Dynamic Count Filters

J. Aguilar-Saborit[*] P. Trancoso[+] V. Muntes-Mulero[*]
J.L. Larriba-Pey[*]
[*]DAMA-UPC, Computer Architecture Department
Universitat Politecnica de Catalunya
[+] Department of Computer Science
University of Cyprus
e-mail: {jaguilar,vmuntes,larri}@ac.upc.edu, pedro@cs.ucy.ac.cy

ABSTRACT

Bloom filters are not able to handle deletes and inserts on multisets over time. This is important in many situations when streamed data evolve rapidly and change patterns frequently. Counting Bloom Filters (CBF) have been proposed to overcome this limitation and allow for the dynamic evolution of Bloom filters. The only dynamic approach to a compact and efficient representation of CBF are the Spectral Bloom Filters (SBF).

In this paper we propose the Dynamic Count Filters (DCF) as a new dynamic and space-time efficient representation of CBF. Although DCF does not make a compact use of memory, it shows to be faster and more space efficient than any previous proposal. Results show that the proposed data structure is more efficient independently of the incoming data characteristics.

1. INTRODUCTION

Streamed data processing is the source for many interesting problems in areas ranging from financial analysis to telecom processing. In such cases, one of the basic problems is to recognize whether a new data item belongs to a set or to know its number of occurrences. The problem becomes challenging when there are large quantities of data to be processed per unit of time that evolve and change rapidly. In those situations, the problem calls for methods that are fast and adaptive.

Counting Bloom filters (CBF) [6] have been designed with some of the previous objectives in mind. CBF are similar to Bloom filters [1] but substitute every presence bit by a fixed size counter. This is a data structure with a fast access time but it suffers from an important problem which is related to the fact that its counters are not flexible. Consequently, the counters may become saturated resulting in inaccuracy in the stored information. As an alternative, *Spectral Bloom Filters* (SBF) [4] have been designed to overcome the lack of adaptiveness. SBF are composed of variable sized counters that adapt to rapidly changing data sets. However, SBF require the use of indexing structures to support this degree of adaptiveness, which make the access to each counter more complex and costly compared to CBF.

Counting Bloom Filters have been investigated for their use in network environments to summarize the content of peer-to-peer systems [3], to reduce file name lookups in large scale distributed systems [8], and in Internet Protocol routing lookups [5]. In the database environment CBF may be used to answer queries regarding the multiplicities of individual items, for example, in aggregate queries or ad-hoc iceberg queries [7, 9].

Besides providing a new data structure to represent counting filters, the Spectral Bloom Filters also propose new methods for reducing the probability and magnitude of lookup errors [4]. Bloom histograms, a further compressed view of SBF, are used to keep counting statistics for paths in XML Data [12].

In this paper we propose a new data structure to represent CBF, that we name *Dynamic Count Filters* (DCF). The DCF structure is designed for speed and adaptiveness in a very simple way. It captures the best of SBF and CBF. As a by-product of its simplicity, DCF does not require the use of indices. This fact reduces the amount of memory requirements in most of the cases.

In Table 1 we present a brief qualitative comparison between the three approaches: CBF, SBF, and DCF.

	Counters size	Access Time	#Rebuilds	Saturated counters
CBF	Static	fast	n/a	Yes
SBF	Dynamic	slow	high	Eventually
DCF	Dynamic	fast	low	No

Table 1: Qualitative Comparison of CBF, SBF, and DCF.

It is relevant to notice that DCF borrows the qualities of the two other techniques, the dynamic counters from SBF and the fast access from CBF. Also, DCF's dynamic counters avoid saturation. Finally, although the cost of a single rebuild of our data structure is high and only slightly better than that to rebuild SBF's, the number of rebuilds for DCF is orders of magnitude smaller, leading to a much smaller overall execution time.

The results from the execution of different real-life data operation scenarios, using both DCF and SBF to represent the data, show that DCF's overall memory size less than the half compared to SBF's, and its overall execution time is less than half the execution time of SBF. In addition, the flexibility of the DCF structure sustains its accuracy even in the presence of unpredicted peaks in the data set size.

The contributions of this paper are as follows:

- The proposal of *Dynamic Count Filters* (DCF) with a detailed description of the basic data structure and operations.
- An efficient mechanism to dynamically resize the DCF structure and avoid useless rebuilds.

- A quantitative evaluation of DCF as well as its comparison with SBF.

This paper is organized as follows: In Section 2 we explain the related work. Section 3 describes the Dynamic Count Filters. In Section 4 we present the setup and discuss the experimental results for the different scenarios. Finally, in Section 5 we present the conclusions.

