TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

Spectral Bloom Filters

Thesis submitted in partial fulfillment of the requirements for the
M.Sc. degree in the School of Computer Science, Tel-Aviv University

by

Saar Cohen

The research work for this thesis has been carried out at
Tel-Aviv University
under the supervision of Prof. Yossi Matias

September 2003

Acknowledgments

I wish to thank my advisor Prof. Yossi Matias, for his immeasurable assistance and helpful directions throughout the work on this thesis.

I wish to thank my wife Shirley, for her unlimited support and infinite patience.

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 thesis 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 (the *String-array index*), 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.

The SBF, and the String-array index data structure are both efficient and fairly easy to implement, which make them a very practical solution to situation in which filtering of a given spectrum are necessary. The methods proposed and the data structure were fully implemented and tested under various conditions, testing their accuracy, memory requirements and speed of execution. Those experiments are reported within this thesis, as well as analysis of the expected behavior for several common scenarios.

Contents

1 Introduction **5**
1.1 Previous work 6
1.1.1 Distributed processing 6
1.1.2 Filtering and validation 7
1.1.3 Extensions and improvements 8
1.1.4 Iceberg queries and streaming data 8
1.1.5 Succinct data structures 9
1.2 Contributions 10
1.3 Thesis outline 11

2 Spectral Bloom Filters **12**
2.1 The Bloom Filter 12
2.2 The Spectral Bloom Filter 13
2.3 Minimum Selection error analysis for Zipfian Distribution . . 16

3 Estimation Optimizations **21**
3.1 Probabilistic Estimator 21
3.1.1 Boosting the variance 23
3.2 Minimal Increase 25
3.3 Recurring Minimum 27
3.3.1 The Trapping Recurring minimum algorithm 30
3.4 Methods comparison 32

4 Data structures **33**
4.1 The variable length access problem 34
4.2 Current known solutions 34
4.3 The String-Array Index 35
4.4 Handling updates 38
4.5 An alternative approach 40

4.6 Storage requirements improvement 41
4.6.1 String-array index memory reduction 41
4.7 Implementation issues 43
4.7.1 Memory management 43
4.7.2 Offset vectors division 46

5 Applications 48
5.1 Aggregate queries over specified items 48
5.2 Ad-hoc iceberg queries 49
5.3 Spectral Bloomjoins 53
5.4 Bifocal sampling 55
5.5 Range queries 56

6 Experiments 60
6.1 Algorithms comparisons 60
6.2 Deletions and sliding window 63
6.3 Encoding methods 63
6.4 String-array index performance 67

7 Conclusions 74

Chapter 1

Introduction

Bloom Filters are space efficient data structures which allow for membership queries over a given set [Blo70]. The Bloom Filter uses k hash functions, $h_1, h_2, \ldots, h_k$ to hash elements from a set S into an array of size m. For each element $s \in S$, 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 S by examining the bits at positions $h_1(q), h_2(q), \ldots, h_k(q)$. The item q is reported to be in S if (and only if) all the bits are set to 1. This method allows a small probability of a false positive error (it may return a positive result for an item which actually is not contained in S), but no false-negative error, while gaining substantial space savings. Bloom Filters are widely used in many applications.

This thesis 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.

Unlike the standard Bloom Filter, which uses a straight-forward approach to storage (a bit vector), the SBF is by nature more complex. Since

counters have to be stored in an economical fashion, a major consideration is the ability to hold, update and access the information in an efficient and compact manner. To do so, this thesis presents the String-Array Index data structure, fulfilling these requirements. We also propose and analyze methods for querying the SBF, improving over the standard lookup scheme and reducing the error probability and size.

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 [GM99] and data streams [AMS99, BBD$^+$02].

The applicability of Bloom Filters as an effective, compact data representation is well recognized. In this section, we briefly survey several major applications of Bloom Filters. These uses include peer-to-peer systems, distributed calculations and distributed database queries and other applications. Several modifications have also been published over the basic Bloom Filter structure, optimizing the performance and storage for different scenarios.

1.1.1 Distributed processing

Bloom Filters are often used in distributed environments to store an inventory of items stored at every node. In [FCAB98], Bloom Filters are 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 [FCA, Squ], with a variation of this algorithm called Cache Digest implemented in a later version of Squid. In this scenario, the Bloom Filters are exchanged between nodes, creating an efficient method of representing the full picture of the items stored in every proxy among all proxies.

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 [RK02], 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. The main difference between this method and the Summary Cache algorithm is that in this article, the notion of distance and route between nodes is taken into consideration, while in [FCAB98], every remote node reachable (and whose data is maintained) in every node is considered to be within the same distance from the originating node.

A different aspect of distributed processing is distributed database systems. In such system, the data is partitioned and stored in several locations. Usually, the scenario in question involves several relations which reside on different locations, and a query that requires a join between those relations. The use of Bloom Filters was proposed in handling such joins. Bloomjoin is a scheme for performing distributed joins [ML86], 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 together with the ability to perform strong filtering of the results during the execution of the query saves significant transmission size while not sacrificing accuracy (as the results can be verified by checking them against the real data).

1.1.2 Filtering and validation

Bloom Filters were proposed in order to improve performance of working with Differential Files [Gre82]. A differential file stores changes in a database until they are executed as a batch, thud reducing overheads caused by sporadic updates and deletions to large tables. However, 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 differential file itself. Since every query and update must consider the contents of the differential file, having an efficient method to prevent unnecessary file probes improves performance dramatically.

Another area in which Bloom Filters can be used is checking validity of proposed passwords [MW94] against previous passwords used and a dictionary. This method can quickly and efficiently prevent users from reusing old passwords or using dictionary words. Recently, Broder et al [Bro02] used

Bloom Filters in conjunction with hot list techniques presented in [GM98] to efficiently identify popular search queries in the Alta-Vista search engine.

1.1.3 Extensions and improvements

Several improvements have been proposed over the original Bloom Filter. Note that in many distributed applications (such as in Summary Cache [FCAB98]), the Bloom Filters are used rather as a message within the system, sent from one node to the other when exchanging information. In [Mit01] 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. It is easily shown that a Bloom Filter that is space-optimized is characterized by its bit vector being completely random (see Section 2.1), which makes compression inefficient and at times useless. The article shows that by maintaining a locally larger Bloom Filter, it is possible to achieve a compressed version of the bit array which is more efficient.

A modification proposed in [MW94] is imposing a locality restriction on the hash functions, to allow for faster performance when using external storage. This improvement tends to localize queries to consecutive blocks of storage, allowing less disk accesses and faster performance when using slow secondary storage. In [FCAB98] 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 trying to encode the frequencies of items within multi-sets, in which items may easily appear hundreds and thousands of times.

1.1.4 Iceberg queries and streaming data

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 [FSGM+98] and tracking large flows in network traffic [EV02]. 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. One of the major differences between the articles is that the former assumes the data is available for queries and scanning, while the latter assume a situation of streaming data, in which the information is available only once, as it arrives, and cannot be queried afterwards. This situation is very common in network applications, where huge amounts of data flow rapidly and need to be handled as it passes. Usually it is not possible to store the entire data as it flows, and therefore it is not possible to perform retroactive queries over it. A recent survey describes several applications and extensions of the Bloom Filter, with emphasis on network applications [BM02].

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 thesis 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 [Mat].

1.1.5 Succinct data structures

The Bloom Filter is an instance of a succinct data structure that addresses membership queries over a data set, while being as compact and efficient as possible. In this sense, the Bloom Filter is a synopsis data structure, which aims to solve a given problem while emphasizing on compactness. The literature contains a broad selection of such data structures which address common problems. Within this work, we define and address the variable length access problem which can be easily reduced to the *select* problem. The *select* problem deals with building a data structure over a bit vector V such that for an index i, it returns the index within V of the ith 1 bit.

Known solutions to the *select* problem allow $O(1)$ time lookups using $o(N)$ bits of space [Jac89, Mun96]. However, these solutions handle the *static* case, in which the underlying bit vector does not change during the lifespan of the data structure. In the general case, this is an adequate solution to the access problem we are facing, but it fails to meet the demands for updates, which are mandatory for our implementation of the SBF. Solutions which support updates use the same amount of space, and given a parameter $b \geq$

$\log N / \log \log N$, support *select* in $O(\log_b N)$ time, and update in amortized $O(b)$ time [RRR00]. Specifically, *select* can be supported in constant time if update is allowed to take $O(N^\epsilon)$ amortized time, for $\epsilon > 0$.

It should be noted that the solutions given to the *select* problem are rather complicated and are difficult to implement, as pointed out in [Jac89]. In Section 4 we present our solution for the variable length access problem, consisting of a novel data structure - the String-Array Index. This structure is a fairly simple structure and arguably practical, as demonstrated in our implementation and the experiments conducted during this work. We also present a method to support updates, which appears to be practical in the context of current methods as well.

1.2 Contributions

This thesis 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 thesis 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 at query time. Another application 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 [EV02] 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 Thesis outline

The rest of this thesis 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.

Chapter 2

Spectral Bloom Filters

This section reviews the Bloom Filter structure, as proposed by Bloom in [Blo70]. 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 [Blo70].

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 equal to one even though $q \notin S$, if other keys from S were mapped into these positions. We denote such an occurrence *bloom error*, and denote its probability 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 bits. 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 to 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 $\hat{f}_x = m_x$.

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

Proof. Since for each insertion of x, all its counters are increased, then it is clear that $m_x \geq f_x$. The case of inequality is exactly the situation of a Bloom Error as defined for the simple Bloom Filter, where all counters are stepped over by other items hashing to the same positions in the array, and therefore has the same probability E_b. □

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. This happens frequently in distributed data base systems, where a single relation is partitioned to several sites, each containing a fraction of the entire data-set. A query directed at this relation will require processing of the data stored within each site, and then merging the results into a final answer. When such 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.

Queries over joins of sets

Applications which allow for joins of sets, 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 [MW94], a multi-level hashing scheme was proposed for Bloom filters, 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 [MW94] 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

There are several issues which are particular to the SBF and need to be resolved for this data structure. The first issue is *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. The other is *query performance*, with respect to *two* error metrics: the error rate (similar to the original Bloom Filter), and the size of the error.

2.3 Minimum Selection error analysis for Zipfian Distribution

Using the MS algorithm yields an error with probability of $E_b \approx (1-e^{-\gamma})^k$. For membership queries, this provides a full description of the error, since its size is fixed. However, when answering count-estimate queries, we need to address the issue of the size of the error in the estimate, and provide an estimate to this quantity. We cannot provide such an estimate for arbitrary data set, since the size of the error is directly dependent on the distribution of the data inserted into the SBF. An item with a very small frequency (or even frequency of 0) might get its counters stepped over by the k most frequent items in the dataset, causing an error whose size is unknown without further knowledge of the distribution.

It is common for real-life data sets to demonstrate a Zipfian distribution [Zip49]. We provide analytical results regarding the size of the errors by analyzing data which is distributed according to Zipf's law. This is based on the fact that most data-sets can be described by such distribution, using the correct parameters. In a Zipfian distribution, the probability of the ith most frequent item in the data-set to appear is equal to $p_i = c/i^z$, with c being some normalization constant, and z is the Zipf parameter, or skew of the data. For data with a total of N items, the expected frequency of item i is therefore $f_i = Nc/i^z$. From now on, we assume that the frequencies are sorted in descending order, such that f_i is the frequency of the ith most frequent item, and for every $i < j$ we have $f_i \geq f_j$.

The calculations in this section all assume that a situation of Bloom error has occurred. We only deal with figuring out the size of the error stemming from that situation. We also assume that for the ith item, which is subject to error, each of its k counters is shared with no more than one other item. This implies that there is no situation where the size of the error is the accumulating frequency of two or more items. This assumption is required only for the counter which is subject to the smallest error, since other counters do not participate in the calculation of the estimated frequency of i.

The probability for a single counter to be subject to at least two items stepping over it is $E' = 1-(1-1/m)^{Nk} - Nk(1/m)(1-1/m)^{Nk-1}$, with $(1-1/m)^{Nk}$ representing the probability that no item stepped over it, and the second term is the probability that exactly one item steps over it. Some algebraic manipulations transform this probability to $E' \approx 1-e^{-\gamma}(1+\frac{\gamma m}{m-1})$. The probability that an item is subject to a Bloom error with one counter having two items stepping over it is therefore $E' \cdot (1-e^{-\gamma})^{k-1}$, which for

$\gamma = 0.7$ and $k = 5$ yields a probability of less than 1%. This is a bound on the actual probability of interest, since in most cases the counter subject to a double error will not be the minimal counter, because of the accumulating error. Thus, the expected probability of that event is significantly smaller than the probability for a Bloom error, and therefore we ignore it in the remainder of this discussion.

