Spectral Bloom Filters

Saar Cohen
School of Computer Science
Tel Aviv University
saarco@cs.tau.ac.il

Yossi Matias[*]
School of Computer Science
Tel Aviv University
matias@cs.tau.ac.il

ABSTRACT

A Bloom Filter is a space-efficient randomized data structure allowing membership queries over sets with certain allowable errors. It is widely used in many applications which take advantage of its ability to compactly represent a set, and filter out effectively any element that does not belong to the set, with small error probability. This paper introduces the Spectral Bloom Filter (SBF), an extension of the original Bloom Filter to multi-sets, allowing the filtering of elements whose multiplicities are below a threshold given at query time. Using memory only slightly larger than that of the original Bloom Filter, the SBF supports queries on the multiplicities of individual keys with a guaranteed, small error probability. The SBF also supports insertions and deletions over the data set. We present novel methods for reducing the probability and magnitude of errors. We also present an efficient data structure and algorithms to build it incrementally and maintain it over streaming data, as well as over materialized data with arbitrary insertions and deletions. The SBF does not assume any a priori filtering threshold and effectively and efficiently maintains information over the entire data-set, allowing for ad-hoc queries with arbitrary parameters and enabling a range of new applications.

1. INTRODUCTION

Bloom Filters are space efficient data structures which allow for membership queries over a given set [2]. The Bloom Filter uses k hash functions, $h_1, h_2, \ldots, h_k$ to hash elements into an array of size m. For each element s, the bits at positions $h_1(s), h_2(s), \ldots, h_k(s)$ in the array are set to 1. Given an item q, we check its membership in the data-set by examining the bits at positions $h_1(q), h_2(q), \ldots, h_k(q)$. The item q is reported to be contained in the data-set if (and only if) all the bits are set to 1. This method allows a small probability of producing a false positive error (it may return a positive result for an item which actually is not contained in the set), but no false-negative error, while gaining substantial space savings. Bloom Filters are widely used in many applications.

This paper introduces the Spectral Bloom Filter (SBF), an extension of the original Bloom Filter to multi-sets, allowing estimates of the multiplicities of individual keys with a small error probability. This expansion of the Bloom Filter is spectral in the sense that it allows filtering of elements whose multiplicities are within a requested spectrum. The SBF extends the functionality of the Bloom Filter and thus makes it usable in a variety of new applications, while requiring only a slight increase in memory compared to the original Bloom Filter. We present efficient algorithms to build an SBF, and maintain it for streaming data, as well as arbitrary insertions and deletions. The SBF can be considered as a high-granularity histogram. It is considerably larger than regular histograms, but unlike such histograms it supports queries at high granularity, and in fact at the single item level, and it is substantially smaller than the original data set.

1.1 Previous work

As the size of data sets encountered in databases, in communication, and in other applications keeps on growing, it becomes increasingly important to handle massive data sets using compact data structures. Indeed, there is extensive research in recent years on data synopses [14] and data streams [1].

The applicability of Bloom Filters as an effective, compact data representation is well recognized. Bloom Filters are often used in distributed environments to store an inventory of items stored at every node. In [10], it is proposed to be used within a hierarchy of proxy servers to maintain a summary of the data stored in the cache of each proxy. This allows for a scalable caching scheme utilizing several servers. The Summary Cache algorithm proposed in the same paper was implemented in the Squid web proxy cache software [9, 23], with a variation of this algorithm called Cache Digest implemented in a later version of Squid.

In Peer-to-Peer systems, an efficient algorithm is needed to establish the nearest node holding a copy of a requested file, and the route to reach it. In [22], a structure called "Attenuated Bloom Filter" is described. This structure is basically an array of simple Bloom Filters in which component filters are labeled with their level in the array. Each filter summarizes the items that can be reached by performing a number of hops from the originating node that is equal to the level of that filter. The paper proposes an algorithm for efficient location of information using this structure.

[*]Research supported in part by the Israel Science Foundation.

The use of Bloom Filters was proposed in handling joins, especially in distributed environments. Bloomjoin is a scheme for performing distributed joins [17], in which a join between relations R and S over the attribute X is handled by building a Bloom Filter over R.X and transmitting it to S. This Bloom Filter is used to filter tuples in S which will not contribute to the join result, and the remaining tuples are sent back to R for completion of the join. The compactness of the Bloom Filter saves significant transmission size.

Bloom Filters were also proposed in order to improve performance of working with Differential Files [15]. A differential file stores changes in a database until they are executed as a batch. This reduces overheads caused by sporadic updates and deletions to large tables. When using a Differential File, its contents must be taken into account when performing queries over the database, with as little overhead as possible. A Bloom Filter is used to identify data items which have entries within the differential file, thus saving unnecessary access to the file. Another area in which Bloom Filters can be used is checking validity of proposed passwords [18] against previous passwords used and a dictionary. Recently, Broder et al [4] used Bloom Filters in conjunction with hot list techniques presented in [13] to efficiently identify popular search queries in the Alta-Vista search engine.

Several improvements have been proposed over the original Bloom Filter. In [21] the data structure was optimized with respect to its compressed size, rather than its normal size, to allow for efficient transmission of the Bloom Filter between servers, as proposed in [10]. Another improvement proposed in [18] is imposing a locality restriction on the hash functions, to allow for faster performance when using external storage. In [10] a counter has been attached to each bit in the array to count the number of items mapped to that location. This provides the means to allow deletions in a set, but still does not support multi-sets. To maintain the compactness of the structure, these counters were limited to 4 bits, which is shown statistically to be enough to encode the number of items mapped to the same location, based on the maximum occupancy in a probabilistic urn model, even for very large sets. However this approach is not adequate when dealing with frequencies of multi-sets, in which items may appear hundreds and thousands of times.

The concept of multiple hashing (while not precisely in the form of Bloom Filters) was used in several recent works, such as supporting iceberg queries [11] and tracking large flows in network traffic [8]. Both handle queries which correspond to a very small subset of the data (the tip of the iceberg) defined by a threshold, while having to efficiently explore the entire data. These implementations assume a prior knowledge of the threshold and avoid maintaining a synopsis over the full data set. A recent survey describes several applications and extensions of the Bloom Filter, with emphasis on network applications [3].

Current implementations of Bloom Filters do not address the issue of deletions over multi-sets. An insert-only approach is not enough when using widely used data warehouse techniques, such as maintaining a sliding window over the data. In this method, while new data is inserted into the data structure, the oldest data is constantly removed. When tracking streaming data, often we would be interested in the data that arrived in the last hour or day, for example. In this paper we show that the SBF provides this functionality as a built-in ability, under the assumption that the data leaving the sliding window is available for deletion, while allowing (approximate) membership and multiplicity queries for individual items. An earlier version of this work appears in [20].

1.2 Contributions

This paper presents the Spectral Bloom Filter (SBF), a synopsis which represents multisets that may change dynamically in a compact and efficient manner. Queries regarding the multiplicities of individual items can be answered with high accuracy and confidence, allowing a range of new applications. The main contributions of this paper are:

- The Spectral Bloom Filter synopsis, which provides a compact representation of data sets while supporting queries on the multiplicities of individual items. For a multiset S consisting of n distinct elements from U with multiplicities $\{f_x : x \in S\}$, an SBF of $N + o(N) + O(n)$ bits can be built in $O(N)$ time, where $N = k\sum_{x\in S}\lceil\log f_x\rceil$. For any given $q \in U$, the SBF provides in $O(1)$ time an estimate $\hat{f}_q$, so that $\hat{f}_q \geq f_q$, and an estimate error ($\hat{f}_q \neq f_q$) occurs with low probability (exponentially small in k). This allows effective filtering of elements whose multiplicities in the data set are below a threshold given at query time, with a small fraction of false positives, and no false negatives. The SBF can be maintained in $O(1)$ expected amortized time for inserts, updates and deletions, and can be effectively built incrementally for streaming data. We present experiments testing various aspects of the SBF structure.