2. RELATED WORK

A Bloom Filter, proposed by Burton Bloom in 1970 [1], is basically a bit-vector of m bits that represents a set of n data elements, $S = s_1, ...s_n$. The Bloom Filter uses k hash functions [10, 11], $h_1, h_2, ..., h_k$, that map each data element into the Bloom Filter. Each hash function returns a value ranging from 1 to m, thus for each data element, $s \in S$, positions $h_1(s), h_2(s), ..., h_k(s)$ are set to 1. Different data elements from S may map to the same position in the filter, hence, a given data element is in S, $s \in S$ with a given probability of error [1], only if $h_i(s) = 1$ for $1 \leq i \leq k$.

Bloom filters do not address the issue of deletions over multisets. In order to overcome this limitation, Fan *et al.* [6] proposed the *Counting Bloom Filters* (CBF), where a Bloom filter is extended to have the capability of counting the number of different data elements that map to the same location.

2.1 Counting Bloom Filter (CBF)

A CBF represents a total of M data elements, including repeated values. This is done by replacing bit entries of a Bloom Filter by counters ($C_1, C_2, ..., C_m$ counters). Similarly to the original Bloom Filter, each time a new data element s is to be inserted into the set, k hash functions are used to update k entries of the filter. While in the Bloom Filter the entries would be simply set to one, in the CBF the counter in each of the k entries is incremented by one. In an analogous way, whenever a data element s is to be removed from the set, the counters in the same k filter entries are decremented by one. At any point in time the sum of the contents of all counters is equivalent to $\sum_{j=1..m} C_j = k \times M$.

The usual data structure representing the CBF consists of a static data set representation where counters have a fixed size over time. Such a representation has two major drawbacks: (1) whenever an insertion of a new element results in a counter overflow, delete operations to data elements that map to that same filter entry can no longer be reflected; (2) CBF's representation is not optimal as all counters have the same bit length, thus resulting in memory waste.

Dharmapurikar *et al.* [5] addresses the problem of overflowed counters. In their proposed approach, the CBF structure is rebuilt with a larger size once the number of overflowed counters passes a certain threshold. As such, the refresh of the structure is very costly, as far as all messages must be re-inserted again. Moreover, overflowed counters may be useless during a large period of time.

2.2 Spectral Bloom Filter (SBF)

Cohen and Matias [4] proposed the Spectral Bloom Filter (SBF), which is a compact representation of the CBF. The main goal of SBF is to achieve an optimal counter space allocation. It consists of a compact base array of a sequence of $C_1, C_2, ..., C_m$ counters, which represents a set of M data elements using k hash functions as with CBF. At any point in time, the goal of SBF is to keep the size of the base array as close to N bits as possible, where $N = \sum_{j=1..m} \lceil logC_j \rceil$. Note that throughout this paper we assume log to be $\log_2$.

To achieve its goal, each counter C_j in the SBF structure, dynamically varies its size such that it has the minimum necessary bits needed to count the number of items hashed into position j. To allow for this flexibility, the counter space in SBF includes $\varepsilon \times m$ *slack bits* that are placed among the counters. A slack bit is added between every $\lfloor \frac{1}{\varepsilon} \rfloor$ counters, where $0 < \varepsilon \leq 1$.

While the counter space in SBF is kept close to the optimal value, in order to support the flexibility of having counters with different sizes, SBF requires complex index structures. In the context of our paper, we identify the index structures described in [4] as:

- *Coarse Vector* (CV): a bit-vector index that provides offset information for the beginning of a subgroup of counters. Offsets are provided using counters of a fixed-size length in bits.
- *Offset Vector* (OV): a bit-vector which provides straightforward representation of the offsets provided by the CV.

Figure 1 shows the data structures used by SBF. The first-level Coarse Vector (CV1) contains $\frac{m}{\log N}$ offsets of $\log N$ bits each, thus, each offset represents a subgroup of counters (SC). As explained in detail in [4], for the subgroup of counters that fulfills $\sum_{C_j \in SC} \log\lceil C_j \rceil < \log^3 N$, a second-level coarse vector (CV2) is required, providing a more detailed information about the offsets for each counter $C_j \in$ SC. In the case that a subgroup of counters $\sum_{i \in SC} \log\lceil C_j \rceil \geq \log^3 N$, then, as indicated in [4], an Offset Vector (OV) is used, that contains the exact offset for each counter. For simplicity, without loss of generality, the analytical models presented in this paper assume the former case, where the CV2 is required. CV2 divides SC in chunks of $\log\log N$ counters (SC$'$), and holds a total of $\frac{\log N}{\log\log N}$ offsets. Since offsets in CV2 are at most $\log^3 N$, each offset can be represented with $3\log\log N$ bits, totaling $3\log N$ per each SC$'$. Finally, the information needed to locate the exact position of the j^{th} counter is given by one offset vector (OV), one per each subgroup SC$'$. The OV consists of $\log\log N$ offsets of $3\log\log N$ bits each offset, totaling $3(\log\log N)^2$ bits per subgroup SC$'$.