We state the following lemma, concerning the distribution of the relative error in Zipfian distribution:

Lemma 2. *Let S be a multi-set with n distinct items taken at random from a Zipfian distribution of skew z, hashed into a SBF. Let T be a threshold $T > 0$, and let RE_i^z be the relative error for the ith most frequent item in S, $RE_i^z = (m_i - f_i)/f_i$. Given that $RE_i > 0$, the probability of this relative error exceeding T is*

$$P(RE_i^z > T) \leq k\left(\frac{i}{(n-k)T^{1/z}}\right)^k$$

Proof. We begin our proof by calculating the expected relative error for the ith most frequent item in the data. First, we note that the error for an item is the frequency of the least-frequent item which shares its counters. If that item is the jth most frequent item, for a skew of z, the relative error is

$$RE_{ij}^z = f_j/f_i = i^z/j^z$$

This calculation can be used to bound the relative error. For data with n distinct items, the maximal relative error is $RE_{nk} = (n/k)^z$. For example, for data with 1000 distinct items, skew of 1 and 5 hash functions, this amounts to 200, which is 20000%. Luckily the probability of such an event is very small.

In order to calculate the distribution of errors, we need to calculate the probability $P(j)$ that for any item i, the least frequent item that shares its counters is j. For that purpose, we note that there are $\binom{n-1}{k}$ ways to choose k items which step over i. Out of which, only combinations in which $k-1$ items are in the range $(1 \ldots j-1)$, and the kth item is j will produce the probability we are looking for. The number of these combinations is $N(j) = \binom{j-1}{k-1}$ So the probability $P(j)$ is

$$\begin{aligned} P(j) &= \frac{\binom{j-1}{k-1}}{\binom{n-1}{k}} = \frac{(j-1)!}{(k-1)!(j-k)!}\frac{k!(n-k-1)!}{(n-1)!} = \\ &= k\frac{(n-k-1)!}{(n-1)!}\frac{(j-1)!}{(j-k)!} \end{aligned}$$

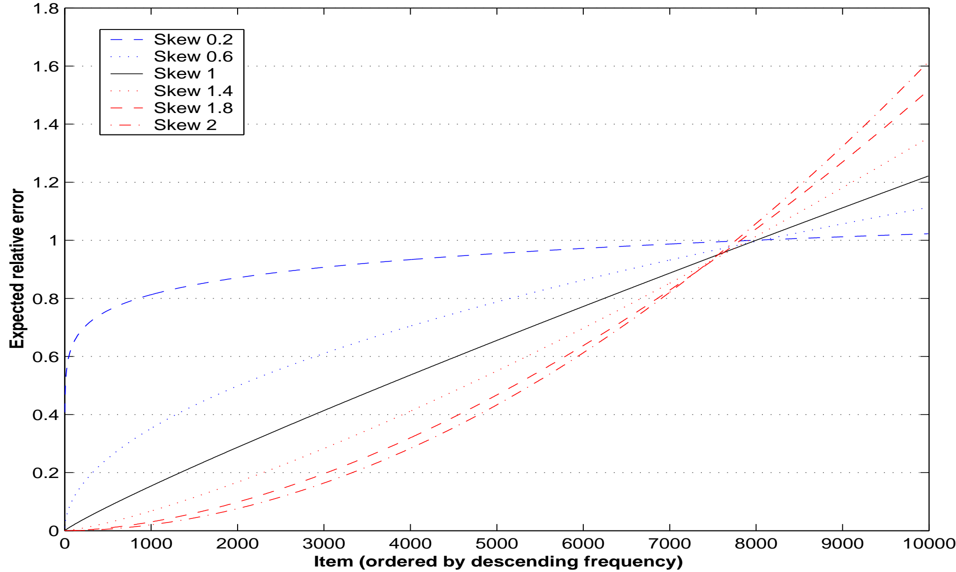


Figure 2.1: Estimate on the expected relative errors $E'(RE_i^z)$ for data set items ordered by decreasing item frequencies. Shown for data sets with Zipfian distribution of several skews ($z = 0.2, 0.6, 1, 1.4, 1.8, 2$).

The expected relative error for the ith most frequent item is

$$\begin{aligned} E(RE_i^z) &= \sum_{j \neq i} RE_{ij}^z P(j) \\ &= i^z k \frac{(n-k-1)!}{(n-1)!} \sum_{j \neq i} \frac{1}{j^z} \frac{(j-1)!}{(j-k)!} \\ &< i^z \frac{k}{(n-k)^k} \sum_{j \neq i} j^{k-z-1} \end{aligned} \tag{2.1}$$

Let $S_z = \sum_j j^{k-z-1}$. The above calculation shows that we can bound $E(RE_i^z)$ by $E'(RE_i^z) = i^z \frac{k}{(n-k)^k} S_z$. Within $E'(RE_i^z)$ there are two quantities that depend on z: the first is S_z, which is constant per skew; the other is i^z which determines the shape of the function when testing it for various items over a given skew. Figure 2.1 shows this function for several skews over data with 10,000 distinct items.

The graphs shown have several distinctive properties. The first one is that this function is rising monotonically as items are less frequent in the data set. This property is intuitive, since as the frequency of the item

decreases, the ratio between the frequency of item and the frequency of the items causing the error diminishes. Another observation is that as the skew increases, the expected error for the frequent items becomes smaller. However, the graphs show that there is a crossover point, where for less frequent items, the expected error for high skews rises above the error of lower skewed data sets. This crossover point stems from the tradeoff between two factors: as the skew increases, there are less items with high frequency in the data set, however, the ratio between the frequency of those items and the frequency of the least frequent items increases too as the skew increases.

In order to get a simple expression for S_z we can use the fact that all the indices are positive. For $k - z - 1 > 0$ we can use the following calculation:

$$\begin{array}{rcl} \int_{x-1}^{x} y^i dy < & x^i & < \int_{x}^{x+1} y^i dy \\ \int_{0}^{n} y^i dy < & \sum_{x=1}^{n} x^i & < \int_{1}^{n+1} y^i dy \\ \frac{n^{i+1}}{i+1} < & \sum_{x=1}^{n} x^i & < \frac{(n+1)^{i+1}-1}{i+1} \\ \frac{n^{k-z}}{k-z} < & S_z & < \frac{(n+1)^{k-z}-1}{k-z} \end{array}$$

Hence, we have

$$E(RE_i^z) < i^z \frac{k}{(n-k)^k} \cdot \frac{(n+1)^{k-z}-1}{k-z} < i^z \frac{k}{k-z} \cdot \frac{(n+1)^{k-z}}{(n-k)^k}$$

And finally, we can calculate the expected relative error over all items distributed with a given skew z:

$$\begin{aligned} E(RE^z) &< \frac{1}{n} \sum_{i=1}^{n} i^z \cdot \frac{k}{k-z} \cdot \frac{(n+1)^{k-z}}{(n-k)^k} \\ &< \frac{1}{n} \cdot \frac{k}{k-z} \cdot \frac{(n+1)^{k-z}}{(n-k)^k} \cdot \frac{(n+1)^{z+1}}{z+1} \\ &= \frac{k \cdot (n+1)^{k+1}}{n(k-z)(z+1)(n-k)^k} \end{aligned} \qquad (2.2)$$

This last result is a nonlinear function which has a minimal value with respect to z. Simple derivative shows that the minimum is achieved when $z_{min} = (k+1)/2$, and that the minimal value is

$$E(RE^{z_{min}}) < \frac{4k \cdot (n+1)^{k+1}}{n(n-k)^k(k-1)(k+3)}$$

For the item whose rank is i, we can calculate the probability that the relative error for that item will be below a given threshold, $RE_i^z \leq T$. That is, $i^z/j^z \leq T$ or $j \geq i/T^{1/z}$. The probability of a relative error which is higher than T is

$$\begin{aligned}
P(RE_i^z > T) &= \sum_{j=k}^{i/T^{1/z}} P(j) \\
&= \sum_{j=k}^{i/T^{1/z}} k \frac{(n-k-1)!}{(n-1)!} \frac{(j-1)!}{(j-k)!} \\
&\approx \sum_{j=k}^{i/T^{1/z}} k \left(\frac{j}{n-k}\right)^k \leq k \left(\frac{i}{(n-k)T^{1/z}}\right)^k .
\end{aligned}$$

□

To summarize, this analysis yields three interesting results:

- The expected relative error for the ith most frequent item, $E(RE_i^z)$, shown in Equation (2.1) and Figure 2.1.
- The expected relative error for all items distributed with a skew z, shown in Equation (2.2). This result has a minimum for $z_{min} = (k+1)/2$, and therefore can lead to selection of SBF parameters when expecting a certain skew.
- The final result, expressing the probability for relative errors passing any threshold.

To demonstrate the properties of the last result, we calculate it for possible real-life parameters. For instance, by setting values of $n = 1000, k = 5, z = 1$ and $T = 0.5$ (errors of less than 50% of the real value), we get $P(RE_i > 0.5) \leq 5\left(\frac{i}{497.5}\right)^5$, which has values bigger than 1 for $i > 360$.

Again, the basic fact that has to be remembered is that in these calculations we assumed that a Bloom Error has occurred. Remember that the probability for a Bloom Error is $E_b \approx (1-e^{-\gamma})^k$, which in the optimal case, for those values yield $E_b \approx 0.03$.

Chapter 3

Estimation Optimizations

In this section we present methods to improve the accuracy of queries performed over the SBF. The first method is statistically interesting, since it provides an unbiased estimator for the frequency of an item. In practice, it fails to produce good results for individual queries, but may produce good results for aggregative queries, due to its unbiased nature. Then 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. These methods are the Recurring Minimum method (or RM), and the Minimal Increase method (MI). For membership queries (i.e., threshold equals 1), the error remains unchanged.

3.1 Probabilistic Estimator

In many cases, an *unbiased estimator* to a given probabilistic value is a valuable tool. This is especially true when measuring aggregate values such as `sum`, `avg` etc. since the expected error size is zero, we get better aggregate results as the number of queries increase. However, unbiased estimators do not ensure a small variance, and may produce results that average well, but are individually inaccurate.

In the case of the SBF, an unbiased estimator may be important for a specific type of queries, mainly aggregate ones. For individual queries, such an estimator is problematic, since the errors produced by the SBF are by nature one-sided. When using such an estimator, it reduces the estimate error for items which are subject to Bloom Error. On the other hand, it introduces an unneeded fix for items which are initially accurate. Therefore,

the estimator produces *false negative* errors for those items, which is highly undesirable in most cases.

In order to produce an unbiased estimator, we find the average error imposed on the counters by the other items being mapped to the same locations. Assuming that the hash functions are uniformly random, we perform an analysis of this effect. The resulting estimator is described in the following Lemma:

Lemma 3. *For any $x \in U$, the estimate $\bar{f}_x = \frac{\bar{v}_x - \frac{kN}{m}}{1-k/m}$ is an unbiased estimator for f_x.*

Proof. Let x be an item in the set that is mapped into the SBF. For $1 \leq j \leq k$, we can determine the error of the jth bit with regards to x, denoted e_x^j by $e_x^j = v_x^j - f_x$. When hashing another item y into the SBF, it can be considered as hashing k "bundles" into the array, each of size f_y. The contribution of one such bundle to any given counter is f_y with probability $1/m$, and 0 otherwise. The total contribution of the k "bundles" to the jth counter in the array is therefore $S_y^j = f_y \cdot B(k, 1/m)$. Summing over all the items (other than x) in the set, we get the expected error for a given counter, which is equal to the total contribution to its count, expressed by

$$e_x^j = E(\sum_{y \neq x} S_y^j) = (N - f_x)k/m$$

Using this result, we can estimate the actual frequency of x by calculating $\bar{f}_x = v_x^j - (N - f_x)k/m$. Substituting f_x with $\bar{f}_x$ in this calculation, we get:

$$\begin{aligned} \bar{f}_x &= v_x^j - \frac{k}{m}(N - \bar{f}_x) \\ \bar{f}_x(1 - \frac{k}{m}) &= v_x^j - \frac{kN}{m} \\ \bar{f}_x &= \frac{v_x^j - \frac{kN}{m}}{1 - k/m} \end{aligned}$$

And by averaging over the k bits of x, we get that

$$\bar{f}_x = \frac{\bar{v}_x - \frac{kN}{m}}{1 - k/m}$$

To prove that this is indeed an unbiased estimator, we show that $\forall x$, $E(\bar{f}_x) = f_x$. To prove that, we note that the expected value of each of x's k counters

is f_x plus the average error per counter, i.e. $\bar{v}_x = f_x + \frac{kN - kf_x}{m}$:

$$E(\bar{f}_x) = \frac{f_x + \frac{kN-kf_x}{m} - \frac{kN}{m}}{1 - k/m} = \frac{mf_x + kN - kf_x - kN}{m-k} = \frac{f_x(m-k)}{m-k} = f_x$$

□

3.1.1 Boosting the variance

As mentioned, this estimate is problematic because of its rather high variance. Since the total error for a given counter is Binomial, the variance of that error is $Var(e_x^j) = (N - f_x)k/m(1 - 1/m) \approx (N - f_x)k/m$, so the variance almost equals the expected size of the error. We can use the fact that we have k counters to try and reduce the variance by dividing the k counters into k_2 groups of k_1 variables, calculating the average over each of the k_2 groups and then taking the median of these results [AMS99]. When averaging over k_1 variables, the variance is divided by k_1. By Chebyshev:

$$P(|e_x^j - E(e_x^j)| \geq t) \leq \frac{Var(e_x^j)}{t^2} = \frac{\frac{N-f_x}{m}(1-1/m)\frac{k}{k_1}}{t^2} \leq \frac{N}{mt^2}\frac{k}{k_1}$$

Now, we assume that this value equals $1/4$. Given the lth counter within a group of k_2 counters, we define I_l to be an indicator that the error over that counter exceeds the distance of t from its expectancy. We define I to be the sum of those indicators:

$$\forall l, 1 \leq l \leq k_2 : I_l = \begin{cases} 1 & p = 3/4 \\ 0 & p = 1/4 \end{cases} \qquad I = \sum_l I_l$$

I is a binomial variable $I = B(k_2, 3/4)$, with an average of $\frac{3k_2}{4}$. We want to calculate the probability of I being lower than $k_2/2$, since this will mean that the median is within t from the expectancy. By Chernoff:

$$\begin{aligned} P(I < (1-\delta)\mu) &< e^{\frac{-\mu\delta^2}{2}} \\ (1-\delta)\frac{3k_2}{4} = k_2/2 &\Rightarrow \delta = 1/3 \\ P(I < \frac{k_2}{2}) &< e^{\frac{-3k_2}{4}\frac{1}{9\cdot 2}} = e^{\frac{-k_2}{24}} \end{aligned}$$

This analysis shows that indeed the variance can be controlled by increasing the number of counters. However, when confronted with real-life parameters, it can be seen that this approach is not practical in all cases. The calculation

implies that when allowing an error rate of ϵ (an error meaning that the estimate is not within t of the expected value) we need to have $k_2 = 24 \ln 1/\epsilon$. For error of 0.1, this gives a k_2 of 55 which is not very practical. On top of this, we still need to ensure that $\frac{N}{mt^2}\frac{k}{k_1} = 1/4$, meaning that $k_1 = \frac{4Nk}{mt^2}$. Since we require that $k_1 < k$, we require that $4N/mt^2 < 1$, so as N increases we can only support larger values of t. If, for example, we allow $t = 4$, N cannot exceed $4m$.