- We show how the SBF can be used to enable new applications and extend and improve existing applications. Performing ad-hoc iceberg queries is an example where one performs a query expected to return only a small fraction of the data, depending on a threshold given only on query time. Another example is spectral Bloomjoins, where the SBF reduces the number of communication rounds among remote database sites when performing joins, decreasing complexity and network usage. It can also be used to provide a fast aggregative index over an attribute, which can be used in algorithms such as bifocal sampling.

The following novel approaches and algorithms are used within the SBF structure:

- We show two algorithms for SBF maintenance and lookup, which result with substantially improved lookup accuracy. The first, Minimal Increase, is simple, efficient and has very low error rates. However, it is only suitable for handling inserts. This technique was independently proposed in [8] for handling streaming data. The second method, Recurring Minimum, also improves error rates dramatically while supporting the full insert, delete and update capabilities. Experiments show favorable accuracy for both algorithms. For a sequence of insertions only, both Recurring Minimum and Minimal Increase significantly improve over the basic algorithm, with advantage for Minimal Increase. For sequences that include deletions, Recurring Minimum is significantly better than the other algorithms.

- One of the challenges in having a compact representation of the SBF is to allow effective lookup into the i'th string in an array of variable length strings (representing counters in the SBF). We address this challenge by presenting the *string-array index* data structure which is of independent interest. For a string-array of m strings with an overall length of N bits, a string-array index of $o(N) + O(m)$ bits can be built in $O(m)$ time, and support access to any requested string in $O(1)$ time.

1.3 Paper outline

The rest of this paper is structured as follows. In Section 2 we describe the basic ideas of the Spectral Bloom Filter as an extension of the Bloom Filter. In Section 3, we describe two heuristics which improve the performance of the SBF with regards to error ratio and size. Section 4 deals with the problem of efficiently encoding the data in the SBF, and presents the string-array index data structure which provides fast access while maintaining the compactness of the data structure. Section 5 presents several applications which use the SBF. Experimental results are presented in Section 6, followed by our conclusions.

2. SPECTRAL BLOOM FILTERS

This section reviews the Bloom Filter structure, as proposed by Bloom in [2]. We present the basic implementation of the Spectral Bloom Filter which relies on this structure, and present the Minimum Selection method for querying the SBF. We briefly discuss the way the SBF deals with insertions, deletions, updates and sliding window scenarios.

2.1 The Bloom Filter

A Bloom Filter is a method for representing a set $S = \{s_1, s_2, \ldots, s_n\}$ of keys from a universe U, by using a bit-vector V of $m = O(n)$ bits. It was invented by Burton Bloom in 1970 [2].

All the bits in the vector V are initially set to 0. The Bloom Filter uses k hash functions, $h_1, h_2, \ldots, h_k$ mapping keys from U to the range $\{1 \ldots m\}$. For each element in $s \in S$, the bits at positions $h_1(s), h_2(s), \ldots, h_k(s)$ in V are set to 1. Given an item $q \in U$, we check its membership in S by examining the bits at positions $h_1(q), h_2(q), \ldots, h_k(q)$. If one (or more) of the bits is equal to 0, then q is certainly not in S. Otherwise, we report that q is in S, but there may be false positive error: the bits $h_i(q)$ may be all one even though $q \notin S$, if other keys from S were mapped into these positions. We call this *bloom error* and denote it by E_b.

The probability for a false positive error is dependent on the selection of the parameters m, k. After the insertion of n keys at random to the array of size m, the probability that a particular bit is 0 is exactly $(1 - 1/m)^{kn}$. Hence the probability for a bloom error in this situation is

$$E_b = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k.$$

The right-hand expression is minimized for $k = \ln(2) \cdot (\frac{m}{n})$, in which case the error rate is $(1/2)^k = (0.6185)^{m/n}$. Thus, the Bloom Filter is highly effective even for $m = cn$ using a small constant c. For $c = 8$, for example, the false positive error rate is slightly larger than 2%. Let $\gamma = nk/m$; i.e, γ is the ratio between the number of items hashed into the filter and the number of counters. Note that in the optimal case, $\gamma = \ln(2) \approx 0.7$.

2.2 The Spectral Bloom Filter

The Spectral Bloom Filter (SBF) replaces the bit vector V with a vector of m counters, C. The counters in C roughly represent multiplicities of items, all the counters in C are initially set to 0. In the basic implementation, when inserting an item s, we increase the counters $C_{h_1(s)}, C_{h_2(s)}, \ldots, C_{h_k(s)}$ by 1. The SBF stores the frequency of each item, and it also allows for deletions, by decreasing the same counters. Consequently, updates are also allowed (by performing a delete and then an insert).

SBF basic construction and maintenance

Let S be a multi-set of keys taken from a universe U. For $x \in U$ let f_x be the frequency of x in S. Let

$$v_x = \{C_{h_1(x)}, C_{h_2(x)} \ldots, C_{h_k(x)}\}$$

be the sequence of values stored in the k counters representing x's value, and $\hat{v}_x = \{\hat{v}_x^1, \hat{v}_x^2 \ldots, \hat{v}_x^k\}$ be a sequence consisting of the same items of v_x, sorted in non-decreasing order; i.e. $m_x = \hat{v}_x^1$ is the minimal value observed in those k counters.

To add a new item $x \in U$ to the SBF, the counters $\{C_{h_1(x)}, C_{h_2(x)} \ldots, C_{h_k(x)}\}$ are increased by 1. The Spectral Bloom Filter for a multi-set S can be computed by repeatedly inserting all the items from S. The same logic is applied when dealing with streaming data. While the data flows, it is hashed into the SBF by a series of insertions.

Querying the SBF

A basic query for the SBF on an item $x \in U$ returns an estimate on f_x. We define the SBF error, denoted E_{SBF}, to be the probability that for an arbitrary element z (not necessarily a member of S), $\hat{f}_z \neq f_z$. The basic estimator, denoted as the *Minimum Selection (MS)* estimator is $\hat{f}_x = m_x$. The proof of the following claim as well as of other claims are omitted due to space limitation, and are given in the full paper [5].

CLAIM 1. *For all $x \in U$, $f_x \leq m_x$. Furthermore, $f_x \neq m_x$ with probability $E_{SBF} = E_b \approx \left(1 - e^{-kn/m}\right)^k$.*

The above claim shows that the error of the estimator is one-sided, and that the probability of error is the bloom error. Hence, when testing whether $f_x > 0$ for an item $x \in U$, we obtain identical functionality to that of a simple Bloom Filter. However, an SBF enables more general tests of $f_x > T$ for an arbitrary threshold $T \geq 0$, for which possible errors are *only false-positives*. For any such query the error probability is E_{SBF}.

Deletions and sliding window maintenance

Deleting an item $x \in U$ from the SBF is achieved simply by reversing the actions taken for inserting x, namely decreasing by 1 the counters $\{C_{h_1(x)}, C_{h_2(x)} \ldots, C_{h_k(x)}\}$. In sliding windows scenarios, in cases data within the current window is available (as is the case in data warehouse applications), the sliding window can be maintained simply by preforming deletions of the out-of-date data.

Distributed processing

The SBF is easily extended to distributed environment. It allows simple and fast union of multi-sets, for example when a query is required over several sets. Once a query is required upon the entire collection of sets, SBFs can be united simply by addition of their counter vectors. This property can be useful for partitioning a relation into several tables covering parts of the relation. Other features of the SBF relevant to distributed execution of joins are presented in Section 5.3.

SBF multiplication

Several applications, such as Bloomjoins (see Section 5.3), can be implemented efficiently by multiplying SBF. The multiplication requires the SBF to be identical in their parameters and hash functions. The counter vectors are linearly multiplied to generate an SBF representing the join of the two relations. The number of distinct items in a join is bounded by the maximal number of distinct items in the relations, resulting in an SBF with fewer values, and hence better accuracy.