The offset vector OV, can also be substituted by a lookup table depending on a threshold based on the length of SC$'$. More details about this approach can be found in [4].

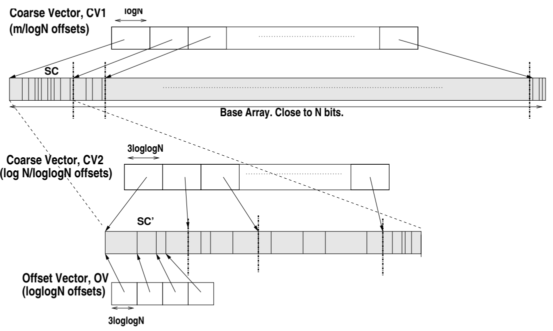

Figure 1: SBF data structures.

3. DYNAMIC COUNT FILTERS (DCF)

Dynamic Count Filters (DCF) are composed of two different vectors. The first vector is a basic CBF with each entry being a counter of fixed size $x = log\frac{M}{n}$, where M is the total number of data elements in the set and n is the number of distinct values in the set. If we consider that the filter has m counters (C_j for $j = 1..m$), the CBF vector, hereby named CBFV, accounts for a total of $m \times x$ bits. The second vector is the *Overflow Counter Vector* (OFV), which also has the same number of entries, each one including a counter (OF_j for $j = 1..m$) that keeps track of the number of times that the corresponding entry in the CBFV suffered an overflow. The size of each counter in the OFV changes dynamically depending on the distribution of the data elements in the data set. At a certain point in time, the size of each counter is equal to the number of bits required to represent the largest value stored in OFV ($y = \lfloor \log(\max(OF_j)) \rfloor + 1$). As such, the size of the OFV accounts for a total of $m \times y$ bits.

Figure 2 represents the data structure for the DCF approach. From this Figure it is possible to observe that the DCF data structure is composed of m entries with m counters split in pairs of counters, $\langle C_1, OF_1 \rangle$,...,$\langle C_m, OF_m \rangle$. All counters in the DCF have equal size of $x + y$ bits, where y varies dynamically its bit length.

The decision of having a fixed size for each counter implies that, on the one hand many bits in the DCF structure will not be used and, on the other hand, the access to both vectors is direct, hence, fast. Therefore, DCF trades counter memory space for a fast access. Overall, DCF's fixed-sized counters result in time and space benefits as it allows for a fast read/write mechanism that has an asymptotic cost of $O(1)$, avoiding the use of complex indexing structures, and saving memory space in most of the cases.

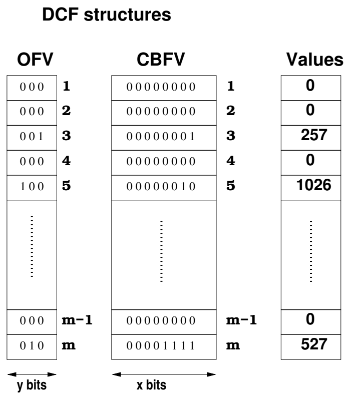

Figure 2: DCF data structure.

3.1 Querying a Data Element

Querying the filter for a certain data element s results, as with the rest of Bloom-like filters, in checking k filter entries, using k different hash functions, $h_1(s), h_2(s), ..., h_k(s)$. Checking a filter entry is performed by accessing the corresponding entries of the CBFV and OFV vectors. As counters in both vectors have the same number of bits, locating an entry is immediate and performed with simple shift and modulo operations. The counter bits are then extracted from the vectors using fast bitwise masking operations (AND and OR). Both accesses are fast with an asymptotic cost of $O(1)$. Hence, for a certain entry j, once we get C_j and OF_j, the compound value V_j for the counter stored in position j of the DCF is calculated as $V_j = (2^x \times OF_j + C_j)$.

This way, when querying a data element s, we end up having k values $V_{h_i(s):i=1..k}$ with a cost of $k \times O(1) \simeq O(1)$. The values associated to s are used depending on the application. For instance if we want to determine the presence of the data element, it is necessary to perform the simple operation of checking if one of the $V_{h_i(s):i=1..k}$ values is zero. If so, then s is not in the data set, otherwise s is in the data set with a probability of false positive as explained in [1].