The scenario in which it may be useful is when aggregating over a large number of results, where the increased number of variables is translated into a decrease in the expected variance of the calculation. The actual size of the groups that need to be aggregated for an accurate estimate depends on the distribution of the data. According to this analysis, is it impractical to effectively reduce the variance of the unbiased estimator per query. However, this analysis only shows a bound on the probabilities in question. Thus, in real-life situations this method might yet produce good results.

Discussion The estimator is based on reducing a fixed amount from every count recorded in the SBF. This approach has two major drawbacks:

- The majority of counters within the SBF are in fact accurate (depending on the parameters on the SBF). These counters need no fix, and in fact will be harmed by introducing the correction.

- The errors of the SBF are one-sided. By introducing the fixed correction, we cannot guarantee this property anymore. All counters whose error rate is below the average error will turn into false-negatives.

Since it addresses the average case, the estimator applies a constant fix to the average of the counters. This becomes a major problem when dealing with highly skewed data. Since the estimator is averaging by nature, the higher the skew (and the deviation from the average), the higher the error will be. Because the fix applied does not take into account the actual value of the counters, a few frequent items can create an error that will be reflected in the estimation of all of the small values (which will be the majority of the data in a very skewed data). The main problem of this estimator is that it ignores completely the nature of the Bloom Filter, namely the fact that the counters are not correct with the same probability. Since the minimum of the counters is an upper bound on f_x, it is only natural to give more attention to the smaller counters and ignore the larger counters.

To improve this estimator, it may be combined with the recurring minimum heuristic (described in section 3.3), which serves as an indication for a

possible error. The Recurring Minimum method allows us to recognize potential problematic cases (i.e. counters that are erroneous), in which cases we might activate the unbiased estimator to produce an estimate. In all other cases we do not use the estimator, and thus refrain from generating false-negative errors.

An unbiased estimator may still be of use for aggregate queries. In these queries we do not worry about the high variance of the estimator or its tendency to produce false-negatives, since the only important factor is the average result over the set of queries performed. For all other scenarios, the unbiased estimator has poor performance, and in fact is a good example of a case in which unbiased does not imply successful.

3.2 Minimal Increase

The Minimal Increase (MI) algorithm is based on a simple observation: 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 update every other counter to the maximum of its old value and $m_x + r$.

A similar method was devised independently in [EV02], 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 4 (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.*

Proof. First, it is clear that the MI method generates no new errors, compared to the Minimum Selection method, as to facilitate an error, an item

must have all counters shared with other items. Now we examine the case where the MS algorithm fails, which is the usual bloom error, i.e. an item x has items $Y = \{y_1, y_2 \ldots, y_k\}$ each sharing one of its counters, all with frequency larger than 0 in the set. It is possible for a counter to be "stepped over" by more than one item, in which case we replace those items with a virtual item whose frequency is the sum of their original frequencies in the data-set. The size of the error for x in the MS algorithm is $E_x^{MS} = \min(f_{y_1}, f_{y_2}, \ldots, f_{y_k})$. In the MI algorithm, the ith counter cannot be larger than $f_{y_i} + f_x$, due to its method of operation. Therefore, the minimal counter will have a count of $E_x^{MS} + f_x$, and $E_x^{MI} = E_x^{MS}$. It is thus clear that the MI algorithm is at least as good as the MS algorithm in terms of confidence and error size. □

Note that the Minimal Increase heuristic produces the minimal number of insertions into the SBF, still maintaining the property that for each item x, $m_x \geq f_x$. It generates no unneeded insertions, and therefore creates a compact and efficient, while accurate, data structure.

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 5. *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.*

Proof. In the uniform case, an error occurs when all items in Y appear at least once before x appears. Assuming that the data is uniform and $f_x = f_{y_1} = \ldots = f_{y_k} = F$, using the MS algorithm, the error on x will be exactly F. Using the MI method, with random positioning of items, we assume here for simplicity that the entire sequence is made out of F subsequences, each containing all item in $\{Y \cup x\}$ once in random order. For each such sequence, it will contribute to the error on x only if x appears last in the sequence. The probability for x to appear last is $1/k$, and the total error expectancy is thus F/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.

3.3 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 3.1 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 3.2 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

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 3.1: 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 3.2: Effect of increased memory for primary SBF and secondary SBF, with original $k = 5$.

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 3.1 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).

3.3.1 The Trapping Recurring minimum algorithm

A common type of error when using the Recurring Minimum algorithm is the scenario of *late detection*. In this event, the item x is recognized as having a single minimum only after all its counters were contaminated. This scenario can be handled by using slightly more storage. In this refinement, each bit has a "trap" attached to it, namely one bit that flags a possibly "stepped over" bit. A lookup table L maps each trap to its associated item. The idea the algorithm uses is that once an item is transferred to the secondary SBF, its minimal counter's trap is set. The trap is associated with that item. If later on another item steps on that trap, its frequency is reduced from the value transferred to the secondary SBF, to compensate for errors which were not detected earlier. The algorithm is shown in Figure 3.1.

This more complex algorithm might compensate for errors by recognizing which item steps over x's bits and fixing the minimum values accordingly. However, it still does not cover all possible cases. Notice that for the value to be fixed, the item y, which stepped over x must appear again in the data after x being transferred to the secondary SBF.

The following condition will cause errors when using this algorithm:

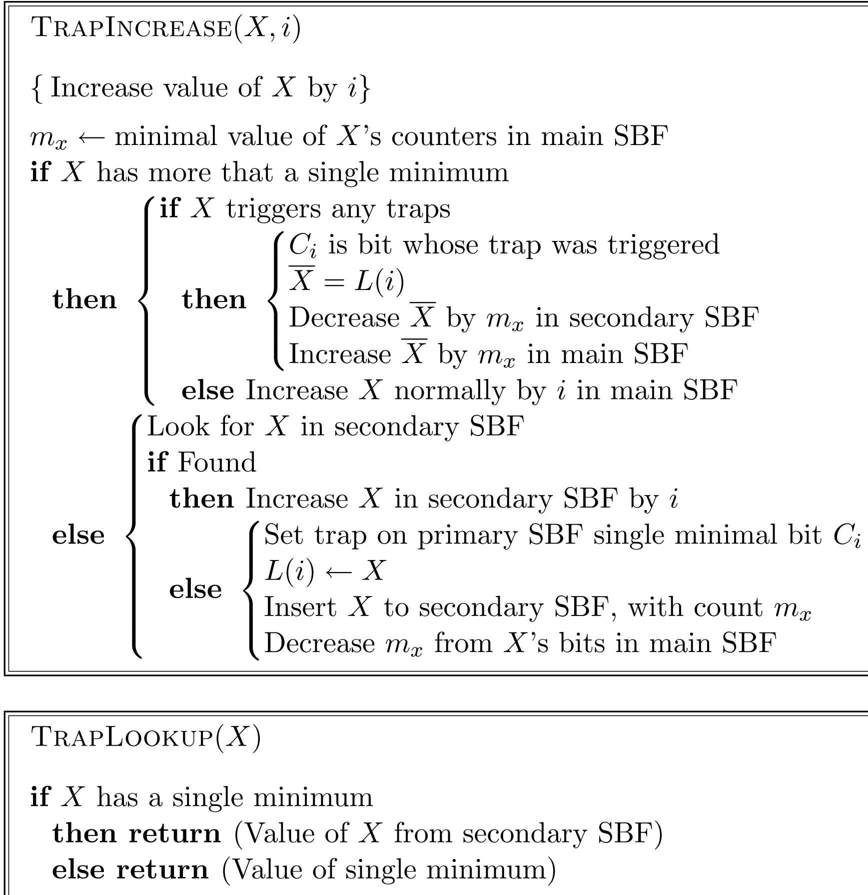
- y not appearing after x was transferred to the secondary SBF. Consider this palindrome:

  $$v_1, v_2, v_3 \dots v_{n/2}, v_{n/2}, v_{n/2-1} \dots v_1$$

  In this sequence, for each i, after the first appearance of v_i, all of the items $v_{i+1} \dots v_{n/2}$ appear twice. Then v_i appears again and is possibly sent to the secondary SBF, and activates the trap. However, this trap will never be triggered and the error will never be recovered.

- Two bits are stepped over with the same counters, such that the minimum is not correct but is repeated twice.

Notice that these errors are very rare. The Palindrome case is a specific pathological case. Usually we can expect that either y is frequent, meaning that the error potential is large, but since y is frequent it will most likely appear again and trigger the trap; or y can be rare, not triggering the trap, but causing a small error. In either case the average error imposed due to this event is very small.



TrapIncrease(X, i)

{ Increase value of X by i}

$m_x \leftarrow$ minimal value of X's counters in main SBF
if X has more that a single minimum
  then
    if X triggers any traps
      then
        C_i is bit whose trap was triggered
        $\overline{X} = L(i)$
        Decrease $\overline{X}$ by m_x in secondary SBF
        Increase $\overline{X}$ by m_x in main SBF
      else Increase X normally by i in main SBF
  else
    Look for X in secondary SBF
    if Found
      then Increase X in secondary SBF by i
      else
        Set trap on primary SBF single minimal bit C_i
        $L(i) \leftarrow X$
        Insert X to secondary SBF, with count m_x
        Decrease m_x from X's bits in main SBF

TrapLookup(X)

if X has a single minimum
  then return (Value of X from secondary SBF)
  else return (Value of single minimum)

Figure 3.1: The Trapping Recurring Minimum algorithm

3.4 Methods comparison

We compare the Minimum Selection algorithm with the Recurring Minimum and Minimal Increase 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 of the MI algorithm is 1 to 2 orders of magnitude larger than that of the RM algorithms, for various skews.

Chapter 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 6. *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 4.3 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 4.4 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 4.5, 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. Finally, in section 4.7, we discuss several possible improvements and issues regarding the implementation of this data structure.

4.1 The variable length access problem

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

The 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.

4.2 Current known solutions

The variable length access problem is closely related to the *select* problem, which deals with finding the index of the ith 1 bit within an arbitrary bit stream. It can be reduced into a *select* problem as follows: Create a bit vector V of the same size N, in which all bits are zero except those that are positioned at the beginning of substrings in S, which will contain the value 1. When looking for the beginning of the ith substring in S, we simply have to perform *select*(V, i).

Known solutions to the *select* problem allow $O(1)$ time lookups using $o(N)$ bits of space [Jac89, Mun96], which is an adequate solution to the access problem we are facing. However, these solutions handle the *static* case, in which the underlying bit vector does not change during the lifespan of the data structure. Thus it fails to meet the demands for updates, which are essential for our implementation of the SBF. The best known solutions for *select* with updates use the same amount of space, and given a parameter $b \geq \log N / \log \log N$, support *select* in $O(\log_b N)$ time, and update in amortized $O(b)$ time [RRR00]. Specifically, *select* can be supported in constant time if update is allowed to take $O(N^\epsilon)$ amortized time, for $\epsilon > 0$.

It should be noted that the solutions given to the *select* problem are rather complicated and are difficult to implement. The solution which we

present, namely the string-array index, is a relatively simple structure, which was implemented during this work. In the following sections we describe the structure itself, and then expand the presentation and present several optimizations that make it highly competitive with the current solutions. Our solution also implies a method to perform *select* where items are inserted at random to the bit vector.