External memory SBF

While Bloom Filters are relatively compact, they may still be too large to fit in main memory. However, their random nature prevents them from being readily adapted to external memory usage because of the multiple (up to k) external memory accesses required for a single lookup. In [18], a multi-level hashing scheme was proposed, in which a first hash function hashes each value to a specific block, and the hash functions of the Bloom Filter hash within that block. The analysis in [18] showed that the accuracy of the Bloom Filter is affected by the segmentation of the available hashing domain, but for large enough segments, the difference is negligible. The same analysis applies in the SBF case, since the basic mechanism remains the same.

SBF implementation

The major issues that need to be resolved for this data structure are *maintaining the array of counters*, where we must consider the total size of the array, along with the computational complexity of random access, inserts and deletions from the array, and *query performance*, with respect to *two* error metrics: the error rate (similar to the original Bloom Filter), and the size of the error.

3. OPTIMIZATIONS

In this section we present two methods that significantly improve the query performance that is provided by the SBF when the threshold is greater than 1; both in terms of reducing the probability of error E_{SBF}, as well as reducing the magnitude of error, in case there is one. For membership queries (i.e., threshold equals 1), the error remains unchanged.

3.1 Minimal Increase

The Minimal Increase (MI) algorithm uses a pretty simple logic: since we know for sure that the minimal counter is the most accurate one, if other counters are larger it is clear that they have some extra data because of other items hashed to them. Knowing that, we don't increase them on insertion until the minimal counter catches up with them. This way we minimize redundant insertions and in fact, we perform the minimal number of increases needed to maintain the property of $\forall x \in U,\ m_x \geq f_x$, hence its name.

Minimal Increase When performing an insert of an item x, increase only the counters that equal m_x (its minimal counter). When performing lookup query, return m_x. For insertion of r occurrences of x this method can be executed iteratively, or instead increase the smallest counter(s) by r, and set every other counter to the maximum of their old value and $m_x + r$.

A similar method appeared in [8], referred to as Conservative Update. We develop this method further and set some claims as to its performance and abilities. The performance of the Minimal Increase algorithm is quite powerful:

Claim 2 (Minimal Increase Performance). *For every item $x \in U$, the error probability in estimating f_x using the MI algorithm, E_{SBF}, is at most E_b, and the error size is at most that of the MS algorithm. Its counters hold the minimal values which maintains $m_x \geq f_x$.*

The Minimal Increase algorithm is rather complex to analyze, as it is dependent upon the distribution of the data and the order of its introduction. For the simple uniform case we can quantify the error rate reduction:

Claim 3. *When the items are drawn at random from a uniform distribution over U, the MI algorithm decreases the error E_{SBF} by a factor of k.*

Thus, the MI algorithm is strictly better than the MS algorithm for any given item, and can result with significantly better performance. This is indeed demonstrated in the experimental studies. Note that no increase in space is required here.

Minimal Increase and deletions.. Along with the obvious strength of this method, it is important to note that even though this approach provides very good results while using a very simple operation scheme, it does not allow deletions. In fact, when allowing deletions the Minimal Increase algorithm introduces a new kind of errors - *false-negative* errors. This result is salient in the experiments dealing with deletions and sliding-window approaches, where the Minimal Increase method becomes unattractive because of its poor performance, mostly because of false negative errors.

3.2 Recurring Minimum

The main idea of the next heuristics is to identify the events in which bloom errors occur, and handle them separately. We observe that for multi-sets, an item which is subject to Bloom Error is typically less likely to have recurring minimum among its counters. For item x with recurring minimum, we report m_x as an estimate for f_x, with error probability typically considerably smaller than E_b. For the set consisting of all items with a single minimum, we use a secondary SBF. Since the number of items kept in the secondary SBF is only a small fraction of the original number of items, we have improved SBF parameters (compared to the primary SBF), resulting with overall effective error that can be considerably smaller than E_b.

let E_x be the event of an estimation error for item x: $m_x \neq f_x$ (i.e., $m_x > f_x$). Let S_x be the event where x has a single minimum, and R_x be the event in which x has a recurring minimum (over two or more counters).

Table 1 shows experimental results when using a filter with $k = 5, n = 1000$, secondary SBF size of $m_s = m/2$, various γ values and Zipfian data with skew 0.5. Values shown are γ, usual Bloom Error E_b, fraction of cases with recurring minimum ($P(R_x)$), fraction of estimation errors in those cases ($P(E_x|R_x)$), the γ parameter for the secondary SBF $\gamma_s = n(1 - P(R_x))k/m_s$, E_b^s - the calculated Bloom Error for the secondary SBF. The next column shows the expected error ratio which is calculated by

$$E_{RM} = P(R_x)P(E_x|R_x) + (1 - P(R_x))E_b^s$$

The last column is the ratio between the original error ratio and the new error ratio. Note that for the (recommended) case of $\gamma = 0.7$, the SBF error (E_{RM}) is over 18 times smaller than the Bloom Error.

Note that the Recurring Minimum method requires additional space for the secondary SBF. This space could be used, instead, to reduce the Bloom Error within the basic, Minimum Selection method. Table 2 compares the error obtained by using additional memory, presented as a fraction of the original memory m, to increase the size of the primary SBF within the Minimum Selection method, vs. using it as a secondary SBF within the Recurring Minimum method. The error ratio row shows the ratio between the error of Minimum Selection and the error of the Recurring Minimum methods. In the Minimum Selection method, when we increased the primary SBF, we increased k from its original value $k = 5$, maintaining γ at about 0.7 (so as to have maximum impact of the additional space). The new value for k is shown in the table. A ratio over 1 shows advantage to the Recurring Minimum method. For instance, when having additional 50% in space, Recurring Minimum performs about 3.3 times better than Minimum Selection (note that as per Table 1 the total improvement is by a factor of about 18).

The algorithm. The algorithm works by identifying potential errors *during insertions* and trying to neutralize them. It has no impact over "classic" Bloom Error (false-positive errors) since it can only address items which appear in the data; it reduces the size of error for items which appear in the data and are "stepped over" by other items. The algorithm is as follows:

When adding an item x, increase the counters of x in the primary SBF. Then check if x has a recurring minimum. If so, continue normally. Otherwise (if x has a single minimum), look for x in the secondary SBF. If found, increase its counters, otherwise add x to the secondary SBF, with an initial value that equals its minimal value from the primary SBF.

When performing lookup for x, check if x has a recurring minimum in the primary SBF. If so return the minimum. Otherwise, perform lookup for x in secondary SBF. If returned value is greater than 0, return it. Otherwise, return minimum from primary SBF.

A refinement of this algorithm which improves its accuracy but requires more storage uses a Bloom Filter B_f of size m to mark items which were moved to secondary SBF. When an item x is moved to the secondary SBF, x is inserted into B_f as well, and this marks that x should be handled in the secondary SBF from now on. When inserting an item and it exists in B_f it is handled in the secondary SBF, otherwise it is handled as in the original algorithm. When performing lookup for x, B_f is checked to determine which SBF should be examined for x's frequency.

The additional Bloom Filter might have errors in it, but since only about 20% of the items have a single minimum (as seen in the tables), the actual γ of B_f is about a fifth of the original γ. For $\gamma = 0.7, k = 5$, this implies a Bloom Error ratio of $(1 - e^{-0.7/5})^5 = 3.8 \cdot 10^{-5}$, which is negligible when compared with other errors of the algorithm.

Deletions and sliding window maintenance

Deleting x when using Recurring Minimum is essentially reversing the increase operation: First decrease its counters in the primary SBF, then if it has a single minimum (or if it exists in B_f) decrease its counters in the secondary SBF, unless at least one of them is 0. Since we perform insertions both to the primary and secondary SBF, there can be no false negative situations when deleting items. Sliding window is easily implemented as a series of deletions, assuming that the out-of-scope data is available.

