be true for another application. Thus, the developer must be able to give a function of FP, FN and importance, and the Bloom Filter should minimize the cost, given a set of constraints. A mathematical, or emperical model can be formulated which will give the appropriate parameters to set, given the relevant input. This will ensure that analysis need not be done for every application to set the parameters, since merely applying the generic model on the requirements of the application would give us the optimum IBF for that application.

- The *Crossover Point* is a good emperical metric. However, a more rigorous metric is required that captures the benefit that an IBF gives over an SBF. WFN and WFP are also too crude, and a lot of information is lost, since the impairment for less important elements may cancel out the benefit for more important elements. Also these do not capture the benefit of IBF over SBF.

- There is a tremendous inter-play of the parameters, and IBF performance may also depend on application-specific metric. For example, the cost in terms of time or storage, if a result is missing due to FN, etc. Such cost-specific measures may be explored in the future. For instance, a specific application (say web-caching) may be taken, and the best IBF algorithm for it, with respect to the actual importance values in the application may be found out.

- Further experiments on real world data may be beneficial. We may take the click statistics of a proxy or a server or a client, and artificially assign importance values to the URLs. This would give us real data, but artificial importance values. Alternatively, we could use the files present on a web-proxy, and use their size as importance. As explained in the introduction, this is a realistic importance value, since the penalty for fetching large files is larger. These experiments would give us some insight on whether IBF actually works well in real world applications.

- Further, mathematical analysis of the IBF could be done, using Markov Chain analysis. Such a theoretical model for the IBF would be able to give better values of the optimum parameters, and would be able to prove (or disprove) the results that we got through experiments.

8 Conclusions

We have shown how data-importance can be leveraged in answering set membership queries over data streams. To this end, we have developed importance-aware functions that modify the operation of a Bloom filter. The various combination of insert and deletion heuristics we have proposed, have their own merits and limitations, but they perform much better than previous work. Most importantly, the Importance-aware Bloom filter (IBF) can be employed for an unbounded stream, and in settings where the data-importance distributions are not known. IBF shows promise since it suffers low FP rates, comparable to SBF, and the FN rates are non-increasing with respect to importance. These results are in lieu with our motivation, since our goal was to develop a scheme that will be considerate to application requirement that some data items are more important than others, and suffering a FN implies a higher cost to fix it. FPs on the other hand may be more tolerable, for example in web-caching where FP may indicate that object is already cached. Nevertheless, our work provides a framework to control FP and FN rates. More work, as described in section 7, could extend the basic framework we have provided to make the IBF a practical data structure to be used in several applications.

References

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

[2] Cheng, K., Xiang, L., Iwaihara, M., Xu, H., and Mohania, M. M. Time-decaying bloom filters for data streams with skewed distributions. *Research Issues in Data Engineering, International Workshop on 0* (2005), 63–69.

[3] Cohen, S., and Matias, Y. Spectral bloom filters. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2003), ACM, pp. 241–252.

[4] Deng, F., and Rafiei, D. Approximately detecting duplicates for streaming data using stable bloom filters. In *ACM SIGMOD'06*.

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

[6] Guo, D., Wu, J., Chen, H., Yuan, Y., and Luo, X. The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering*.

[7] Hristidis, V., Gravano, L., and Papakonstantinou, Y. Efficient ir-style keyword search over relational databases. In *VLDB* (2003), pp. 850–861.

[8] Yoon, M. Aging bloom filter with two active buffers for dynamic sets. *IEEE Transactions on Knowledge and Data Engineering 22* (2010), 134–138.