4.3 The String-Array Index

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 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 7 (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 4.1; 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 sub-problems. An entry to the lookup

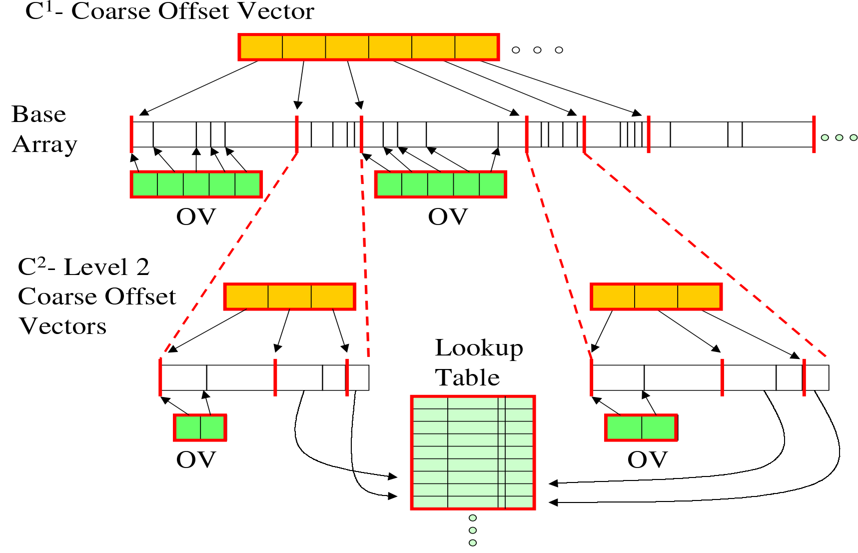


Figure 4.1: The String-Array Index data structure.

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 $\langle L(S''), i\rangle$, where $L(S'')$ is a bit sequence which provides a unique representation of the lengths of items in S''. Note that since we are only interested in obtaining an offset within S'', we need not take into consideration the bit sequence of S'' itself, thus we need to precompute only the possible combinations of counter lengths such that the total length of S'' is $\leq T$. This reduces the number of keys within the lookup table dramatically. The subarray $L(S'')$ consists 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 4.5). The length of $L(S'')$ is either $O(\log\log N)$ or $o(T)$. In addition to the representation $L(S'')$, 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 $\langle L(S''), i\rangle$, 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 6: 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. The 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.

4.4 Handling updates

We show how to extend the string-array index data structure described above, to allow dynamic changes in the data-set, 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 slacks are again placed in $1/\epsilon$ intervals, and the string-array index is updated accordingly.

Lemma 8. *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 $O(1)$. □

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.

4.5 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 that is described in Section 4.4 can be used to allow dynamic maintenance of the structure.

Elias encoding

The Elias encoding [Eli75] 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

[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

overhead of $\log\log n$ bits and the constants that are involved is substantial and should be avoided. For example, to encode the number 1 (actually encoding the number 2) we need 4 bits. In many data-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 a Huffman-like 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, for data-sets as described above. 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).

4.6 Storage requirements improvement

The storage bounds presented in Theorem 6 should be competitive with current known solution to the variable length access problem, presented in Section 4.2. In this section we will propose an improvement to the string-array index structure which reduces its storage requirements and makes it competitive with those methods.

4.6.1 String-array index memory reduction

The key notion that enables the reduction in the memory requirements is that the number of offsets in each offset vector can actually be reduced to create a smaller offset vector. Our goal is to produce a string-array index which, for a bit-array of N bits, requires additional $O(N/\log\log N)$ bits of storage. To reach this goal, we will reduce each and every substructure of the string-array index to within the required space. The following theorem states this formally, and the modifications needed in the string-array index are described in its proof.

Theorem 9. *The string-array index structure for a bit array of N bits, supporting lookups in $O(1)$ time and insertions,deletions and updates in $O(1)$ expected amortized time, can be implemented using $o(N/(\log\log N)^c) + O(m/(\log\log N)^c)$ bits, for any given $c \geq 0$.*

Proof. The following description does not change the structure of the string-array index. The basic building-stones and the structure of the layers are the same, with changes only in the constants and thresholds used in the construction of the data structure. The remainder of this proof describes

the changes made in each of the layers of the string-array index, starting with the first level of coarse offset vectors (C^1), and ending with the lookup table.

In C^1, each offset is of $\log N$ bits. Instead of holding $m/\log N$ such offsets, allocate only $m/(\log N)^{1+c}$ such offsets, resulting in a total storage of $m/(\log N)^c$ bits for C^1. As a result, C^1 divides the bit-array into subgroups of size $(\log N)^{1+c}$ items.

The size of a complete offset vector for such subgroup S' of size T bits is $(\log N)^{1+c} \log T$ bits. Therefore, S' will have a complete offset vector in C^2 if it satisfies $T/(\log\log N)^c > (\log N)^{1+c} \log T$. This is necessary to ensure that the string-array index is smaller by a factor of $(\log\log N)^c$ than the original vector. From this we can derive the constrain on T:

$$T/\log T > (\log N)^{1+c}(\log\log N)^c$$

To find a minimal value for T from this, we use the following claim (all usage of $\log x$ in this claim refers to $\log_2 x$):

Claim 10. *The inequality $T/\log T > \beta$ is satisfied for $T > 3\beta\log\beta$ and $\beta > 3$*

Proof. Let $T' = 3\beta\log\beta \Rightarrow \log T' = \log 3 + \log\beta + \log\log\beta$.

$$\beta\log T' = \beta(\log 3 + \log\beta + \log\log\beta) < 3\beta\log\beta = T'$$

The last inequality is correct for $\beta > 3$. Since the claim is true for T', and the expression $T/\log T$ is increasing with T, the claim follows for $T > T'$. □

To satisfy the above inequality, we require that T satisfies the looser bound:

$$T/\log T > (\log N)^{1+c}(\log N)^c = (\log N)^{1+2c}$$

From Claim 10, this is satisfied when

$$T > T_0' = 3(1+2c)(\log N)^{1+2c}\log\log N$$

and therefore satisfied when $T_0' = (3+6c)(\log N)^{2+2c}$. Notice, however, that this inequality actually allows for the offset vector to be smaller than the original vector by a factor of $(\log N)^c$. When calculating the bound with the original value of $\beta = (\log N)^{1+c}(\log\log N)^c$, this bound can be reduced to $T > T_0' = (3+6c)(\log N)^{1+c}(\log\log N)^{1+c}$.

Subsequences which are smaller than $(3+6c)(\log N)^{2+2c}$ bits are treated with a coarse offset vector in C^2. The size of each offset is

$$\log((3+6c)(\log N)^{2+2c}) = \log(3+6c) + (2+2c)\log\log N$$

In the second level, each subgroup will be divided into subgroups of $(\log\log N)^{1+c}$ items, generating coarse offset vectors of total size $O(m/(\log\log N)^c)$ bits.

The last part of the structure is the third level, consisting of offset vectors and lookup table. Each subgroup in this level consists of $(\log\log N)^{1+c}$ items. Similarly to the calculation shown above, we set a constraint at

$$T/(\log\log N)^c > (\log\log N)^{1+c}\log T \Rightarrow T/\log T > (\log\log N)^{1+2c}$$

By Claim 10 it produces the limit $T_0'' = (3+6c)(\log\log N)^{2+2c}$. Subsequences larger than this limit will use an offset vector, and smaller than it will use the lookup table, which needs to support bit sequences of maximal size T_0''. The table consists of $(\log\log N)^{1+c}\cdot 2^{T_0''}$ entries, each of size $\log T_0''$ bits. This calculation is asymptotically smaller than $N/(\log\log N)^c$ for large enough values of N, meaning we can store the lookup table in $o(N/(\log\log N)^c)$ bits, as required.

This completes the modifications needed to reduce the storage requirements of the string-array index. Given the reduced storage, it is competitive with the various solutions given to the variable length index problem, supporting lookup in $O(1)$ time and update in amortized $O(1)$ time. □

4.7 Implementation issues

During the implementation of the string-array index, the emphasis was on providing the fastest and most efficient implementation available. The implementation also needed to address several issues, and several optimization schemes were thought of during that phase. In this section we give a survey of those issues.

4.7.1 Memory management

Within the string-array index, there is usage of blocks of allocated memory, for the original counter vector, the various offset vectors and the lookup table. A simple implementation allocates space for each and every such memory block individually, using the memory allocation scheme of the given compiler. This method often creates fragmented memory area, in which the

memory is not allocated as one continuous block, but is spread across the available memory.

One of the popular uses of Bloom Filters is in distributed systems, where the filter is often sent from one node to another as a message. By creating fragmented memory area, it is impossible to send the string-array index as-is without preparing it to be sent and packing it as a message. This action is possible, but incurs computation overhead when preparing to send the string-array index, and also when receiving it. The goal is to create the data structure as one continuous block and when it is needed to be sent, simply transmit the contents of the memory block that includes all the information needed to fully reproduce the string-array index. In the remainder of this section we present the methods needed to facilitate such an implementation, and present the challenges and their algorithmic solutions.

The following description explains the implementation details for each layer of the string-array index structure, and the overheads involved (if any). It starts with the top levels, namely the counters array and the first level coarse offset vectors, and continues to drill through the structure, ending with the third level of offset vectors, and the lookup table.

Raw counters array and coarse offsets level 1 The first levels of the structure are rather easy to implement in a continuous fashion. The raw counter array itself is inherently a sequence of bits, and needs no further adjusting. The first level coarse offset vector can be places immediately after the raw vector, which requires that we record the size (in bits) of the raw vector - an overhead of $\log \log N$ bits which we can allow.

In order to fully represent an offset vector (coarse or complete), we need to know two details about it: first, we must know if it is coarse or not. We need not know the actual number of offsets within it, since this number is implied from the string-array index structure. Second, we need to know the size of the offsets, in order to allow a direct access to any offset within the offset vector. In the case of the first level coarse offset vector (C^1), we know for sure that it is coarse, and that it contains $m/\log N$ fixed-size offsets of $\log N$ bits each. Since we know N (we kept it previously, to allow access to the beginning of C^1), we can store C^1 as a continuous bit-array which contains the information of all the offsets in their binary form, where each offset inhabits $\log N$ bits. To access C_i^1, we need to access the $(i \log N)$th bit from the beginning of C^1, and read the next $\log N$ bits, which contain the actual offset.

Level 2 offset vectors Second level offset vectors (C^2) are a far more complex challenge. These offset vectors differ in their size, since they point at subsequences of varying lengths, which translated to different sizes of offsets. Furthermore, some of these offset vectors are coarse (pointing at short subsequences) and others are complete offset vectors. The required information for an offset vector C^2_j can be gathered from C^1: by sequentially reading the offsets of the jth and the $(j+1)$st group, we can calculate the size of the subsequence and decide if the offset vector is coarse or not, and what is the size of each offset. Assuming that we store all the C^2 vectors in a continuous bit array, we still need to know exactly where in this bit-array the jth offset vector begins.

This problem is in fact another instance of the variable length access problem. It is an appealing idea to solve it by using a string-array index in a recursive fashion. However, this problem can be solved with an acceptable overhead by simply holding an offset vector which points to the bit-array representing C^2. The total size of C^2, as shown above, is bounded by $N/\log N + 3m$ bits. It is divided into $m/\log N$ offset vectors, so an offset vector for C^2 will hold $m/\log N$ offsets, each of size $\log(N/\log N + 3m) \approx \log N$ bits. This accumulates to an additional size size of approximately m bits. Accessing the correct offset vector is simple: the information regarding its size and coarseness can be obtained from C^1 as described. Its starting point within C^2 is read from the offset vector, where a single lookup (and reading of $N/\log N + 3m$ bits) provide the offset. For additional space savings, the size of the C^2 bit-array can be kept (requiring approximately $\log N$ bits), and all offsets that point at C^2 will be limited in size to $|C^2|$.

Level 3 offset vectors and lookup table Level 3 of the string-array index is similar to the second level, with the additional complexity of the lookup table. The lookup table itself can be omitted when transmitting the string-array index, because it is dependant only on the parameters of the string-array index and can be generated in the receiving node. Otherwise it can be easily kept as a bit-array with simple lookup into it. The level 3 offset vectors are kept in a bit-array, where each offset vector accommodates a constant size. The size needed for such an offset-vector is $3(\log\log N)^2$ bits. However, to remain within the stated storage bounds, we cannot allocate this amount of storage to each and every subgroup. We must skip the subgroups handled by the lookup table when encoding this bit-array.

To solve this problem, we encode in the bit-array only those offset vectors

which actually are in use. In this case, when looking for the jth offset vector, we need to translate it to $r_j \leq j$, which is actually the index of the same subgroup in the collection that includes only those subgroups handled by offset vectors. We create a bit-vector F of size $m/\log\log N$ bits, with the ith bit being a flag marking whether the ith subgroup is handled by an offset vector. Given this bit-array, we can calculate r_j by using the *rank* operator, $r_j = rank(F, j)$[2]. Calculation of the *rank* operator for a bit-vector of N bits is possible using $o(N)$ bits in $O(1)$ time [Jac89, Mun96], so using additional $m/\log\log N$ bits of storage we can perform the needed translation.