Analysis. Since the primary SBF is always updated, in case the estimate is taken from the primary SBF, the error is at most that of the MS algorithm. In many cases it will be considerably better, as potential bloom error are expected to be identified in most cases. When the secondary SBF provides the estimate, errors can happen because of Bloom errors in the secondary SBF (which is less probable than Bloom errors in the primary SBF), or due to late detection of single minimum events (in which case the magnitude of error is expected to be much smaller than in the MS algorithm). A full analysis is given in the full paper.

3.3 Methods comparison

We compare the three methods.

Error rates. The MS algorithm provides the same error rates as the original Bloom Filter. Both RM and MI methods perform better over various configurations, with MI being the most accurate of them. These results are consistent in the experimental results, taken over data with various skews and using several γ values. For example, with optimal γ and various skews, MI performs about 5 times better in terms of error ratio than the MS algorithm. The RM algorithm is not as good, but is consistently better than the MS algorithm.

Memory overhead. The RM algorithm requires an additional memory for storing the secondary SBF, so it is not always cost-effective to use this method. The MI algorithm is the most economical, since it needs the minimal number of insertions. Note that, as seen in the experiments, when using the same overall amount of memory for each method, the RM algorithm still performed better than the MS algorithm (but MI outperforms it).

Complexity. The RM algorithm is the most complex method, because of the hashing into two SBFs, but this happens only for items with non-recurring minimum. As shown above, this happens for about 20% of the cases, which accounts for 20% increase in the average complexity of the algorithm. When using the flags array in the RM algorithm, the complexity naturally increases.The MS method is the simplest.

Updates/Deletions. Both the MS and RM methods support these actions. The MI algorithm does not, and may produce false-negative errors if used. Experiments show that in these cases, the MI algorithm becomes practically unusable. For example, using sliding window, the additive error

γ	E_b	$P(R_x)$	$P(E_x\|R_x)$	γ_s	E_b^s	E_{RM}	E_b/E_{RM}
1	0.101	0.657	0.0045	0.686	0.03	0.0132	7.59
0.83	0.057	0.697	0.0028	0.502	0.0096	0.0048	11.7
0.7	0.032	0.812	0.002	0.263	0.0006	0.0017	18.48
0.625	0.021	0.799	0.0012	0.251	0.00054	0.001	20.3
0.5	0.009	0.969	0	0.031	$2.65 \cdot 10^{-8}$	$8.21 \cdot 10^{-10}$	11480352

Table 1: Error rates with recurring minimum and without it. E_b is the usual Bloom Error, $P(R_x)$ is the ratio of recurring minimum, $P(E_x|R_x)$ is the ratio of errors given recurring minimum, γ_s, E_b^s are the secondary BF parameters (with size m/2), E_{RM} is E_{SBF} for recurring minimum, and the last column is the gain.

memory increase	1	0.5	0.33	0.25	0.2	0.1
Error Ratio	0.641	3.341	4.546	3.628	2.496	0.562
Modified k	10	7	6	6	6	5

Table 2: Effect of increased memory for primary SBF and secondary SBF, with original $k = 5$.

of the MI algorithm is 1 to 2 orders of magnitude larger than that of the RM algorithms, for various skews.

4. DATA STRUCTURES

While the data structure implementation of the (original) Bloom Filter is a simple bit-vector, the implementation of the SBF presents a different challenge. The SBF of a multiset of M items, consists of a sequence of counters $C_1, C_2, \ldots, C_m$, where C_i is the number of items hashed into i, so that $\sum_{i=1}^{m} C_i = k \cdot M$. Let $N = \sum_{i=1}^{m} \lceil \log C_i \rceil$; then, $k(n-1+\log M) \leq N \leq kn \log(M/n)$, where n is the number of distinct items in the set. The goal is to have a compact encoding of the SBF which is as close to N as possible. Clearly, a straight-forward implementation of allocating $\log M$ bits per counter is excluded. In this section we show:

Theorem 4. *An SBF of size $N+o(N)+O(m)$ bits can be constructed in $O(N)$ time, supporting lookup in $O(1)$ time. Furthermore, the SBF can be maintained so that insertions, deletions and updates take each $O(1)$ expected amortized time.*

The basic representation of the SBF consists of embedding the counters C_i in their $\lceil \log C_i \rceil$-bit binary representation, consecutively in a *base array* of size N bits. (For simplicity of exposition, we will omit below the ceiling operator.) In the static case the counters are placed without any gap between them, totaling N bits, whereas to support dynamic changes we add $\epsilon' m$ slack bits between counters, where $\epsilon' > 0$ is a small constant. This representation introduces a challenge in executing the lookup operations, since locations of various strings are not known due to their variable sizes.

In Section 4.1 we address this challenge, presenting a data structure that enables effective "random access" to the i'th substring, for any i, in a sequence consisting of arbitrary variable length substrings. Section 4.2 shows how to handle the dynamic problem, supporting inserts and deletes over the data set represented by the SBF. The proposed SBF implementation is general, with no assumption made on the distribution of the data. Finally, in Section 4.3, we show an alternative method which requires only $O(m)$ bits in addition to the base array (rather than $o(N)+O(m)$), but which is less efficient when performing lookups.

4.1 The String-Array Index

We first define a general access problem related to the one encountered in the context of the SBF.

Variable length access problem. Let $\{s_1, s_2, \ldots, s_m\}$ be binary strings of arbitrary lengths. Let $S = s_1 s_2 \ldots s_m$ be the concatenation of those substrings, with length $|S| = N$. Given an arbitrary i, $1 \leq i \leq m$, return the position of s_i in S, and optionally, s_i itself.

Note that the lookup problem for the SBF compact base-array representation is the variable length access problem with two additional constraints: *(i)* $\forall i, |s_i| \leq \log M$; and *(ii)* the strings roughly represent the frequencies of items in the given data set, and the order between them is determined at random using the hash functions of the SBF. We describe a data structure, the *string-array index*, that addresses the general, unconstrained variable length access problem.

The string-array index uses a combination of various instances of three types of simple data structures, which hold offset data for given sequences of some σ items, totaling some T bits:

1. Coarse Vector - this is the backbone of the string-array index, and its role is to effectively reduce a given problem into a set of smaller sub-problems. It partitions the given sequence into σ/σ' subsequences of σ' items each, and provides offset information for the beginning of each subsequence, using an array of fixed-sized offsets. The coarse vector requires $(\sigma/\sigma')\log T$ bits, and reduces the access problem (for a given i) into a problem with σ' items and some $T' < T$ length.

2. Offset Vector - provides a straightforward representation of the σ offsets in an array, requiring $\sigma \log T$ bits, and supports $O(1)$ lookup time. It is used when σ is small relative to T; in particular when $\sigma \log T \ll T$, and it can therefore be stored for such subsequences within the required space bounds. If $T \gg \sigma \log N$ then the offsets are with respect to the base array.

3. Lookup Table - a global array, whose indices represent all possible sequences and queries over those sequences, for a sufficiently small T. It requires $2^{O(T)}$ bits, which is $o(N)$ for $T = o(\log N)$. A problem with a sufficiently small T can use it for $O(1)$ lookup time, by storing additional appropriate encoding information that maps

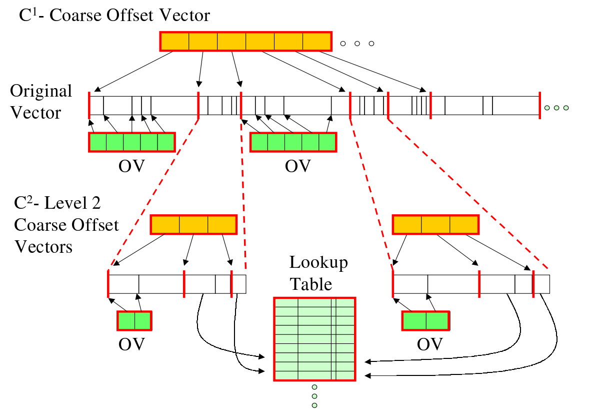

Figure 1: The String-Array Index data structure.

it into its appropriate array index.