3.2 Updating the Data Set

Each time we insert or delete a data element s, we must update k filter entries ($h_1(s), h_2(s), ..., h_k(s)$) in the CBFV and OFV. Updates in the CBFV are fast and performed on each counter $C_{h_i(s):i=1..k}$ using simple increment and decrement operations, depending on the operation, insert or delete, respectively. Updates in the OFV are more infrequent but may be more expensive. In addition to updating the $OF_{h_i(s):i=1..k}$ counters when an overflow or underflow occurs in the corresponding $C_{h_i(s)}$ counter, the OFV may need to be resized. In the next sections we explain in more detail the insert and delete operations for a certain data element s.

Inserting a Data Element.

As mentioned before, when a data element s has to be inserted, k counters $C_{h_i(s):i=1..k}$ in the CBFV are incremented by one. In case a counter C_j, for any of those incremented entries $j = h_i(s) : i = 1..k$, overflows, *i.e.* its value increases from $2^x - 1$ to 2^x, the value of C_j is set to zero and the corresponding counter in OFV, OF_j has to be incremented by one. Thus, the data insertion requires at most two read and write operations, which have an asymptotic cost of $O(1)$.

In the case that the overflow counter OF_j is to be incremented from $2^y - 1$ to 2^y, then, before the operation can be performed, one bit must be added to all counters in the OFV in order to avoid counter saturation. We name the action of changing the size of the OFV *Rebuild*. Rebuild operations are expensive as they require the allocation of a new vector, the copy of the contents from the old OFV to the new extended OFV vector, and finally the deallocation of the old OFV vector. Therefore, as the vectors have m entries, the rebuild operation has an asymptotic cost of $O(m)$. Notice that although the rebuild operation is costly, the motivation for having a OFV structure separately from the CBFV is that rebuilding the OFV is cheaper than rebuilding the CBFV. This is because the OFV is smaller than the CBFV and also because it is probable that many more entries contain a zero in the OFV, which will not be the case for CBFV, thus. Note that containing a zero implies no need to copy the counter. Overall, although the asymptotic cost would be the same, memory allocations and data copying for the new OFV vector are cheaper than if we re-create the whole DCF structure.

Delete a Data Element.

Upon deletion of a data element s from the data set, k counters $C_{h_i(s):i=1..k}$ in the CBFV are decremented by one. Whenever one of the counters suffers an underflow, *i.e.*, C_j, for any of those decremented entries $j = h_i(s) : i = 1..k$, is to be decremented but its original value is zero, then its value is set to $2^x - 1$ and its corresponding counter in the OFV, OF_j is decremented by one. Therefore, as in the insert case, at most only two read and write operations are needed, resulting in an asymptotic cost of $O(1)$. Notice that when we decrement a counter, either C_j or OF_j must be greater than 0, as we do not delete data elements not belonging to the data set.

Similarly to the insert-triggered OFV rebuild, delete operations may also result in OFV rebuilds, in this case in order to save counter memory space that is not needed any longer. Whenever a counter OF_j is decremented from 2^{y-1} to $2^{y-1}-1$ we may check all the other OFV counters and if their values are all smaller than 2^{y-1} then we can reduce the OFV size by one bit per counter. While in theory this operation requires to check all the counter values, in practice this operation is simpler if a simple counter structure keeps track of the bit-usage for the m counters. We show this optimization in the next Section. Shrinking and enlarging the OFV result in the same DCF rebuild operation with an asymptotic cost of $O(m)$.

3.3 Delayed OFV Shrinking

The main difference between the rebuild due to insertion and the rebuild due to deletion is that while the former is required in order to avoid counter saturation, the latter is optional and may be delayed in order to avoid unstable situations of consecutive delete/insert operations that could result in excessive OFV rebuilds.

Therefore, we introduce a threshold between values 2^{x+y-2} and $2^{x+y-1}-1$. We define such threshold as $T = 2^{x+y-2}+(2^{x+y-1}-2^{x+y-2}) \times \lambda$, where λ ranges from 0 to 1. Hence, when decreasing entry j in the DCF by one, and being V_j the associated value to the counter $\langle OF_j, C_j\rangle$, then, if $V_j < T$, we rebuild the OFV when all the counters in the OFV $\forall j : j = 1..m : V_j < T$.

Threshold Maintenance

As mentioned in Section 3.2, in order to avoid checking all the counter values for underflow, we perform an optimization and keep an overflow counter structure. We name l, the *overflow level* of any counter in the DCF structure. The overflow level represents the number of bits used in the overflow counter and therefore, it may have a value between 0 and y. Consequently, an entry j in the DCF has overflow level $l > 0$ if its overflow counter OF_j has a value between $2^{l-1} < OF_j \leq 2^l - 1$. It has an overflow level $l = 0$ if $OF_j = 0$. We arrange the different *overflow level* counters into a structure called the *Counter Level* (CL).