Summary This section outlined the method for storing the entire string-array index in a continuous fragment of memory, while still allowing random access to any given element. To facilitate this improvement, additional $\log\log N + m + m/\log\log N$ bits of storage are needed.

4.7.2 Offset vectors division

The offset vectors of C^1 and C^2 are divided by their size to coarse offset vectors and complete offset vectors. The division is necessary for space limitations, when holding a complete offset vector requires too much space, we are forced to use a coarse offset vector. However, there may be situations in which a single subgroup is so large that it can compensate for the small size of other smaller groups, such that groups that individually would not merit a complete offset vector may be handled by one.

The advantages of this approach are clear: when using a complete offset vector instead of a coarse one, we reduce the number of internal lookups needed for a single item lookup. Also, a subgroup that is handled by a complete offset vector does not need further processing in the following levels of the string-array index and therefore is more space efficient.

The algorithm for producing this optimization is rather simple: let I be the number of items within each subgroup, and T^i be the total size in bits of subgroup i. The condition for keeping a complete offset vector is $I \log T^i < T^i$, meaning that the size of the complete offset vector is still smaller than the size of the original group (we might, of course, be using a tighter threshold, requiring that the offset vector is substantially smaller than the original size of the group). The algorithm will collect a group of subgroups G, according to a given selection criterion, and keep building complete offset vectors as long as $\sum_{i \in G} I \log T^i < \sum_{i \in G} T^i$. The selection

[2] $rank(V, j)$ returns the number of 1 bits occurring before and including the jth bit in the bit vector V.

criterion might be as simple as adding the following groups in consecutive order, or more complex, such as attempting to create an optimal packing of the groups, such that as little groups as possible are left without a complete offset vector.

This optimization is very useful in situations where the data is highly skewed. In these cases, the data is usually dominated by a small number of frequent items, with a large number of relatively rare items. This will result in a small number of subgroups whose binary encoding is rather large, and those groups can encompass within them a large number of smaller groups. As the data tends to be more uniform (with skew ≈ 0), this strategy loses some of its strength, but still provides an improvement.

Chapter 5

Applications

In this section we explore a range of applications that may take advantage of the SBF. The first category of such applications consists of extensions to methods or abilities of the regular Bloom Filter. For example the Bloomjoins method, which allows for efficient joins within a distributed database, is improved be the usage of SBF within it, transforming it to *Spectral Bloomjoins*. New queries can be answered while still maintaining efficiency and accuracy.

The second category has new applications which use the SBF to efficiently perform tasks which weren't possible with a simple Bloom filter. One example of such task is ad-hoc iceberg queries, in which one is interested in a small subset of the most frequent items within a data-set. These items can be thought of as the "tip of the iceberg", where we ignore the majority of the items in the data-set which lie beneath the surface. The SBF allows us to perform ad-hoc iceberg queries, in which the threshold determining the size of the result-set is set only at query time, improving on current methods which require a given threshold to perform preprocessing of the data.

5.1 Aggregate queries over specified items

Spectral Bloom Filters hold mostly accurate information over each and every item of the data set. Therefore it 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, $\bar{f}$. An example for such query is:

```
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 proportional to the size of the set.

5.2 Ad-hoc iceberg queries

In many situations, a data-set is tracked regularly in the lookout for items which are more frequent than a certain threshold. It is desirable to set triggers that will alert us once an item with a high count is encountered. For example, a company which tracks customer calls 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. Queries of this kind are often referred to as "iceberg queries", since they deal with a small fraction of the data, while the main body of the data-set remains hidden underneath the surface.

The example described above presents an *ad-hoc* iceberg query, in which the threshold against which items are tested upon insertion is dynamic and possibly changes between queries. Methods to handle iceberg queries, proposed in [FSGM+98, MM02] require a certain preprocessing the data given a static threshold. When the threshold changes, the methods of [FSGM+98, MM02] require rescanning of the data using the new threshold (or in the case of streaming data [MM02], 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.

Traditional methods for iceberg queries

Iceberg Queries [FSGM+98] are queries of the form

```
SELECT t1,t2, ... ,tk,count(rest) FROM R
GROUP BY t1,t2, ... ,tk HAVING count(rest) >= T
```

Usual database execution methods are not efficient for such queries, since they usually require sorting of the entire relation and then performing the reduction. If the threshold T is such that only a small part of the relation is returned by the query, then the execution plan is far from efficient. Another version of iceberg queries is dealt with in [EV02], where iceberg techniques are required to perform over streaming data. The solutions presented in [FSGM$^+$98, EV02] are both constrained in the sense that they require a prior knowledge of T to function, and they do not maintain complete knowledge of the data set. This optimization allows for very compact memory structures, but prevents the usage of the algorithms in ad-hoc situations, where the threshold might change during inspection.

Suppose, for example, that a query is executed with a threshold of 1%, which turns out to be too high and the query returns with no results. To lower the threshold and execute the query again, the data must be fully scanned again and the data structure needs to be built again. Since the data structures are very compact and assume that many items hash to every bucket, the information stored has a very high error ratio, so it can hardly be used for exact queries, trying to figure out the items which comply to the new threshold. To prevent this from happening, an initial low threshold must be selected, but this neutralizes many of the the advantages of the proposed algorithms, requiring them to use a lot of memory (or forcing a high error ratio on the results).

In this section we present two ways to utilize SBFs in ad-hoc iceberg queries: a straight forward implementation, using a regular SBF to answer the queries, and a method similar to the MULTISCAN-SHARED method of [FSGM$^+$98], performing progressive filtering of the data.

Algorithm & Error Analysis for Iceberg Queries

The SBF can be used as-is for purposes of iceberg queries answering. For streaming data, the SBF can be built while the data flows, and any item whose frequency passes the given threshold is reported. For non-streaming data hashed into an SBF, a single scan of the data is performed. Each item inserted is checked within the SBF for its frequency, if it exceeds the threshold, the item is reported. The threshold can be dynamic and determined at query time, and not while hashing the data.

Using an SBF to handle iceberg queries might generate errors due to its probabilistic nature. These errors can be eliminated by performing a scan of the potentially heavy itemset to retrieve the actual counts of each item from the range of items R, as in [FSGM$^+$98]. This is not possible under

the assumption of streaming data unless some additional data structures are built to support the extra queries. In the remainder of this section, no such scan is assumed.

The SBF may produce *false-positives*. That is, all items that should be reported are indeed reported, along with several items which do not pass the threshold. If we denote by Q the set of all items returned by our algorithm, and for an item t, we denote by f_t its true frequency in R, it is guaranteed that $\forall t \in R \ s.t. \ f_t \geq T, \ t \in Q$. However, Q might include some items for which $f_t < T$. Notice that for iceberg queries purposes, the error is only a subset of the usual Bloom Error, because the errors have to be big enough to pass the threshold.

Assume that the distribution of item frequencies in R behaves according to some function $d(f)$, such that for a frequency f', $d(f')$ represents the ratio of items in R with that frequency. For example, for uniform data $d(f)$ will be a constant. We denote by n the number of distinct items in R.

Note that since we are answering a boolean query (is the item over the threshold or not), for items with frequency greater than T, we care not whether there was a bloom error or not, since it does not affect the outcome. We consider items whose frequency is $f' < T$. There are $nd(f')$ items with frequency f'. For an item in that group to belong to the output set Q it must be stepped over by k items of frequency larger than $T - f'$. This is approximately equal to the Bloom Error generated by hashing only the items with big enough frequencies (we ignore secondary errors generated by two items mapped to the same bit and so on). We denote by $D_{f'} = n\sum_{i=T-f'}^{\infty} d(i)$ the number of items with such frequencies, so for each f', the actual error rate in this scheme is $E_{f'} \approx \left(1 - e^{-kD_{f'}/m}\right)^k$, using the same calculation given in Section 2.1. Thus, the total error rate across all items is

$$E = \sum_{f=0}^{T-1} d(f)E_f \approx \sum_{f=0}^{T-1} d(f) \left(1 - e^{-kn/m \sum_{i=T-f}^{\infty} d(i)}\right)^k$$

This function represents a tradeoff: for the same parameters, as T increases there are more items below the threshold, but there are less items big enough to make them pass the threshold. In figure 5.1 we present the error rates for Zipfian distribution with several skews and several Ts in question, in which the tradeoff is obvious. For all except uniform distribution (skew 0), the error rate increases for very small T, and then it reaches a maximum and drop as T continues to increase, the maximum moves to lower T as the

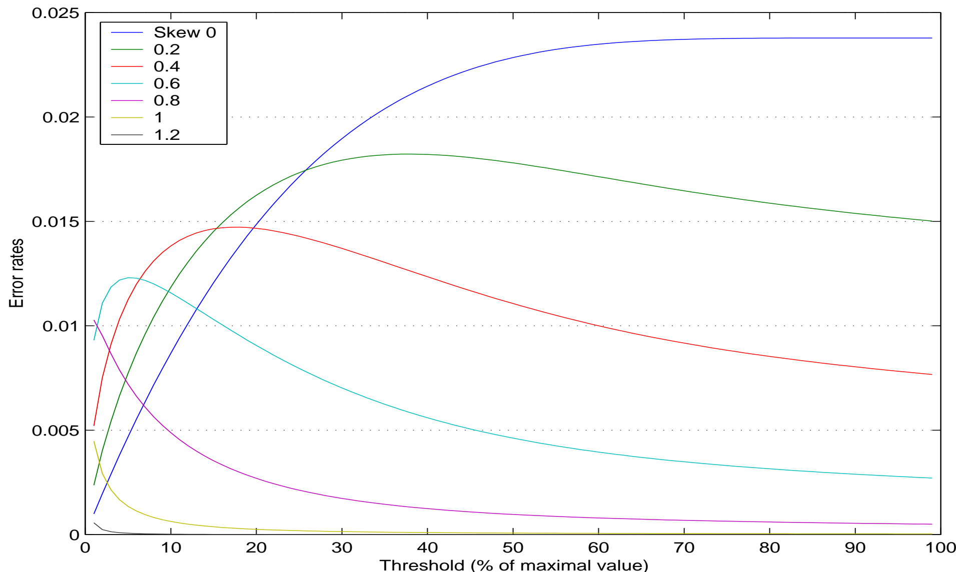


Figure 5.1: Error rates for data with Zipfian distribution of several skews with different thresholds.

skew increases. The parameters used were $k = 5, \gamma = 1$, which represent a smaller Bloom Filter than the optimal. The theoretical Bloom Error for these parameters is $E_b = 0.1$, while in the iceberg scenario, the expected error never exceeds 0.025, while at most relevant thresholds it drops below 0.01.

Multiscan SBF method

Another method of performing iceberg queries is using SBFs in a way similar to the MULTISCAN-SHARED method, as described in [FSGM+98], using several scans of the data. The idea is to perform several stages of filtering, an item passes the combined filter only if it hashes to heavy buckets in all the stages. By building this filter incrementally, we assume that the first filter will filter out a fraction of the items. Therefore, the second filter will have to deal with less items and thus can be smaller. We propose using SBFs for the various stages, and using the parameters of the SBF (namely m and k) to control the strength of the filter. In this implementation, knowledge of the threshold is required while building the SBF, and it limits the options for ad-hoc queries.

To be competitive with the methods proposed in [FSGM+98], the SBFs

need to be of very small sizes, around 1% of n. This transforms the implementation to Lossy Bloom Filter, since we assume in advance that each bucket will have many items hashing to it, with the Bloom Error reaching probability of 100%. Notice that if the first filter fails to filter out items, the next filters (which are smaller) have a very small probability to filter items out. We can determine the properties of the next filter on the fly, relying on the performance of the current filter. For example, we can calculate the average count over the buckets of the current SBF, and if it exceeds the threshold we know that the filtering will be very weak, and therefore we might want to enlarge the next filter (or reduce the number of hash queries), to allow the next filter to be more effective.

Advantages Using SBF for iceberg queries allows a degree of freedom with threshold selection and query parameters. It transforms the problem from a threshold-bound algorithm, in which the threshold must be provided while the data flows, to an ad-hoc process, in which the data is processed with no connection to the querying process. When using very small SBFs and progressive filtering, the memory requirements are competitive with those in [FSGM+98], and the SBF allows for more possibilities of using the space, and for less scans of the data.

5.3 Spectral Bloomjoins

Bloomjoins [ML86] 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 as follows: R_1 sends its Bloom filter (denoted BF_1) to R_2, R_2 is scanned and tuples with a match in BF_1 are sent back to site 1 as R_2'. At site 1, R_1 is joined with R_2' to produce final results. 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.

A Spectral Bloomjoin is an extension of the Bloomjoin scheme using SBFs. This method can be used to perform distributed aggregative queries. Consider the following query, which filter the results using a given threshold T:

```
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 capability to represent multiplicities 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 [ML86] 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. The error ratio expected is the standard Bloom error, as described in Section 2.1. Also, the frequencies reported are subject to Bloom Error and may be higher than their actual value. The size of these errors can be estimated using the calculation described in Section 2.3, or improved by using the Minimal Increase method (when no deletions are necessary). 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 distributed joins, by allowing the query to be answered after transmitting one synopsis from site to site, eliminating the need for a feedback. 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 [GGMS96], 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* [HNSS93] 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. We sketch the modifications made in the bifocal sampling, with reference to the algorithm described in [GGMS96]. 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.