For a given variable length access problem consisting of m strings totaling N bits, a string-array index can be constructed as follows.

Lemma 5 (String-Array Index). *The string-array index data structure of size $o(N) + O(m)$ bits can be built in $O(m)$ time, and subsequently support access to any requested item s_i in $O(1)$ time.*

The string-array index is depicted in Figure 1; it consists of two levels of arrays of pointers to sub-sequences of S. The first level consists of a *coarse offset array* C^1, which holds $m/\log N$ offsets of the positions of $\log N$-size groups of items in the SBF base array. Since offsets are at most N, they can be represented using $\log N$ bits, for a total size of m bits. The offset in C^1_j points to the $(j \log N)$'th item in S, i.e., to s_r where $r = (j \log N)$. Thus, for any i, one access to C^1 can provide us with the pointer to a subsequence S' of $\log N$ items in S, that includes s_i.

The second level enables effective access within such subsequences S'. If a subsequence is of size larger than $\log^3 N$ bits, then it is supported by a simple offset vector, consisting of the $\log N$ offsets of the individual items of the subsequence, in the SBF base array; each offset is of $\log N$ bits, totaling $\log^2 N$ bits for the entire offset vector. The total size of all such offset vectors is at most $N/\log N$ bits.

Each subsequence S' whose size is at most $\log^3 N$ bits is supported by a *level-2 coarse offset array* C^2_j, which partitions S' to chunks of $\log\log N$ items. It holds $\log N/\log\log N$ offsets of the $\log\log N$-size chunks S'' inside S'. Since offsets are at most $\log^3 N$, each can be represented using $3\log\log N$ bits, totaling $3\log N$ bits per a subarray C^2_j. The total size of all such subarrays is hence at most $3m$.

A lookup using the string-array index requires 2 lookups through the coarse offset arrays, which provides with either the exact position of the requested item in the SBF base array, or a pointer to the beginning of a subsequence S'' of $\log\log N$ items, which includes the requested item. The items within each subsequence S'' are accessed either through an offset vector built for S'', or using a global lookup table shared by all subsequences, depending on the size of S''. We use a threshold $T_0 = (\log\log N)^3$, to determine which method is used. Let S'' be of size $T = T(S'')$ bits.

If $T > T_0$, we keep for S'' an offset vector; since $T \leq \log^3 N$, each offset can be represented using $3\log\log N$ bits, and the offset vector for S'' will consist of such $\log\log N$ offsets, totaling size $3(\log\log N)^2 \ll T(S'')$. Hence, the total size of all such offset vectors is $o(N)$.

It remains to deal with S'' such that $T \leq T_0$. We keep a single global lookup table, that will serve all such subproblems. An entry to the lookup table consists of a string representing a subsequence S'' and an index i, $1 \leq i \leq \log\log N$. For each such entry, the lookup table will return the offset from the beginning of S'' in the SBF base array, of the i'th item in S''.

The lookup table consists of a simple array LT, whose indices represent all binary combinations representing the entries $\langle S'', i\rangle$, and for each entry the result is precomputed and stored in the array LT. Each entry can be represented as the T-bit subarray of the SBF base array representing S'', and a secondary subarray $L(S'')$ consisting of an encoding of the lengths of the items in S'', so as to allow unique interpretation of the T-bit subarray representing S''. The encoding in $L(S'')$ has the property that the size of each code word is proportional to the encoding length of the value it represents. This is obtained using, e.g., Elias Encoding (see Section 4.3). The length of $L(S'')$ is either $O(\log\log N)$ or $o(T)$. In addition to the representation of S'' (including $L(S'')$) the entry includes the index i (consisting of $\log\log\log N$ bits).

It is easy to see that since $T \leq T_0$, the total size of LT is $o(N)$ bits, and that all its entries can be computed in $o(N)$ time. The subarray $L(S'')$ is stored for each S'' whose size T is less than T_0 as part of the SBF. The offset of the ith item in such S'' is obtained by looking up at LT the value corresponding to the entry consisting of the $\langle S'', i\rangle$, as determined using $L(S'')$.

In summary, the string-array index consists of the following components: the coarse offset array C^1, an array C^2 consisting of all level-2 coarse offset arrays C^2_j, the offset vectors of first level and second level sequences, the global lookup table LT, and the length arrays $L(S'')$. The total size of the string-array index is $o(N) + O(m)$, its construction takes $O(m)$ time, and it can be used as discussed to solve the variable length access problem in $O(1)$ time. The lemma follows.

Note that when actually implementing a string-array index, several of the structures could be eliminated or altered due to practical considerations. In particular, even for relatively large values of N, one should not be concerned with paying $O(\log\log N)$ factor overhead for a fraction of the data structure.

The SBF can now be constructed as stated in Theorem 4: the base-array is built in $O(N)$ time by updating the counters C_i as the input data set items are hashed one by one. Subsequently, building the string-array index over the base array. This requires using during construction time a temporary array of $O(m \log M)$ bits. Next subsection shows how to construct the SBF incrementally, as well as how to support update operations, without using any temporary array, and within the storage bounds of $N + o(N) + O(m)$ bits.

4.2 Handling updates

We show how to extend the string-array index data structure described above, to allow dynamic changes in the dataset, for a base array of an SBF. When one of the counters increases its bit-size in the base array, additional space needs to be allocated within the base array to accommodate the enlarged substring. It is also necessary to update the string-array index structure to reflect the changes made in the base-array. Delete operations only affect individual counters, and do not affect their positions, and hence the string-array index. To remain within storage bounds, after a long sequence of deletions the entire data structure is rebuilt, with amortized constant time per deletion.

To support inserts, we allocate a slack of extra bits in the base array. In particular, we add ϵm slack bits, one every $1/\epsilon$ items, for some $\epsilon > 0$. A counter which needs to expand "pushes" the item next to it, which in turn pushes the next item, until a slack is encountered. For each item, the nearest slack is initially allocated within a distance of at most $1/\epsilon$ items. However, upon expansion, the nearest slack may not be available, in case at least one of the items between the expanded item and the slack was already expanded. In such case, farther slack will need to be used. The cost of expansion is linear in the number of items that need to be pushed, assuming that each item fits into machine word.

The next lemma bounds the expected distance from an expanded item to the nearest available slack, using the fact that items location is determined at random by the hash functions of the SBF. For purpose of simplicity, we assume full randomness. It is assumed that the number of inserts is at most $\epsilon' m$, for some $\epsilon' > 0$. After $\epsilon' m$ inserts, the base array is refreshed by moving counters so that that slacks are again placed in $1/\epsilon$ intervals, and the string-array index is updated accordingly.

LEMMA 6. *Suppose that the size of some counter C_j increases, and that the total number of insertions is at most $\epsilon' m$, for $\epsilon' = \epsilon/2e$. Then, the number of items between C_j and the first available slack, denoted ℓ_j, satisfies $\mathbf{E}(\ell_j) = O(1/\epsilon)$.*

PROOF. Suppose first that C_j increases for the first time. A slack is available within the sub-array of i/ϵ items following C_j, if the number d_i of expansions of items within this sub-array is less than i. Since items are hashed into the base array at random, then for any sequence of $\epsilon' m$ insertions, d_i is bounded by a binomial with parameters $(\epsilon' m, i/(\epsilon m))$. Hence, $\mathbf{E}(d_i) \leq i\epsilon' m/(\epsilon m) = i\epsilon'/\epsilon$. The probability that items within i chunks will need to move upon an insertion is bounded by $P_i = Pr(d_i \geq i) = Pr(d_i \geq \frac{\epsilon}{\epsilon'}\mathbf{E}(d_i)) \leq (e\frac{\epsilon'}{\epsilon})^i$, with the last inequality due to Chernoff bounds. Hence, $\mathbf{E}(\ell_j) \leq \sum_{i=1}^{\infty} i/\epsilon \cdot P_i \leq \sum_{i=1}^{\infty}(i/\epsilon)\cdot(e\frac{\epsilon'}{\epsilon})^i = 1/\epsilon \sum_{i=1}^{\infty} i(\frac{1}{2})^i \leq 2/\epsilon$.