In order to use the threshold T as described before, we keep two counters per level in the CL: (1) a counter for the number of OF_j counters with value less or equal than the threshold for the level they belong, *i.e.* counters below $2^{x+y-2}+(2^{x+y-1}-2^{x+y-2}) \times \lambda$ for values with a level greater than 0, and below $(2^x - 1) \times \lambda$ for counters in level 0; and (2) a counter for the number of OF_j counters with a value equal or larger than the threshold. The level l of a counter stored in position j in the DCF structure, is calculated as:

$$\begin{cases} 0 & \text{if } OF_j = 0; \\ \lfloor log(OF_j) \rfloor + 1 & \text{otherwise.} \end{cases}$$

Then, whenever a data element s is either inserted or deleted and values $V_{h_i(s):i=1..k}$ either incremented or decremented, we must update the corresponding level counters.

Note that the storage needed by the CL structure is negligible: we only need $2 \times (y+1)$ counters. Also, the updating process is of asymptotic cost $O(1)$, we only perform simple additions subtractions and comparisons.

Figure 3 shows an example of how the CL structure is used for delayed OFV rebuilds resulting from data deletion operations. In this example we show a DCF structure with eight counters, $m = 8$, and $\lambda = 0.5$. CBFV has $x = 4$ bits per counter, and after several inserts the OFV has $y = 2$ bits. The decimal values of each counter are shown only for clarity of the example, as they have the same information as their corresponding pair of counters. Initially, on the left side, the values for each counter are $\{0, 2, 7, 31, 9, 28, 17, 60\}$, thus, the counters in position 0, 1, 2, and 4 are of level $l = 0$ (with three counters below the threshold and one above), those in positions 3, 5, and 6 are of level $l = 1$ (with one counter below the threshold and two above), and the last counter is of level $l = 2$ (with the counter above the threshold). In the central part of Figure 3, we can see that, after several deletes, level 2 counters and the level 1 counter above the threshold reach zero. Therefore, as all counters in the DCF are below the threshold, we can rebuild the OFV by deleting one bit per position and consequently reducing the size of the whole DCF structure.

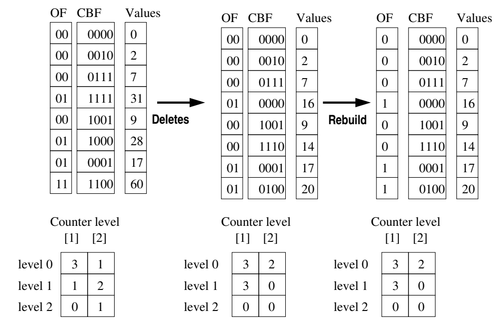

Figure 3: Counter Level and OFV rebuild in DCF.

Choosing the Optimal Threshold λ.

Rebuilds are the most costly operations performed in the DCF structure. Thus, we define as the optimal threshold for DCF, the value of λ which minimizes the number of rebuilds over time.

First, we must determine which are the situations worth to rebuild the structure in case of deletions. For this, we define the ratio $R = n_{inserts}/n_{deletes}$, which is a metric that gives an indication of how the number of insert operations evolve comparing to the number of delete operations over time.

Figure 4 shows the evolution of DCF in terms of the number of rebuilds over time. The chart includes three lines, representing three different scenarios, *T1*, *T2*, and *T3*, each one representing a different ratio of R: T1 for $R > 1$, T2 for $R \simeq 1$, and T3 for $R < 1$.

The results in this chart represent the average number of rebuilds for ten executions using different λ values ranging from 0.1 to 1.0. The first time steps for all scenarios consist of insert operations for the M data elements. After having populated the data set, each time step is composed of both insert and delete operations. The data inserted follows a Zipfian distribution, where the skew defined by θ is randomly selected from a range between 0 and 2, *i.e.* $0 \leq \theta \leq 2$ [2]. Items inserted are also randomly selected for deletion during the delete operations.

From Figure 4 we can identify the following behavior:

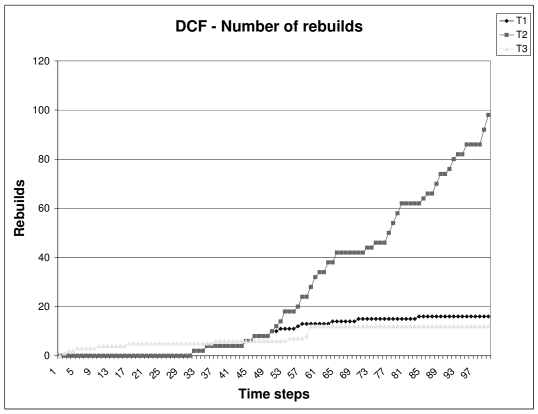

Figure 4: Number of rebuilds in DCF.