When using SBF in this procedure, each error will be multiplied by n/m_2. We might also label items as dense when in fact they are sparse. For this to happen, $mult_R(v)$ needs to be smaller than n/m_2 and the error rate needs to be big enough to make $mult_R(v) \geq n/m_2$. In fact, this kind of errors might balance the first type of errors.

By following the logic of Lemma 3.3 in the original paper, we substitute $mult_R(v)$ by $mult'_R(v)$, which is the result of querying the SBF for the item v. For tuples that are dense in R, we are certain that $\mathcal{E}(\tau) = 0$. For tuples that are not dense, if $mult'_R(v) < n/m_2$ (no Bloom Error or a small one), we have $\mathbf{E}(\mathcal{E}(\tau)) = mult'_R(v) \leq \ mult_R(v) + \gamma$. From this we need to subtract the tuples which are sparse but considered dense due to Bloom Error. These tuples are rare, since they must be subject to Bloom Error and also be sparse, but with sufficiently high multiplicity so that when adding the error, they pass the threshold and become dense. It follows that $A_s \leq \mathbf{E}(\hat{A}_s) \leq A_s(1+\gamma)$, as required.

This deviation in the estimated total will usually be much smaller, and can be very small if using the MI method. However, the error can be incorporated into the calculations in the procedure and the estimation can be adjusted according to the expected average error.

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

5.5 Range queries

Range queries are queries concerning a subset of the relation in which certain attribute is within a (open or closed) range (L, U), for example the following query

```
SELECT count(a) FROM R
WHERE a>L AND a<U
```

The SBF can provide (mostly) exact information per specific item, but due to its structure has no support for range queries. When answering such a query, an exhaustive search over the entire range is not always possible, and is dependent on the size of the range, which can be very expensive when the data is sparse in that range or when the range is very large.

Range Tree Hashing In order to accomplish range queries capabilities, we hash both the specific items in the relation and new items that each represent a range. The ranges are kept in a hierarchy, each range is the union of the ranges represented by its descendants.

Theorem 11. *For an attribute in the range $R = (L, U)$ with $|R| = r$, range queries can be supported with insertion and deletion complexity of* $\log r$ *and constant lookup complexity for distinct queries. For lookup queries over a range $Q \subseteq R$, the query requires $O(\log |Q|)$ SBF queries.*

Proof. We construct a binary tree T, in which every node corresponds to a subrange within R. For each node $n \in T$, we denote its corresponding subrange with R_n, and its sons $n_1, n_2 \ldots, n_k$ correspond to non overlapping subranges of R_n such that $R_{n_1} \cup R_{n_2} \cup \ldots \cup R_{n_k} = R_n$. Each node is associated with a value $v_n \in V, V \cap R = \emptyset$. The hash functions of the SBF hash the extended range $R \cup \{v_1, v_2 \ldots, v_{|T|}\}$.

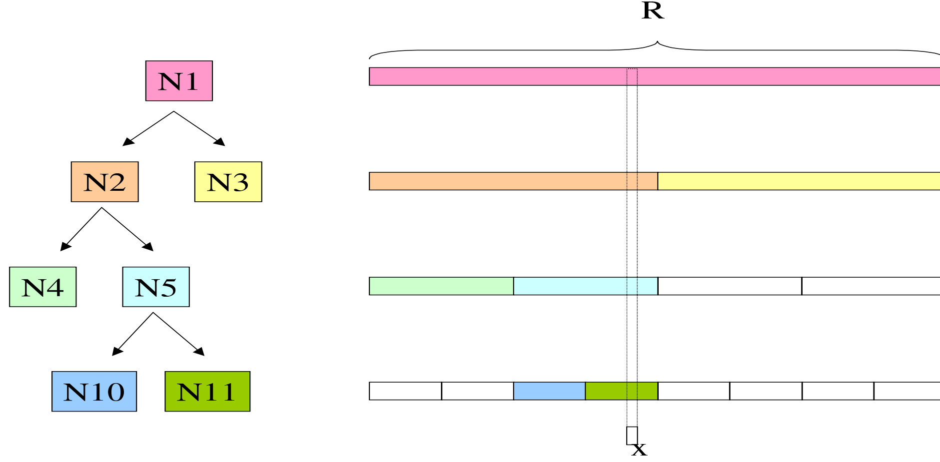


Figure 5.2: A Part of the tree created in order to handle range queries.

When inserting an item v into the SBF, every node n in T with $v \in R_n$ (these correspond to a single branch), inserts its value v_n into the SBF as well. Since the depth of a tree is $\log r$, every insert into the SBF is transformed into $\log r$ inserts. The same holds for deletions.

When performing lookup of a single value, there is no need to traverse the tree. A direct query is performed against the SBF and thus a single lookup is required, with the implied SBF complexity.

For this proof we assume the usage of a binary tree, though other trees may be used as well. For ease of reading, we denote $\log_2 n$ by $\log n$. To query the tree for the range, we perform a BFS over the tree. Once a node n contains a range fully enclosed in Q, we query the SBF for v_n, add the result to our grand total and do not continue to its descendants. If the range of the node does not intersect with Q, we do not continue to query its descendants.

We denote by l_{min} the highest level of the tree in which we performed a query in the SBF. In this level, we can perform as many as 2 queries, because if Q includes 3 adjacent nodes in it, at least two of them belong to the same parent node, and therefore the parent node is fully enclosed in Q, and l_{min} would not be the minimal level in which a query is performed.

The queries in level l_{min} remove from Q the middle part, which leaves

(in the worst case) two smaller ranges Q_l and Q_r for the next level. Both ranges begin precisely in a node boundary, and cannot spread over two or more nodes in this level (otherwise the parent node would have been covered). This means that in the worst case, both Q_l and Q_r generate one additional SBF query in this level. The remainder is directed to the next level, in which the same logic holds, until (in the worst case) we reach the final level of the tree and have to perform one final distinct query for each boundary of the range.

To sum up, each level of the tree (starting with l_{min}) requires up to 2 SBF queries. We consider the subtree T' which encloses the entire range Q. Its height is $\log |Q|$, therefore the entire process requires up to $2 \log |Q|$ queries. □

Note that when using trees with degree of p (rather than binary trees), the lookup complexity changes to $p \log_p |Q|$, and similarly insertion and deletion complexity are reduced to $\log_p r$. These observations are directly related to the depth of the tree.

Size considerations The SBF now must contain additional items corresponding to items in the range tree. In the worst case, there are $|R|$ new items (for example, a binary tree whose leaves are ranges of size 2, will contain $|R|/2$ leaves and a total of $|R|$ nodes, each associated with an item in the tree). We denote by $S \subseteq R$ the subset of values appearing in the relation, $|S| = n$, the number of distinct items inserted into the SBF. We denote by V_a the set of values in the range tree that actually are inserted into the SBF during the hashing of S. We can state the following claim:

Claim 12. $|V_a| \leq n \log r$

Proof. When inserting any item x for the first time, we insert into the SBF all tree items that lie within the corresponding tree branch that ends with x. The length of a full tree branch is $\log r$, therefore for n different items we need at most $n \log r$ tree items. □

By this claim, we require an expanded SBF in order to support the larger domain. This increases the memory demands of the SBF to $O(N \log N)$ bits. However, this data structure supports a very wide range of queries, both range queries and accurate specific queries in the same data structure. Note also that the structure of the range tree was predefined, while a more elaborate building of this tree can provide much better results using a smaller tree.

Discussion Usually, range queries are handled using histograms, which are significantly more space-economical than the SBF. However, histograms can not guarantee a certain precision for a single query, since extrapolation is needed for ranges which cover parts of buckets, in which the distribution of data is normally not known. The SBF guarantees one-sided errors, which is an important property when using the results for decision making. Also, it gives a certain error guarantee per query, something that histograms cannot produce.

To summarize this section, it may be desirable to use SBF where precision of each and every single query and the predictability of errors (both in nature, namely false-positive errors, and size) are the main issues. When memory is the main constraint, the usage of SBF is not the recommended decision.

Chapter 6

Experiments

We have tested the accuracy of the various SBF lookup algorithms described in Sections 2 and 3, as well as the space efficiency of the encoding methods described in Section 4.5. Another set of tests examined the string-array index structure, testing both its storage requirements and its performance for lookups, updates and initialization.

6.1 Algorithms comparisons

We have tested and compared the three lookup schemes from Sections 2 and 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.

In the first two sets of tests, reported in Figures 6.1 and 6.3, 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 larger) 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 6.3a,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 result with $\gamma = nk/m$ ranging from about 0.12 to about 2. The results are shown in Figure 6.1a,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 6.1c. 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.

The above experiments show clearly the significant precision and stability of the Minimal Increase method, and also the substantial improvement that the Recurring Minimum method shows over the Minimum Selection.

In the third set of tests we used real data: the Forest Cover Type database, obtained from the UCI KDD Archive [Arc]. We used the *elevation* measure as the property indexed by the SBF. The database has a total of 581012 records, with 1978 distinct values for the *elevation* measure, distributed as shown in Figure 6.2a. We have tested the performance of the three methods over this database while changing the value of γ, by changing

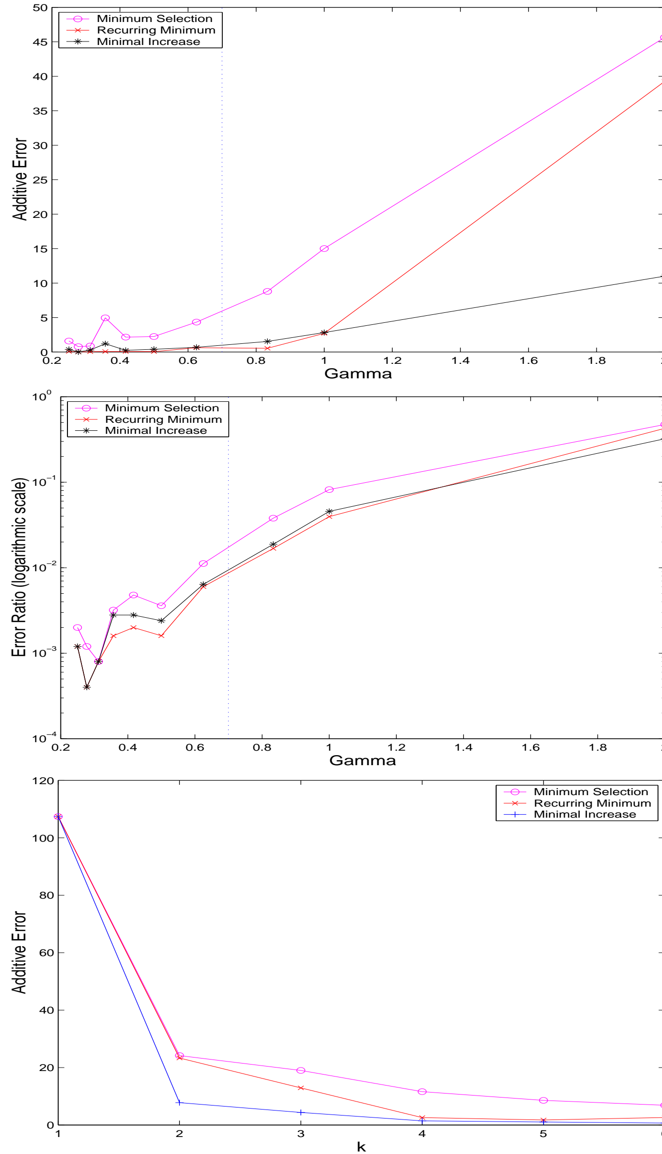


Figure 6.1: Accuracy of MS, MI and RM algorithms for various values of γ, with $k = 5$, with additive error (a), and log of error ratio (b), dotted line represent optimal γ. Additive errors in the three algorithms for various k values, with $\gamma = 0.7$ (c). In all experiments, MI and RM are better than MS, with some advantage to MI.

the size of the SBF. The results, shown in Figure 6.2b and c, are consistent with the results over synthetic data-sets and display an advantage to the Minimal Increase and Recurring Minimum methods over the basic Minimum Selection heuristic. The Minimal Increase and Recurring Minimum methods behave similarly throughout this test, with a slight advantage to the Minimal Increase method.

6.2 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 6.3, 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 6.4, 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.

6.3 Encoding methods

We tested the storage needed by the encoding methods described in Section 4.5, comparing the Elias method, and several configurations of "steps" for data with varying average frequency of items. The results, shown at Figure 6.5, 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 for small counters. However, the Elias encoding improves as the average frequency increases, and beats the performance of the steps methods.