It remains to account for repeated expansions of particular counters. Suppose that a counter C_j has a sequence of x expansions. For the last expansion, it is guaranteed that the nearest $x-1$ slack bits are not available. Further, items within the nearest $x-1$ chunks of size $1/\epsilon$ might also have been expanded resulting with additional slack unavailability. On the other hand, the additional expected cost can be amortized against the 2^x updates to C_j which are required to facilitate x expansions. The expected amortized cost per repeated expansion remains $O(1)$. $\square$

The string-array index is updated when items are moved. The update of the structure has the same computational complexity as that of updating the base array itself, since essentially only offset information about items that are pushed needs to be changed in the string-array index. The expected amortized cost per update therefore remains $O(1)$. Since refreshing the entire base array and updating the string-array index takes $O(m)$ time, the amortized cost of such refresh and update is $O(1/\epsilon')$ per update.

4.3 An alternative approach

The data structure can be made more compact, while sacrificing lookup performance, by using the C^1 and C^2 indexes and not building any further structures. Once the problem is reduced to $\log\log N$ items, we allow a serial scan of the sub-group in order to access the requested item. To allow that, we need a compact prefix-free encoding that can be read sequentially. For this purpose we use a combination of Elias encoding and a method which is more compact for small counters.

In this scenario, a sub-group consists of $\log\log N$ items. Using the encodings presented in this section, each counter with value c can be encoded with close to $\log c$ bits. Therefore, this approach requires N bits to encode the actual counters in the original vector, with additional $o(m)$ bits for the structures of C^1 and C^2, while on average a lookup costs $\log\log N$. The same approach described in section 4.2 can be used to allow dynamic maintenance of the structure.

Elias encoding

The Elias encoding [7] consists of the following method: Let $B(n)$ be the binary representation of the integer, with length $L(n)$. A binary prefix code $B_1(n)$ is created by adding a prefix of $L(n)-1$ zeroes to the beginning of $B(n)$. Now we create the sequence representing n by encoding $B_1(L(n))$ followed by $B(n)$ with its leading 1 removed[1]. The total length of this representation is

$$L_2(n) = \lfloor \log_2 n \rfloor + 2\lfloor \log_2 (\lfloor \log_2 n \rfloor + 1) \rfloor + 1$$

The steps method

Elias encoding is a strong and simple method to create an encoding which is prefix-free while being compact. However, for very small numbers the overhead of $\log\log n$ bits is substantial and should be avoided. For example, to encode the number 1 (actually encoding the number 2) we need 4 bits. In sets, most counters will be 1, so for an optimal hit ratio of 0.5, the average is 2.5 bits per counter.

To solve that problem, we use compact encoding for small numbers. For example, using 0 to represent 0, 10 to represent 1 and 11 means the number is bigger than 1, with the Elias encoding of this number following the prefix. This reduces the cost to 1.5 bits per counter. It is further reduced if we encode longer sequences, reducing the overhead to an ϵ as small as we choose. Full details are omitted due to space limitations).

5. APPLICATIONS

In this section we explore a range of applications which may take advantage of the abilities of the SBF. The SBF enables new applications which use its properties to efficiently perform tasks such as ad-hoc iceberg queries. Other application (such as Bloomjoins or Range queries) are extensions of methods or abilities of the regular Bloom Filter.

[1]The Elias encoding does not encode the number 0. Therefore, when encoding n, we actually encode $n+1$, this does not effectively change the size expectations

5.1 Aggregate queries over specified items

Spectral Bloom Filters hold mostly accurate information over each and every item of the data set, and therefore can approximately answer any (aggregate) query regarding a given subset of the items, so that the error ratio is expected to be E_{SBF}, and the size of the error is expected to be smaller than the average frequency of items in the set, $\bar{f}$. For example, queries of the kind

```
SELECT count(a1) FROM R WHERE a1 = v
```

In performing this query, the SBF acts as an aggregate index built upon the attribute $a1$ and providing the (mostly) accurate frequency of v in the relation. Other aggregates, such as average,sum,max etc. can be easily implemented using this basic ability. The SBF behaves very much like a histogram where each item has its own bucket. Since the SBF keeps the full information, it is very versatile in its uses, while requiring storage relative to the size of the set.

5.2 Ad-hoc iceberg queries

In some cases we are interested in monitoring insertions, and want to set some triggers that will alert us once an item with a high count is inserted. For example, a company which tracks customers can create a calculation that reports their likeliness to churn. Once a customer with a high churning probability contacts the company, the company representative should be alerted, so he can offer him special deals. The threshold for such special treatment is dynamic, and depends on many factors, so the calculation cannot be executed a priori.

This example presents a sort of an Iceberg query, in which the threshold against which items are tested upon insertion is dynamic and possibly changes between queries. Other methods, proposed in [11, 19] require a certain preprocessing the data given a static threshold. When the threshold changes, the methods of [11, 19] require rescanning of the data using the new threshold (or in the case of streaming data [19], it cannot be done), while the SBF does not require any additional scan of the data, other than one that examines the data against the counts stored in the SBF.

5.3 Spectral Bloomjoins

Bloomjoins [17] are a method for performing a fast distributed join between relations R_1 and R_2 residing on different database servers - R_1 in site 1 and R_2 in site 2, based on attribute a. Both relations have a BF built on attribute a. The Bloomjoin method is executed in the following steps: R_1 sends its BF (B_1) to R_2, R_2 is scanned and tuples with a match in B_1 are sent back to site 1 as R'_2. At site 1, R_1 is joined with R'_2 to produce final result. This method is economical in network usage, since in the first transmission, only a synopsis is sent, and the second transmission usually contains a small fraction of the tuples, since a filtering stage was executed.

SBFs can be used to perform distributed aggregative queries, such as the following query, which filter the results using a given threshold:

```
SELECT R.a,count(*) FROM R,S
WHERE R.a = S.a GROUP BY R.a
HAVING count(*) [>,=] T
```

Since in most schemas the join between the relations will be a one-to-many join, the detail table S can send its SBF to R's site. The Bloom Filters are multiplied and R is scanned, testing each tuple in SBF_{RS} against the threshold T. Results can be reported immediately since no value is repeated more than once in R. When using "$>$" (or "$\geq$") as the filter operator, there is only a small fraction ρ of false positive errors, $\mathbf{E}(\rho) = E_{SBF}$, and no false negatives. Since the errors are one-sided, they can be eliminated by retrieving the accurate frequencies for the items in the result set, resulting in a fraction of ρ extra accesses to the data. The effectiveness of this method increases as the size of the result set decreases. When using the "=" operator, two-sided errors are possible, with recall of $1-E_{SBF}$, and possibly additional false-alarms.

The SBF's ability to maintain counters can also be used in queries which perform no filtering, such as the following:

```
SELECT R.a,count(*) FROM R,S
WHERE R.a = S.a GROUP BY R.a
```

To perform this query using a Bloomjoin, the full scheme described in [17] must be executed, with Bloom Filters and tuple stream sent back and forth between the sites. However, using SBF multiplication, a shorter scheme can be executed, assuming that both S and R have a SBF representing the attribute a present, and R being the primary query site: S sends its SBF (SBF_S) to R's site, where SBF_S and SBF_R are multiplied to create SBF_{SR}. Next, R is scanned, and each tuple is checked against SBF_{SR} for existence. If it exists, the item and its frequency are reported.