- T1 shows, in average for any λ, the number of rebuilds occurred when R grows as time passes. In this case, we are always increasing the number of useless rebuilds caused by deletions, *i.e.* in a short time, we may have a rebuild caused by an insertion, because insertions are more frequent.
- T2 and T3 show that, in average for any λ, the number of rebuilds does not increase when $R \simeq 1$ and $R < 1$ respectively. Hence for T2 and T3 scenarios there may exist a value of λ which minimizes the number of rebuilds, and improves the memory usage of the DCF.

Figure 5 shows the results for different values of $R < 1$, that are the values of λ that minimize the number of rebuilds. The plot shows that for ratios $R \leq 0.6$, where $n_{deletes} \gg n_{inserts}$, independently of the λ we choose, we always have a constant number of rebuilds. However, for $R > 0.6$, the value set for λ becomes important, and values of $\lambda \simeq (1 - R)$ are those that minimize the number of rebuilds. When the incoming data is highly skewed, *i.e.* inserts and deletes often affect the same counters, the number of rebuilds is more sensitive to the selected threshold value. We can see that for $\lambda = (1 - R)$ we have the least number of rebuilds.

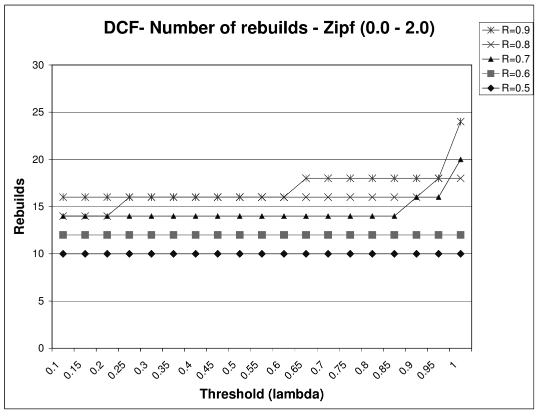

Figure 5: Number of rebuilds in DCF for different λ threshold values and insert-to-delete ratio for Zipfian distributions with θ between 0-2.

4. EXPERIMENTAL RESULTS

For the evaluation of our proposed DCF structures, we have implemented the DCF and SBF representations of the Counting Bloom filters in C language and compiled the programs using full optimization (-O3). The implementation of the SBF has followed the exact specifications given in [4] and detailed in Section 2. We run our tests on a 750MHz IBM PowerPC_RS64-IV processor with 16GB of main memory. The Operating System is AIX version 5.1.

The data we use is based on a total of n distinct values with multiplicities that are inserted and deleted from a data structure. The total number of data elements is $M = av * n$, where av is the average number of operation multiplicities per distinct value. Our experiments consist of performing M consecutive data insert operations, followed by M consecutive data delete operations, picked from the n distinct values in the data set following a Zipfian distribution [2], where the skew is defined by the parameter θ that ranges from 0 to 2. Values for $\theta \simeq 0$ represent uniformly distributed data, while values $\theta \simeq 2$ represent highly skewed data. We use integers as data values.

In order to analyze the behavior of DCF in detail, we focused on a set of experiments aimed at evaluating the performance in terms of access time, memory usage and impact of rebuild operations for the DCF structure, in comparison to the SBF approach.

The number of counters m both for DCF and SBF depends on the fraction of false positives F_p, the number of distinct values n, and the number of hash functions used k. We set by default $F_p = 0.05$ and $k = 3$. The number of distinct values n may change, and is specified for each experiment we describe. Also, the total amount of values M may vary for the different tests.

One metric used to measure the accuracy of the filter is the *accurate representation* of a data element. This is defined as follows: a data element s has an *accurate representation* if the minimum value stored in the k counters matches the real number of times s has been inserted in the data set.

Read, Write, and Rebuild Time

Figure 6 shows the average time to perform a read, a write, and a rebuild operation for both SBF and DCF structures for different data set sizes. The results are shown for four different setups where the number of distinct values n is 1000, 10000, 100000, and 1000000, respectively. For each test, we consider uniform data and a total amount of $M = 100 \times n$ values were first randomly inserted, and then, randomly deleted. The results are expressed in μseconds and the y-axis is represented using a logarithmic scale. Each value in the chart represents the average, for each different operation, over all the instances of each operation during the complete execution of the test.

From the results in Figure 6 it is possible to conclude that DCF is more efficient in terms of the time to access the structure, either read or write operations, compared to the SBF structure. The cost to locate a counter in the DCF is very efficient compared to the SBF lookup that requires traversing the index structures.