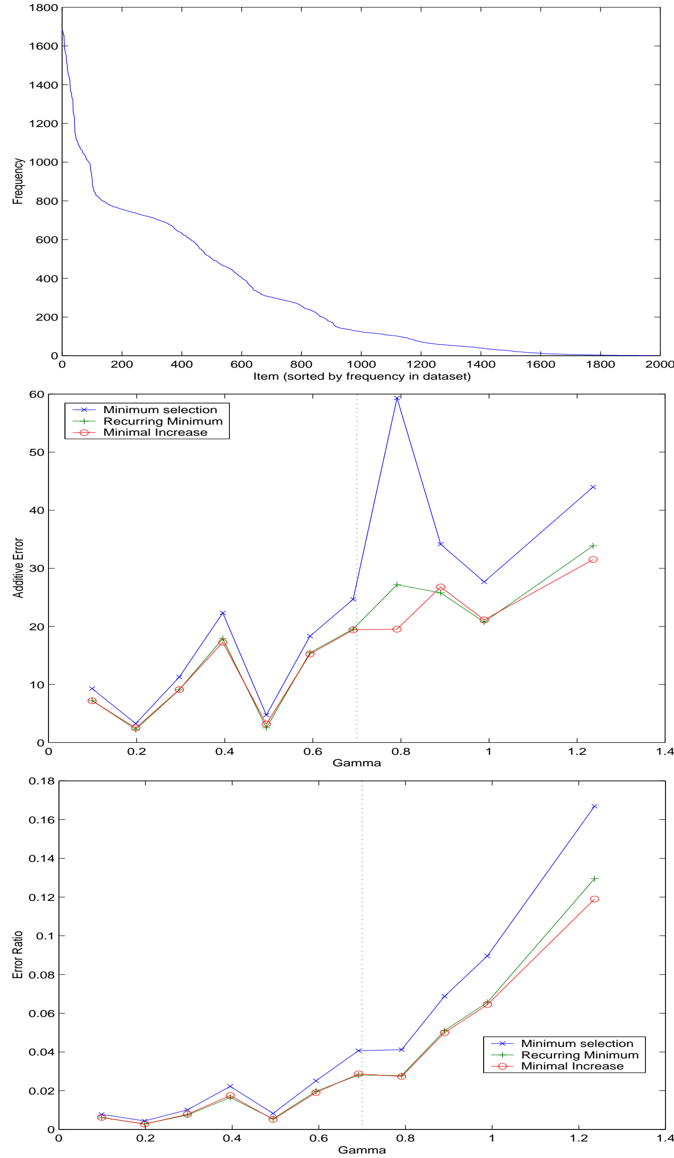


Figure 6.2: Results of tests using the *elevation* property of the Forest Cover Type database. Graphs display the distribution of the *elevation* property (a), the additive error (b) and error ratio (c) of the MS, MI and RM algorithms for various values of γ (dotted line represents optimal γ), with $k = 5$. In all experiments, MI and RM are better than MS, with some advantage to MI.

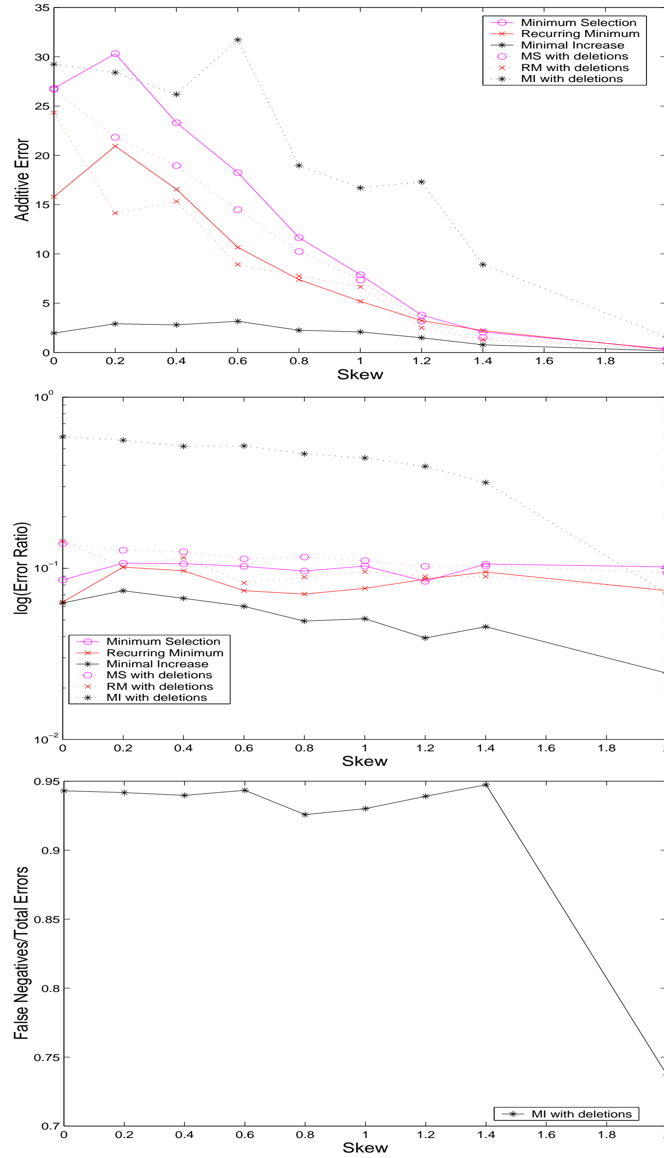


Figure 6.3: Performance of MS, RM and MI algorithms for Zipfian distribution with varying skew (θ), with deletions (dotted lines) and without deletions (full lines). Both additive error (top) and log of error ratio (center) are shown; in all experiments $\gamma = 0.7, k = 5$. The third graph shows the ratio of False Negative errors in the MI algorithm out of the total errors (there are no false negatives in MS and RM).

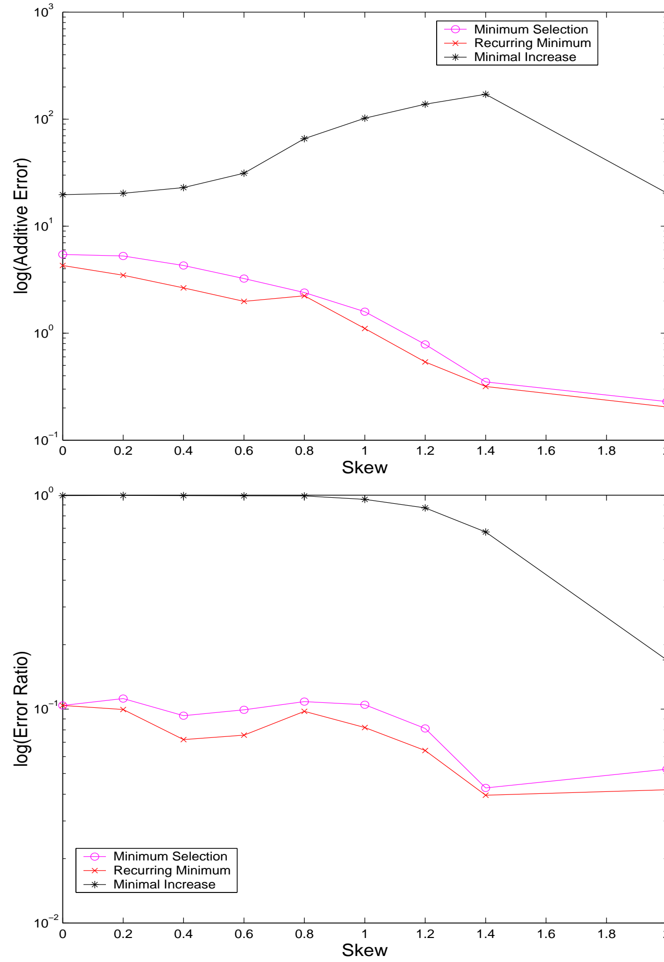


Figure 6.4: Accuracy of MS, RM and MI algorithms for Zipfian distribution of varying skew (θ), in a sliding window scenario. Both log of additive error and log of error ratio are shown, in all experiments $\gamma = 0.7, k = 5$.

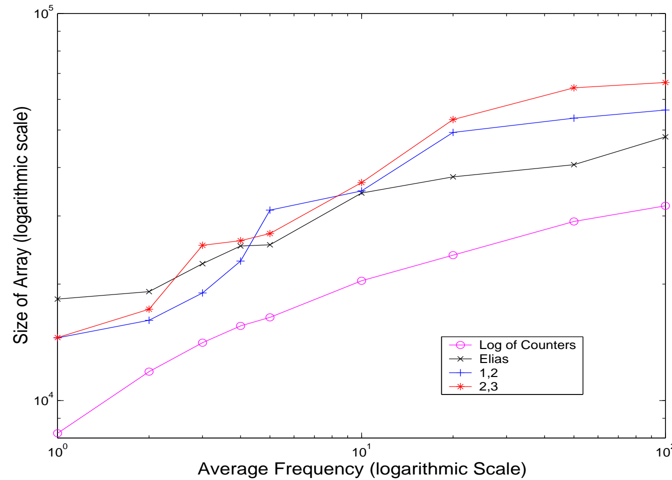


Figure 6.5: Comparison of various encoding methods. Several "steps" configurations were tested along with Elias encoding. The results are compared to the optimal Log of the counters.

6.4 String-array index performance

The string-array index, as described in Section 4.3, needs to be efficient both in its storage requirements and in the complexity of performing the basic actions needed for the SBF. These actions are the initial building of the string-array index, increasing a counter, and performing a lookup for a given item. The string-array index was fully implemented in C++, except for the continuous memory improvement discussed in Section 4.7. We performed several experiments that check the various aspects of its usage. Most of these experiments were conducted regardless of our specific usage of the string-array index as a supporting structure for the SBF, but as a stand-alone module.

Performance The performance of the string-array index was tested by populating the structure with a varying number of items stored in it. For each array size n, we have performed three actions: (i) the structure was initialized with all items being 0, (ii) we performed $10n$ random insertions of items, such that the average frequency at the end of the stage was 10. (iii) Finally we performed lookups for each and every item, totalling at n lookups. We measured the time each of these stages required, dividing the time of stage (ii) by 10, to find the time n insertions needed, in order to create a

comparable amount of time. Those tests were executed on a Pentium III (500MHz) machine, with 512MB of RAM.

The results we show are the total times measured, and also the time per action, being simply the total time divided by n. The time measured for insertions include the time required for rebuilding the array, when slacks are exhausted. For each array size we have performed 5 runs of the test, and the results shown are the average over those runs. Figure 6.6 shows those two measurements over array sizes ranging from 1000 to 1 million items. The first set of results show that, as expected, the complexities of those actions are linear with n. These results are in accord with the analytical results given in Theorem 6. This is also demonstrated in the second chart, where it is clear that the time per-action is indeed constant for those actions, even though the time required for insertions has a large variance. This last observation can be explained by the highly random nature of the insertions, also note that the average time actually decreases when n increases.

Finally, we compared the performance of the string-array index to a hash table. In order to perform this test, we used the hash table implementation found in LEDA [LED], which uses chaining for collision resolving. This test compared the full SBF implementation to the hash table, with the SBF using $k = 5$, and having m equal to the number of buckets allocated in the hash table, and the straight-forward method for lookup and increase. We also plugged in the same hash functions used in the SBF to the hash table, to create maximum match between the two schemes. We executed the same performance check described above for this setup, comparing the performance of the two methods.

The results of this test are shown in Figure 6.7. It is important to remember that every lookup in the SBF translates to k lookups, and the same is true for updates, giving the hash table an inherent advantage in this comparison. However, as items are hashed into the table and collisions accumulate, the complexity of the actions performed by the hash table increase. The SBF suffers no such penalties and perform the same number of lookups and updates with no regards to the hit-ratio within its counters vector. Note that the results for lookups are measured after the insertions, where collisions might affect them.

It is evident from the results that the SBF is, as expected, somewhat slower than the hash table. However, for larger table sizes, the hash table is only about twice as fast as the SBF, where we would expect a ratio closer to k. In fact, there is a degradation in the performance of the hash table as the size increases, which can only be explained by the fact that the hash functions are not perfectly random, and have some effect of clustering of

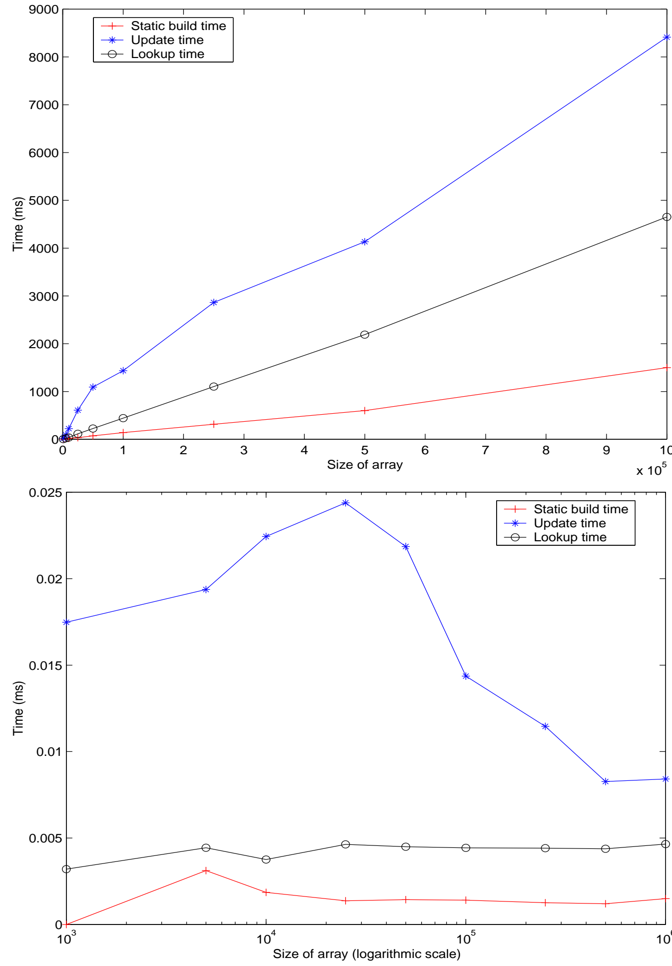


Figure 6.6: String-array index performance over various array sizes. The first graph presents total times, while the second shows time per action.

results. When the size of the table increases, this results in several buckets hashing a large number of items, thus affecting the average complexity of a lookup into the table. This effect exists in the SBF as well, but it does not manifest itself in the performance (since collisions do not cause additional actions), but might result in higher error ratios.