This scheme does not guarantee exact results. Items which appear in R and not in S may be reported because of errors in SBF_S. Also, the frequencies reported are subject to Bloom Error and may be higher than their actual value. To ensure the uniqueness of items in the results, we suggest the use of a validating SBF for that purpose. This method saves the transmission of data back to the main site. If the main site has to be the one reporting the results, the final answer may be sent back to it, with minuscule network usage.

Advantages. Using SBF for Bloomjoins simplifies and shortens the algorithm for performing the distributed joins. While the SBF itself is slightly larger than a Bloom Filter of the same parameters, this is balanced by the shorter operation scheme, requiring less SBFs to be sent between sites, and therefore saving bandwidth.

5.4 Bifocal sampling

A Spectral Bloom Filter can be plugged into various schemes that require an index on a relation for count queries. One such application is Bifocal Sampling [12], where using an SBF one can get similar join estimations without using an expensive index. The paper deals with joining two relations with unknown properties by dividing each relation to two distinct groups: dense and sparse tuples. The join size is estimated by combining the groups in all ways possible, creating a dense-dense join and sparse-any joins. In the sparse-any case, a join of type *t-index* [16] is used, meaning for each tuple in a sample of one relation, a query on the other relation is performed to determine the frequency of the join attribute in the second relation. By replacing the t-index with an SBF, the multiplicities used for estimation are replaced by their approximations, resulting with only a small additional error to the overall estimate.

Advantages. The SBF provides an efficient approximation to the t-index scheme, and enables a more space-efficient implementation of Bifocal Sampling.

6. EXPERIMENTS

We have tested the accuracy of the various SBF algorithms described in Section 3, as well as the space efficiency of the encoding methods described in Section 4.3.

Algorithms comparisons. We have tested and compared the three lookup schemes from Section 3: Minimum Selection (MS), Recurring Minimum (RM), and Minimal Increase (MI). The SBF was implemented using hash functions of modulo/multiply type: given a value v, its hash value $H(v)$, $0 \leq H(v) < m$ is computed by $H(v) = \lceil m(\alpha v \mod 1) \rceil$, where α is taken uniformly at random from $[0, 1]$. We measured two parameters; the first is the mean squared additive error, which is calculated by

$$E_{add} = \sqrt{\frac{\sum_{i \in v} \left(\hat{f}_i - f_i\right)^2}{n}}$$

The second is the *error ratio* E_{ratio}, computed as the fraction of the queries that return erroneous results. Thus, $\mathbf{E}(E_{ratio}) = E_{SBF}$, and for MS, it is E_b. Each reported result is the average over 5 independent experiments with the same parameters.

Two sets of tests were conducted; in both we used synthetic data produces by a Zipfian distribution. We used integers as data values, and the data set was constructed of 1000 distinct values, with $M = 100,000$. We have also conducted experiments in which M, and hence the average item frequency, was changed, generating smaller (and bigger) data sets. The observed behavior was consistent with the experiments reported here.

In the first set of tests, the skew of the data was changed, from $\theta = 0$ (uniform data) to $\theta = 2$ (very skewed data). The results are shown in Figure 3a,b (solid lines). As can be seen, the MI algorithm has the best performance both in terms of additive error and error ratio, and is very stable with regard to changes in the skew. The RM algorithm outperforms the MS algorithm in both parameters, but in most cases is no match to the MI algorithm.

In the second set of tests, the storage size m was changed, to produce $\gamma = nk/m$ ranging from about 0.12 to about 2. The results are shown in Figure 2a,b. For a fair comparison between the algorithms, in this and in all other experiments the RM algorithm used m as an overall storage size; that is the sizes of the primary and the secondary SBFs together being m. This causes the actual γ of the RM algorithm in its primary SBF to be larger than that of the MS and MI algorithms. These experiments show that all three algorithms behave similarly, with RM and MI being almost identical in their error ratios. The MI algorithm performs best in terms of additive error when m is small (and γ increases). This is due to the fact that it performs a minimal number of actual insertions into the base array, which becomes critical as the error ratio increases.

The third experiment tested the behavior of the various schemes when the number of hash functions (k) changes. The data used was again Zipfian with a skew of 0.5, in all configurations γ was fixed at 0.7 by increasing m along with k. The results are shown in Figure 2c. In the $k = 1$ case, all the methods perform the same (as they should). The MI method improves dramatically when k increases, while the RM method needs k of at least 3 to become effective, with major improvement when k increases to 4 and more. These experiments show clearly the incredible precision and stability of the Minimal Increase method, and also the substantial improvement that the Recurring Minimum method shows over the Minimum Selection.

Deletions and sliding window. Next, we tested the SBFs when faced with deletions. The setup consisted of a series of insertions, followed by a series of deletions and so on. In every deletion phase, 5% of the items were randomly chosen and were entirely deleted from the SBF. The results, shown in Figure 3, compare the error ratio and the additive error of the SBFs when subject to deletions to their performance without deletions. It is evident that the MI algorithm deteriorates dramatically when deletions are performed. The third graph shows the main reason for that - false-negative errors. Note that almost all of the errors of the MI algorithm are false negatives (MS and RM have no false-negatives). This makes it a poor choice when deletions are considered, since the one-sided nature of the errors is no longer valid.

The second test shown in Figure 4, used a sliding window scenario. In this experiment, a total of M items were inserted, but the SBFs only kept track of the $M/5$ most recent items as items were inserted, with data leaving the window explicitly deleted. The MS and the RM algorithm are much better that the MI algorithm for this scenario, with advantage to the RM.

Encoding methods. We tested the storage needed by the encoding methods described in Section 4.3, comparing the Elias method, and several configurations of "steps" for data with varying average frequency of items. The results, shown at Figure 5, were compared to the "Log Counters", which is simply $\sum_{i=1}^{m} \log C_i$. For data sets with average frequency close to 1 ("almost set") the steps methods are more economical, due to their low overhead. However, the Elias encoding improves as the average frequency increases, and beats the performance of the steps methods.

7. CONCLUSIONS

This paper presented Spectral Bloom Filters, extending Bloom Filters by storing counters instead of bit flags. The structure supports updates and deletions, while preserving storage size of $N + o(N) + O(m)$ bits. We presented several heuristics for insertions and lookups in a SBF. Minimum Selection uses the same logic as the original Bloom Filter. Minimal Increase is a simple yet powerful heuristic with very low error rates, but no support for updates and deletions. Recurring Minimum uses a secondary storage to take care of "problematic" cases, and it supports deletions and updates with no accuracy loss. We also present the string-array index, a data structure which provide fast access to variable-length encoded data while being compact enough to be used in the Spectral Bloom Filter. We show its structure and maintenance for static data and during dynamic

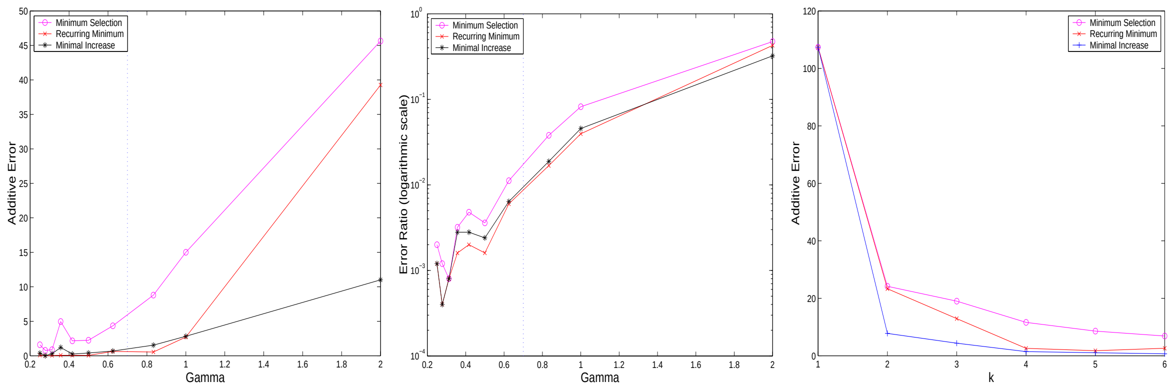

Figure 2: Accuracy of MS, MI and RM algorithms for various values of γ, with $k = 5$, with additive error (a), and log of error ratio (b), dotted line represent optimal γ. Additive errors in the three algorithms for various k values, with $\gamma = 0.7$ (c). In all experiments, MI and RM are better than MS, with some advantage to MI.