Note that as the number of distinct values increases, both techniques keep the read and write times constant. Another important fact is that the rebuild operations, as expected, prove to be the most costly operations. DCF is, in average, slightly faster than SBF and its execution time shows to grow with a similar rate as SBF. How-

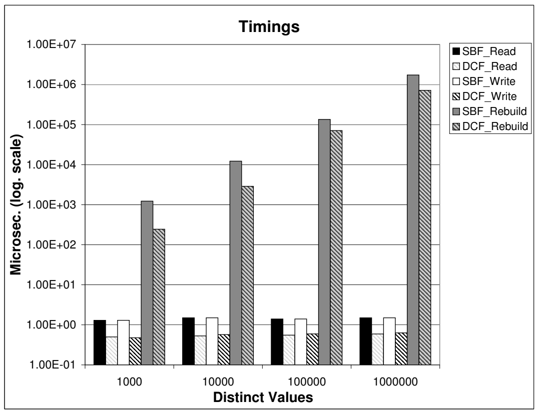

Figure 6: Average time for read, write and rebuild operations for SBF and DCF.

ever, note that DCF requires a total number of rebuilds which is orders of magnitude smaller than SBF. For instance, for $n = 10$K the number of rebuilds for DCF is 8 while that for SBF is more than 45000, and for $n = 1$M the number of rebuilds for DCF just grows to 9 while that for SBF grows to more than 4.5 million rebuilds. This comes from the fact that a DCF rebuild for one counter, automatically rebuilds the rest of the counters, growing the OFV data structure by one bit vector of size m. On the other hand, each SBF rebuild involves only one counter at a time.

Memory Usage

Figure 7 shows results obtained through several static executions varying the skew (Zipfian with $0 \le \theta \le 2$) and the total amount of values M. Values are first randomly inserted, and then, randomly deleted. In that chart we show the ratio $R = \frac{Mem_{SBF}}{Mem_{DCF}}$. Notice that while Mem_{DCF} varies dynamically, during the execution of the test, for this static analysis we show only the maximum memory size that DCF requested for the complete execution.

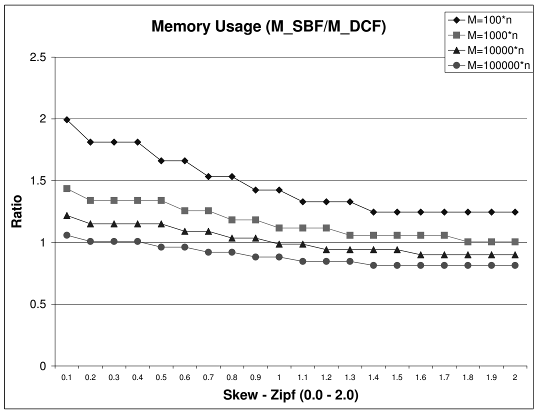

Figure 7: SBF-to-DCF memory ratio for different data distributions and data set sizes.

The chart in Figure 7 shows that, as expected, in the most usual situations DCF needs less memory than SBF. There are two extreme cases: one for non-skewed, almost uniform, data distributions (Zipfian with $\theta = 0.1$), where DCF uses less than the half of the space used by SBF; and another one for highly skewed data or when there is an extremely large multiplicity of elements for a single value, where SBF consumes less memory than DCF.

	Memory (bytes)	Accuracy (%)	Access Time (μsec)
SBF	17280	90.1	1.3
DCF	20125	90.1	0.5
DCF bounded	16950	87.6	0.5

Table 2: SBF and DCF for large number of repeated values.

For those cases where DCF does not behave better in terms of memory usage, we want to understand how the DCF approach would behave if we limited its memory to the maximum memory used by SBF.

Large number of repeated values with limited memory

Table 2 shows the results of the execution for $n = 1000$ distinct values and $M = 10^9 = 10^6 \times n$ uniformly distributed data elements. On the one hand, we observe that without memory limitations, DCF would use 14% more memory than SBF, achieving the same accuracy and being more than two times faster than SBF in access time. On the other hand, if we have a limited amount of memory, forcing DCF to use the same memory as SBF only decreases its accuracy in 2.5%. However, the access time in this case is still more than two times faster than that of SBF.

Execution Time

Figure 8 shows execution time and number of rebuilds (logarithmic scale) for executions where we vary the degree of skewed data from $0 \le \theta \le 2$. Each execution consists of the insertion and later deletion of $M = 100 \times n$ items for $n = 1000$ distinct values. All values are inserted first, filling the structure, and then, randomly deleted until the structure becomes empty.