Storage The second important aspect that was tested is the storage required by the string-array index. We used the same setting described above, and checked the size (in bits) required for every part of the structure. The results shown in Figure 6.8 compare the size of the string-array index to the raw size of the bit vector that holds the counters. This comparison is performed for the empty array (average frequency = 0) and for the array after the insertions (average frequency = 10). The size of the bit array includes the slacks, with slack ratio of 0.5, meaning that 0.5 bits are added to the size of the bit array per item. The sizes of the bit array before and after the insertions are almost identical, thanks to the usage of the slacks, which (mostly) prevent the need to reallocate the array and increase its size.

The comparison shows that for a bit vector size of N, the string-array index requires about $1.5N$ bits in the initial state, and about $2N$ bits in the final state. This difference is explained in the graphs shown in Figure 6.9, which divide the total storage into its various components. A comparison between the two graphs clearly shows that for the empty array there is almost no need for 3rd level offset vectors, since all subgroups are small enough to use the lookup table. However, in the filled array, there is a considerable number of groups that are too large to be handled by the lookup table, requiring that offset vectors be built for them. This is the major difference between the results in the two scenarios, and explain the rise in the size of the string-array index. This size increase is unique to the initial stage, though, and does not continue further when more insertions are introduced into the string-array index, so the storage stabilizes at about $2 - 2.5N$ bits.

Next we compare the storage needed for the string-array index with the storage a regular hash table would require. Both structures require storing of the counter values themselves, with each structure relying on additional storage: the string-array index needs the entire offset storage, while the hash table needs to store the keys themselves, in order to resolve collisions in lookups. The storage needed by the hash table for m distinct keys can be described as $m \log m$, assuming that the keys are integers of the domain $[1..m]$, or for a tighter estimate, the total size is $\sum_{i=1}^{m} \log i$. We compare

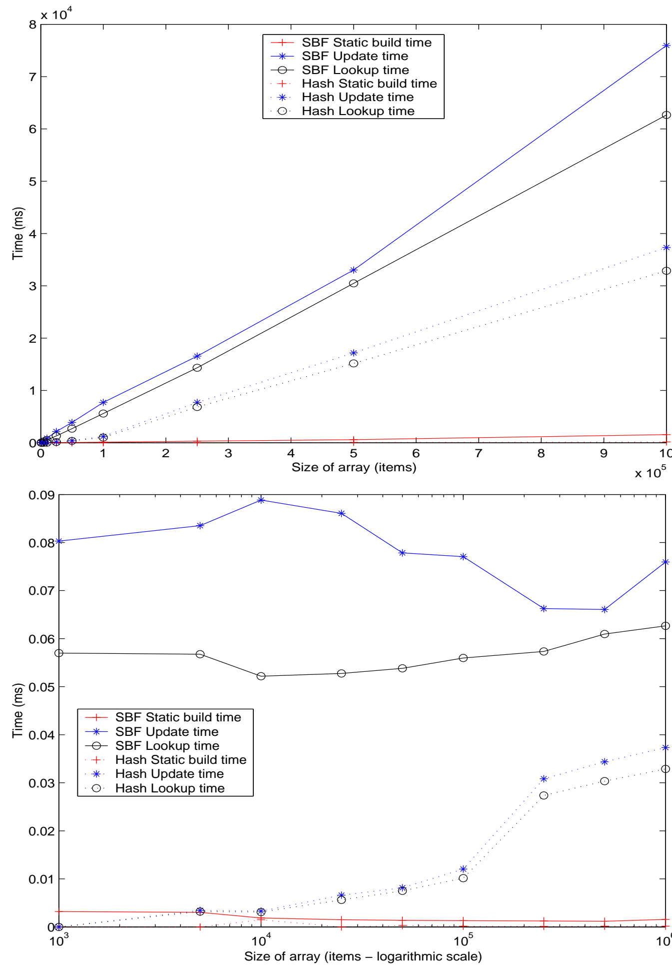


Figure 6.7: Performance of the SBF compared to the LEDA implementation of hash table, for various table sizes. The SBF uses $k = 5$, m equal to the size of the hash table. First graph displays total results, second graph shows results per-action.

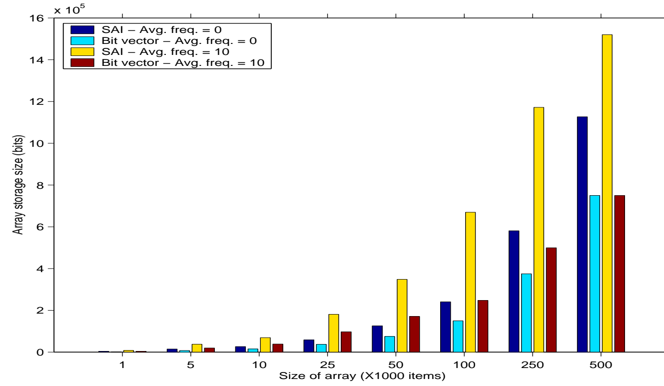


Figure 6.8: Size comparison between the bit vector (raw storage) and string-array index (index structure), for various array sizes. Results show two scenarios - an empty array (average frequency = 0), and average frequency = 10.

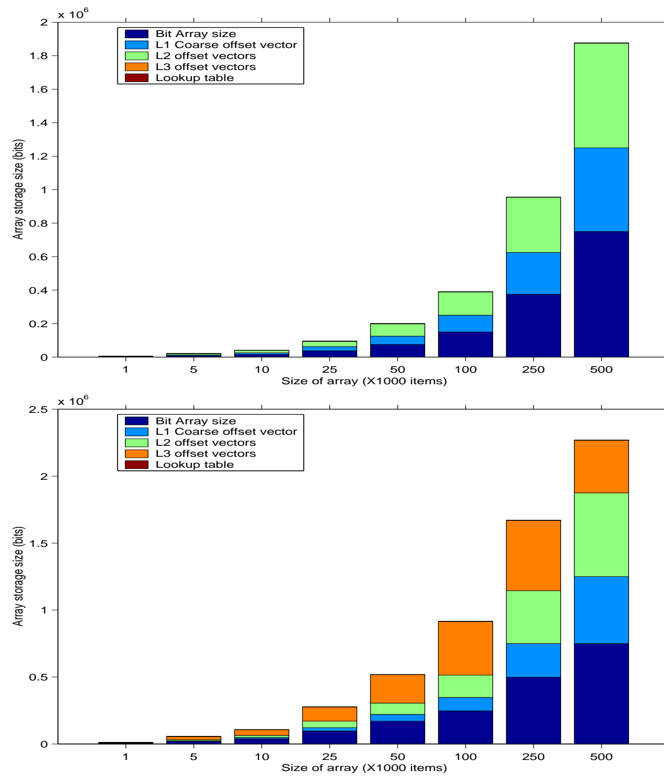


Figure 6.9: Breakup of the total size required to the different parts of the string-array index structure. First graph shows average frequency of 0, the second shows average frequency of 10.

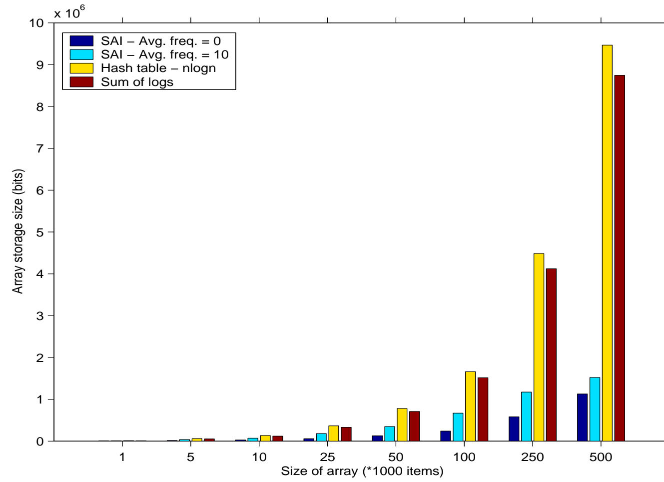


Figure 6.10: Storage size comparison between the string-array index and a conventional hash table. Sizes compared are the additional storage besides the counters. Hash table sizes are given as $m \log_2 m$ (loose estimate) and $\sum_{i=1}^{m} \log_2 i$ (tighter estimate).

those numbers to the additional storage required by the string-array index. Those results are shown at Figure 6.10, and display a clear advantage to the string-array index.

Another option for hash tables usage involves using perfect hashing. This scheme prevents the need for storage of the keys themselves, because no collisions are possible. However, perfect hashing has several disadvantages in our case: those functions do not perform well in the dynamic case. Usually, to create a perfect hashing function, the set of keys has to be known in advance. Another limitation is the size and complexity of the function itself (and its generation). A perfect hashing function requires a considerable space - $O(m \log m)$, making it as expensive as a standard hash table, storage-wise.

Chapter 7

Conclusions

This thesis 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 changes in the data-set.

Several experiments show the error rates the Spectral Bloom Filter provides for several configurations. The SBF was tested using synthetic data with Zipfian distribution, and using real-life data. The error rates of using the Recurring Minimum or Minimal Increase heuristics proved to be significantly better than those of the Minimum Selection algorithm. We also compared these methods when facing deletions and updates, in which case the Minimal Increase method reveals its main weakness and becomes the least successful of the three. We have experimented with the string-array index structure, testing its storage requirements and performance. The structure proved to be efficient in its storage needs, while performing fast lookup queries and updates.

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, 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 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 can be extended using SBF, with better efficiency for many types of queries.

Bibliography

[AMS99] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):148–173, 1999.

[Arc] The UCI KDD Archive. http://kdd.ics.uci.edu.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM Press, 2002.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM02] Andrei Broder and Michael Mitzenmacher. Network applications of Bloom Filters: A survey. In *Proceedings of Allerton*, 2002.

[Bro02] Andrei Z. Broder. Personal communication. 2002.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–202, 1975.

[EV02] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of ACM SIGCOMM*, pages 323–336. ACM Press, 2002.

[FCA] Li Fan, Pei Cao, and Jussara Almeida. A prototype implementation of summary-cache enhanced icp in squid 1.1.14. www.cs.wisc.edu/˜ cao/sc-icp.html.

[FCAB98] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *ACM SIGCOMM Computer Communication Review*, 28(4):254–265, 1998.

[FSGM+98] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *Proc. of 24th International Conference on Very Large Data Bases, VLDB*, pages 299–310, 24–27 1998.

[GGMS96] Sumit Ganguly, Phillip B. Gibbons, Yossi Matias, and Avi Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *Proceedings of the 15th ACM SIGMOD international conference on Management of data*, pages 271–281. ACM Press, 1996.

[GM98] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA*, pages 331–342. ACM Press, 1998.

[GM99] Phillip B. Gibbons and Yossi Matias. Synopsis data structures for massive data sets. *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on External Memory Algorithms and Visualization*, A, 1999.

[Gre82] Lee L. Gremillion. Designing a bloom filter for differential file access. *Communications of the ACM*, 25(9):600–604, 1982.

[HNSS93] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Fixed-precision estimation of join selectivity. In *Proceedings of the 12th ACM Symp. on Principles of Database Systems*, pages 190–201, 1993.

[Jac89] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 549–554, 1989.

[LED] LEDA - Library of Efficient Data Types and Algorithms. http://www.algorithmic-solutions.com/enleda.htm.

[Mat] Yossi Matias. Bloom histograms. Unpublished manuscript, July 2001.

[Mit01] Michael Mitzenmacher. Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 144–150. ACM Press, 2001.

[ML86] Lothar F. Mackert and Guy 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. Morgan Kaufmann, 1986.

[MM02] 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*, pages 346–357, 2002.

[Mun96] J. Ian Munro. Tables. In *Proceedings of the 16th Foundations of Software Technology and Theoretical Computer Science (FST & TCS)*, volume 1180 of *Lecture notes in Computer sci.*, pages 37–42. Springer-Verlag, Berlin, 1996.

[MW94] Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4):191–197, 1994.

[RK02] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[RRR00] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. *Lecture Notes in Computer Science*, 2125:426–437, 2000.

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

[Zip49] G.K. Zipf. Human behaviour and the principle of least effort. Addison–Wesley press, 1949.