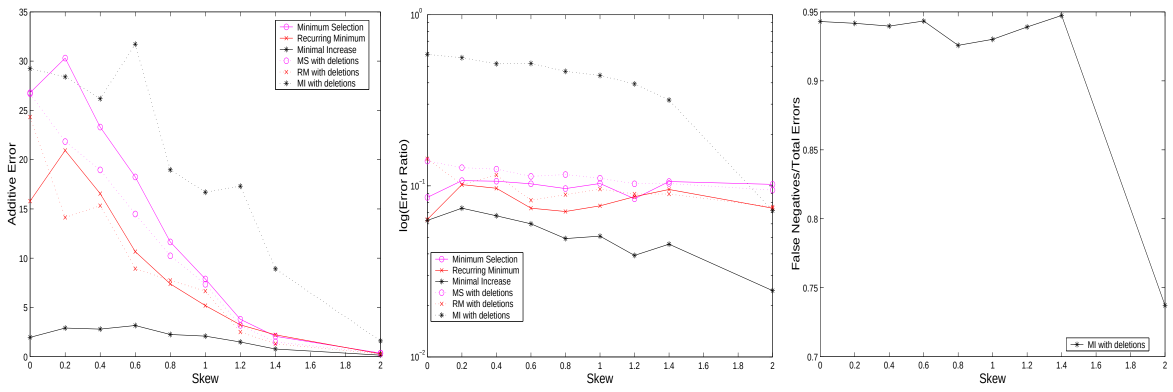

Figure 3: Performance of MS, RM and MI algorithms for Zipfian distribution with varying skew (θ), with deletions (dotted lines) and without deletions (full lines). Both additive error (left) and log of error ratio (center) are shown; in all experiments $\gamma = 0.7, k = 5$. The third graph shows the ratio of False Negative errors in the MI algorithm out of the total errors (there are no false negatives in MS and RM).

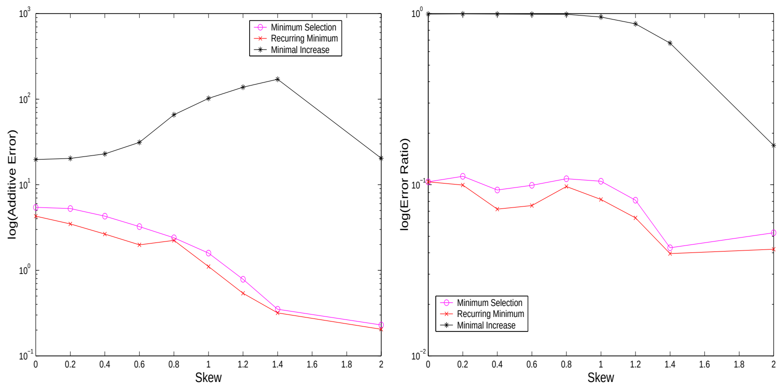

Figure 4: Accuracy of MS, RM and MI algorithms for Zipfian distribution of varying skew (θ), in a sliding window scenario. Both log of additive error and log of error ratio are shown, in all experiments $\gamma = 0.7, k = 5$.

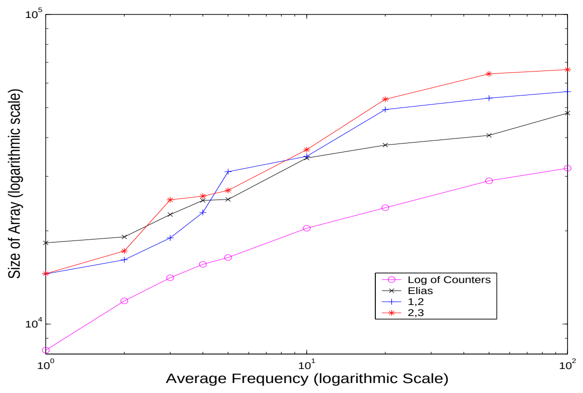

Figure 5: Comparison of various encoding methods. Several "steps" configurations were tested along with Elias encoding. The results are compared to the optimal Log of the counters.

changes in the data-set.

There are several extensions to the basic functionality of the SBF. One property is the ability to union sets effectively, provided that the same parameters are used (hash functions and array size). For such Bloom Filters, a union of two data sets only requires an addition of the counter vectors representing them. The SBF can support both streaming data and sliding window data sets [6], given that old data is available for deletion.

The SBF enables new applications, and enables more effective execution of existing applications. SBFs can be used for maintaining demographics of a multiset or set, and allow data profiling and filtering using an arbitrary threshold. It can be used for ad-hoc iceberg-queries, where the threshold defining the query is not known in construction time, or changes as the data is queried. Bifocal Sampling [12] can use SBF as an index data structure in the sparse-any procedure (in fact, SBF can be used in any join of type *t-index*). The SBF can also be plugged into many applications currently using Bloom Filters. For example, Bloomjoins [17] can be extended using SBF, with better efficiency for many types of queries.

8. REFERENCES

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 21st ACM symposium on Principles of Database Systems*, pages 1–16, 2002.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] A. Broder and M. Mitzenmacher. Network applications of Bloom Filters: A survey. In *Proc. of Allerton Conference, 2002.*

[4] A. Z. Broder. Personal communication.

[5] S. Cohen and Y. Matias. Spectral bloom filters, Technical Report. Tel Aviv University, 2003.

[6] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. of the 13th annual ACM-SIAM symposium on Discrete algorithms*, pages 635–644, 2002.

[7] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–202, 1975.

[8] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proc. of ACM SIGCOMM*, 2002.

[9] L. Fan, P. Cao, and J. Almeida. A prototype implementation of summary-cache enhanced icp in squid 1.1.14. www.cs.wisc.edu/∼cao/sc-icp.html.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *ACM SIGCOMM Computer Communication Review*, 28(4):254–265, 1998.

[11] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. of 24th International Conference on Very Large Data Bases, VLDB*, pages 299–310, 1998.

[12] S. Ganguly, P. B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *Proc. of the 1996 ACM SIGMOD International Conference on Management of data*, pages 271–281, 1996.

[13] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 331–342, 1998.

[14] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In J. Abello and J. Vitter, editors, *External Memory Algorithms*, volume 50, pages 39–70. AMS, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, 1999. A two page summary appeared in SODA'99.

[15] L. L. Gremillion. Designing a bloom filter for differential file access. *Communications of the ACM*, 25(9):600–604, 1982.

[16] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Fixed-precision estimation of join selectivity. In *Proc. of the 12th ACM Symp. on Principles of Database Systems*, pages 190–201, 1993.

[17] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. of 12th International Conference on Very Large Data Bases, VLDB*, pages 149–159, 1986.

[18] U. Manber and S. Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4):191–197, 1994.

[19] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. of the 28th International Conference on Very Large Data Bases, VLDB*, 2002.

[20] Y. Matias. Bloom Histograms, July 2001.

[21] M. Mitzenmacher. Compressed bloom filters. In *Proc. of the twentieth annual ACM symposium on Principles of Distributed Computing*, pages 144–150, 2001.

[22] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[23] Squid Web Proxy Cache. http://www.squid-cache.org.