As it is possible to observe from the results in Figure 8, DCF clearly outperforms SBF independently of the incoming data distribution. The gap between both approaches is smaller as the data are more skewed. This fact is expected as for skewed data, the number of rebuilds performed by the SBF decreases, while at the same time the opposite effect happens for CBF. In Figure 9 we show the percentage of the total execution time spent in rebuilding the structure. It is possible to observe for SBF most of the time is spent rebuilding the structure, due to the high number of rebuilds. In contrast, the time that DCF spends in rebuilds is minimum. Although a rebuild for DCF is costly, this approach performs fewer rebuilds during the complete execution. Consequently, even for highly skewed data, DCF spends almost all of its execution time in the read and write operations.

Overall, DCF is faster for the read and write operations. Moreover, although the rebuild operation is costly, it is only performed a few times during the complete execution which results in a clear benefit compared with SBF.

5. CONCLUSIONS

In this paper we propose a new representation of the Counting Bloom Filters to cope with inserts and deletes in multisets over

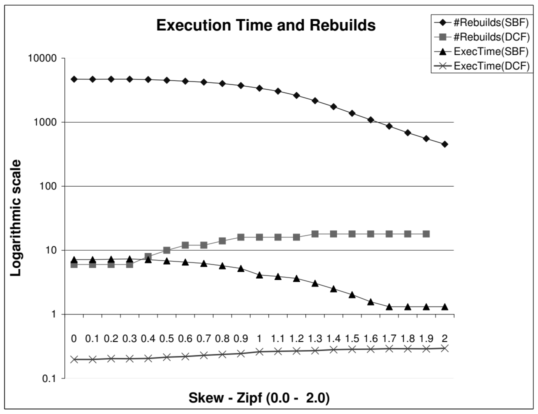

Figure 8: Total execution time.

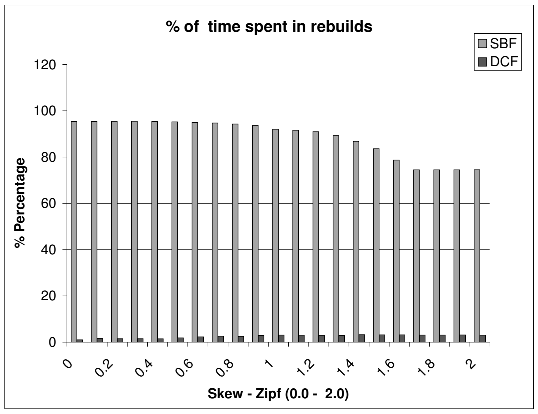

Figure 9: Percentage of time spent in rebuilding the structure.

time: the Dynamic Count Filters (DCF). Our data structure borrows the qualities of previous proposals. On the one hand, it has the fast access times of the Counting Bloom Filters (CBF). On the other hand, it has the adaptivity to changing data patterns of the Spectral Bloom Filters (SBF).

Dynamic Count Filters also show other interesting properties. First, in general, DCF uses a smaller amount of memory than SBF for a fixed amount of counters. However, although in extreme cases (very large number of replicated values) DCF uses a larger amount of memory than SBF, the gains obtained in execution time (about 2 times faster than SBF), make Dynamic Count Filters worth the small additional cost. Second, for a fixed amount of memory, DCF may include more counters and is faster. Third, the total cost for rebuilding DCF is significantly lower than that for SBF.

Overall, we can claim that Dynamic Count Filters are a data structure to be taken into account in many practical situations for their fast access times and for their ability to adapt to the dynamic evolution of data and to situations with small amounts of memory available.

6. ACKNOWLEDGEMENTS

UPC authors thank the Computer Architecture Department, and IBM for the continuous support to our research through CAS grants and fellowships. All the authors thank the HPC-Europa programme, funded by the European Commission's Research Infrastructures activity under contract RII3-CT-2003-506079.

7. REFERENCES

[1] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. *In Proceedings of the IEEE Infocom Conference*, 1999.

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

[4] Saar Cohen and Yossi Matias. Spectral bloom filters. *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 241–252, 2003.

[5] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. Longest prefix matching using bloom filters. *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212, 2003.

[6] L. Fan, , P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE Trans on Networking*, 8(3):281–293, 2000.

[7] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing Iceberg Queries Efficiently. *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 299–310, 1998.

[8] Jonathan Ledlie, Laura Serban, and Dafina Toncheva. Scaling filename queries in a large-scale distributed file systems. Research Report TR-03-02, Harvard University, January 2002.

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

[10] James K. Mullin. A second look at bloom filters. *Commun. ACM*, 26(8):570–571, 1983.

[11] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Commun. ACM*, 32(10):1237–1239, 1989.

[12] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. Bloom histogram: Path selectivity estimation for xml data with updates. *VLDB'04: Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 240–251, 2004.